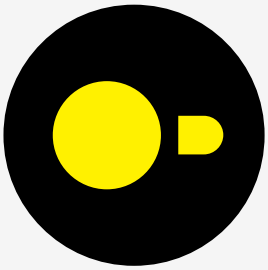
DuckDB Documentation

DuckDB version 2.0.0
Generated on 2026-10-26 at 12:22 UTC



Contents

Contents i

Summary 1

What's New in the Docs for 2.0 3

Connect 7

Connect 8

Concurrency 9

Data Import and Export 10

Importing Data 11

Data Sources 13

CSV Files 14
- CSV Import . . . 14
- CSV Auto Detection . . . 19
- Reading Faulty CSV Files . . . 23
- CSV Import Tips . . . 28

I/O 30

JSON Files 33
- JSON Overview . . . 33
- Creating JSON . . . 35
- Loading JSON . . . 35
- Writing JSON . . . 42
- JSON Type . . . 42
- JSON Processing Functions . . . 43
- JSON Format Settings . . . 53
- Installing and Loading the JSON Extension . . . 55
- SQL to/from JSON . . . 55
- Caveats . . . 58

Multiple Files 59
- Reading Multiple Files . . . 59
- Combining Schemas . . . 62

Parquet Files 64
- Reading and Writing Parquet Files . . . 64
- Querying Parquet Metadata . . . 70

Parquet Encryption . . . 74
Parquet Tips . . . 75

Partitioning **77**
Hive Partitioning . . . 77
Partitioned Writes . . . 78

Appender **82**

INSERT Statements **85**

Lakehouse Formats **86**

Client APIs **89**

Client Overview **90**

ADBC Client **91**

C **101**
Overview . . . 101
Startup & Shutdown . . . 101
Configuration . . . 107
Query . . . 109
Data Chunks . . . 117
Vectors . . . 119
Values . . . 133
Types . . . 156
Prepared Statements . . . 174
Appender . . . 179
Table Functions . . . 189
Replacement Scans . . . 200
Complete API . . . 201

C++ API **342**

CLI **345**
Command Line Client . . . 345
Command Line Arguments . . . 351
Dot Commands . . . 354
Output Formats . . . 364
Agent Mode . . . 367
Editing . . . 368
Friendly CLI . . . 370
Safe Mode . . . 371
Autocomplete . . . 372
Syntax Highlighting . . . 372
Known Issues . . . 374

Go **375**
Go Client . . . 375
Connect . . . 378
Import Data . . . 380
Run Queries . . . 382

Handle Results . . . 385
Write User Defined Functions . . . 387
Profile and Monitor . . . 390
Troubleshoot . . . 391

Java (JDBC) 394
Java (JDBC) Client . . . 394
Connect . . . 395
Import Data . . . 403
Run Queries . . . 406
Handle Results . . . 410
Write User Defined Functions . . . 413
Profile and Monitor . . . 420
Deploy as Native Image . . . 423
Troubleshoot . . . 424

Node.js (Neo) 427
Node.js Client (Neo) . . . 427

ODBC 445
ODBC API Overview . . . 445
ODBC API on Linux . . . 445
ODBC API on Windows . . . 446
ODBC API on macOS . . . 450
ODBC Configuration . . . 451

Python 453
Python API . . . 453
Data Ingestion . . . 457
Appender . . . 460
Conversion between DuckDB and Python . . . 463
Python DB API . . . 469
Relational API . . . 473
Python Function API . . . 571
Types API . . . 576
Expression API . . . 580
Spark API . . . 586
Python Client API . . . 587
Troubleshooting . . . 587
Migrating Python Code to DuckDB 2.0 . . . 590

R Client 592

Rust 595
Rust Client . . . 595
Connect . . . 599
Import Data . . . 600
Run Queries . . . 602
Handle Results . . . 605
Write User Defined Functions . . . 606
Profile and Monitor . . . 609
Troubleshoot . . . 611

Wasm 613
DuckDB Wasm Client . . . 613

Instantiate . . . 615
Import Data . . . 619
Run Queries . . . 622
Load Extensions . . . 624
Deploy . . . 627
Troubleshoot . . . 629

Tertiary Clients **631**
Tertiary Clients . . . 631
Julia Client . . . 631
PHP Client . . . 634
PHP Client (PDO) . . . 637
Swift Client . . . 641

SQL **644**

SQL Introduction **645**

Statements **653**
Statements Overview . . . 653
ANALYZE Statement . . . 653
ALTER DATABASE Statement . . . 653
ALTER TABLE Statement . . . 653
ALTER VIEW Statement . . . 658
ATTACH and DETACH Statements . . . 659
CALL Statement . . . 664
CHECKPOINT Statement . . . 665
COMMENT ON Statement . . . 666
CONNECT and DISCONNECT Statements . . . 667
COPY Statement . . . 668
CREATE MACRO Statement . . . 678
CREATE SCHEMA Statement . . . 684
CREATE SECRET Statement . . . 685
CREATE SEQUENCE Statement . . . 685
CREATE TABLE Statement . . . 689
CREATE VIEW Statement . . . 693
CREATE TRIGGER Statement . . . 694
CREATE TYPE Statement . . . 696
DELETE Statement . . . 697
DESCRIBE Statement . . . 698
DROP Statement . . . 698
EXPORT and IMPORT DATABASE Statements . . . 700
INSERT Statement . . . 701
LOAD / INSTALL Statements . . . 707
MERGE INTO Statement . . . 708
PIVOT Statement . . . 711
Profiling Queries . . . 717
PREPARE, EXECUTE, and DEALLOCATE Statements . . . 718
SELECT Statement . . . 719
SET and RESET Statements . . . 722
SET VARIABLE and RESET VARIABLE Statements . . . 723
SHOW, SHOW DATABASES, and SHOW SCHEMAS Statements . . . 726
SUMMARIZE Statement . . . 727

Transaction Management . . . 727
UNPIVOT Statement . . . 729
UPDATE Statement . . . 734
UPDATE EXTENSIONS . . . 736
USE Statement . . . 737
VACUUM Statement . . . 738

Query Syntax **739**
SELECT Clause . . . 739
FROM and JOIN Clauses . . . 741
WHERE Clause . . . 751
GROUP BY Clause . . . 752
GROUPING SETS . . . 753
HAVING Clause . . . 757
ORDER BY Clause . . . 758
LIMIT and OFFSET Clauses . . . 760
SAMPLE Clause . . . 761
Unnesting . . . 762
WITH Clause . . . 765
WINDOW Clause . . . 775
QUALIFY Clause . . . 775
VALUES Clause . . . 777
FILTER Clause . . . 777
Set Operations . . . 780
Prepared Statements . . . 783

Data Types **786**
Data Types . . . 786
Array Type . . . 788
Bitstring Type . . . 790
Blob Type . . . 791
Boolean Type . . . 792
Date Types . . . 793
Enum Data Type . . . 794
Geometry Data Type . . . 797
Interval Type . . . 804
List Type . . . 807
Literal Types . . . 809
Map Type . . . 812
NULL Values . . . 814
Numeric Types . . . 815
Struct Data Type . . . 819
Text Types . . . 825
Time Types . . . 826
Timestamp Types . . . 827
Time Zone Reference List . . . 832
Union Type . . . 849
Typecasting . . . 851
Variant Type . . . 855

Expressions **858**
Expressions . . . 858
CASE Expression . . . 858

Casting . . . 860
Collations . . . 861
Comparisons . . . 863
IN Operator . . . 865
Logical Operators . . . 866
Star Expression . . . 867
Subqueries . . . 872
TRY Expression . . . 876

Functions **878**
Functions . . . 878
Aggregate Functions . . . 879
Array Functions . . . 895
Bitstring Functions . . . 897
Blob Functions . . . 901
Date Format Functions . . . 904
Date Functions . . . 906
Date Part Functions . . . 911
Enum Functions . . . 918
Geometry Functions . . . 919
Interval Functions . . . 921
Lambda Functions . . . 925
List Functions . . . 929
Map Functions . . . 954
Nested Functions . . . 958
Numeric Functions . . . 958
Pattern Matching . . . 971
Regular Expressions . . . 974
Struct Functions . . . 979
Text Functions . . . 982
Time Functions . . . 1007
Timestamp Functions . . . 1009
Timestamp with Time Zone Functions . . . 1018
Union Functions . . . 1028
Utility Functions . . . 1029
Variant Functions . . . 1042
Window Functions . . . 1046

Constraints **1057**

Indexes **1059**

Meta Queries **1061**
Information Schema . . . 1061
DuckDB_% Metadata Functions . . . 1066

DuckDB's SQL Dialect **1082**
Overview . . . 1082
Indexing . . . 1082
Friendly SQL . . . 1082
Keywords and Identifiers . . . 1086
Order Preservation . . . 1089
PostgreSQL Compatibility . . . 1090
SQL Quirks . . . 1095

PEG Parser **1098**

Samples **1100**

Configuration **1103**

Configuration **1104**

Pragmas **1118**

Secrets Manager **1128**

Extensions **1131**

Extensions **1132**

Installing Extensions **1135**

Advanced Installation Methods **1138**

Extension Distribution **1139**

Extension Repositories **1142**

Community Extensions **1144**

Versioning of Extensions **1145**

Troubleshooting of Extensions **1149**

Core Extensions **1151**

Core Extensions **1152**

AutoComplete Extension **1154**

Avro Extension **1156**

AWS Extension **1158**

Azure Extension **1163**

Delta Extension **1172**

DuckLake **1179**

Encodings Extension **1183**

Excel Extension **1211**

Full-Text Search Extension **1215**

httpfs (HTTP and S3) **1219**

httpfs Extension for HTTP and S3 Support . . . 1219
HTTP(S) Support . . . 1219
Hugging Face Support . . . 1221
S3 API Support . . . 1223
S3 and AWS Authentication . . . 1228

Legacy Authentication Scheme for S3 API . . . 1230

Iceberg **1232**
Iceberg Extension . . . 1232
Writing to Iceberg . . . 1234
Iceberg Functions . . . 1238
Iceberg Options . . . 1245
Catalogs . . . 1250
Troubleshooting . . . 1256

ICU Extension **1258**

inet Extension **1259**

Lance Extension **1263**

Legacy Parser Extension **1266**

MotherDuck Extension **1267**

MySQL Extension **1268**

ODBC **1275**
ODBC Extension . . . 1275
ODBC Extension Functions . . . 1279

Quack Extension **1288**

PostgreSQL **1289**
PostgreSQL Extension . . . 1289
PostgreSQL Extension and the Secret Manager . . . 1296
PostgreSQL Extension Connection Pool . . . 1300
PostgreSQL Extension Functions . . . 1301

Sirius Extension **1307**

Spatial **1308**
Spatial Extension . . . 1308
Spatial Functions . . . 1308
R-Tree Indexes . . . 1361
GDAL Integration . . . 1365

SQLite Extension **1366**

SQLSmith Extension **1372**

TPC-DS Extension **1373**

TPC-H Extension **1375**

UI Extension **1378**

Unity Catalog Extension **1381**

Vortex Extension **1386**

Vector Similarity Search Extension **1387**

Quack Remote Protocol **1391**

Quack Remote Protocol **1392**

Reference **1397**

Security **1403**

Setup **1407**
Quack Setup . . . 1407
Securing Quack with a Reverse Proxy . . . 1407
Deploying Quack . . . 1409
Quack on WebAssembly . . . 1411

Troubleshooting Quack **1413**

Guides **1415**

Guides **1416**

Data Viewers **1420**
Tableau – A Data Visualization Tool . . . 1420
CLI Charting with YouPlot . . . 1423

Database Integration **1426**
Database Integration . . . 1426
MySQL Import . . . 1426
PostgreSQL Import . . . 1427
SQLite Import . . . 1427
Amazon RDS with IAM Authentication . . . 1428

File Formats **1430**
File Formats . . . 1430
CSV Import . . . 1430
CSV Export . . . 1430
Directly Reading Files . . . 1430
Directly Read DuckDB Databases . . . 1432
Excel Import . . . 1433
Excel Export . . . 1435
JSON Import . . . 1436
JSON Export . . . 1436
Parquet Import . . . 1437
Parquet Export . . . 1438
Querying Parquet Files . . . 1438
File Access with the file: Protocol . . . 1438

Meta Queries **1439**
Describe . . . 1439
EXPLAIN: Inspect Query Plans . . . 1440
EXPLAIN ANALYZE: Profile Queries . . . 1441
List Tables . . . 1442
Summarize . . . 1443
DuckDB Environment . . . 1445

Network and Cloud Storage **1447**
Network and Cloud Storage . . . 1447
HTTP Parquet Import . . . 1447
HTTP CSV Import . . . 1447
S3 Parquet Import . . . 1448
S3 Parquet Export . . . 1449
S3 Iceberg Import . . . 1450
S3 Express One . . . 1450
Google Cloud Storage Import . . . 1452
Cloudflare R2 Import . . . 1452
Attach to a DuckDB Database over HTTPS or S3 . . . 1453
Share a DuckDB Database That Only Stores Views . . . 1454
Fastly Object Storage Import . . . 1455
SeaweedFS Import . . . 1455
Tigris Import . . . 1456

ODBC **1458**
ODBC 101: A Duck Themed Guide to ODBC . . . 1458

Performance **1465**
Performance Guide . . . 1465
Environment . . . 1465
Data Import . . . 1467
Schema . . . 1467
Indexing . . . 1469
Join Operations . . . 1471
File Formats . . . 1472
Tuning Workloads . . . 1474
My Workload Is Slow . . . 1478
Out-of-Memory Issues . . . 1478
Benchmarks . . . 1479
Working with Huge Databases . . . 1480

Python **1481**
Installing the Python Client . . . 1481
Executing SQL in Python . . . 1481
Jupyter Notebooks . . . 1482
marimo Notebooks . . . 1486
SQL on Pandas . . . 1490
Import from Pandas . . . 1490
Export to Pandas . . . 1490
Import from Numpy . . . 1491
Export to Numpy . . . 1491
SQL on Apache Arrow . . . 1492
Import from Apache Arrow . . . 1494
Export to Apache Arrow . . . 1494
Relational API on Pandas . . . 1495
Multiple Python Threads . . . 1496
Integration with Ibis . . . 1498
Integration with Polars . . . 1507
Integration with PyTorch . . . 1508
Using fsspec Filesystems . . . 1509

SQL Editors **1511**
Beekeeper Studio SQL Editor . . . 1511
DBeaver SQL IDE . . . 1516
DBine SQL Editor . . . 1526
LibreDB Studio SQL Editor . . . 1528
VisuaLeaf SQL Editor . . . 1537

SQL Features **1543**
AsOf Join . . . 1543
Full-Text Search . . . 1546
Graph Queries . . . 1547
query and query_table Functions . . . 1549
Merge Statement for SCD Type 2 . . . 1551
Timestamp Issues . . . 1555

Snippets **1557**
Create Synthetic Data . . . 1557
Dutch Railway Datasets . . . 1560
Sharing Macros . . . 1562
Analyzing a Git Repository . . . 1563
Importing Duckbox Tables . . . 1567
Copying an In-Memory Database to a File . . . 1568
Calculating a Database Checksum . . . 1568

Troubleshooting **1570**
Command Line . . . 1570
Crashes . . . 1570
Out of Memory Errors . . . 1572

Glossary of Terms **1574**

Browsing Offline **1575**

Operations Manual **1576**

Overview **1577**

Migrating to DuckDB 2.0 **1578**

Backup Extension Repositories **1582**

DuckDB's Footprint **1583**
Files Created by DuckDB . . . 1583
Gitignore for DuckDB . . . 1584
Reclaiming Space . . . 1584

Installing DuckDB **1586**
Install Script . . . 1586
Requirements . . . 1586

Logging **1588**
Logging . . . 1588

HTTP User-Agent **1593**

Securing DuckDB **1594**
Securing DuckDB . . . 1594
Embedding DuckDB . . . 1598
Securing Extensions . . . 1598

Non-Deterministic Behavior **1601**

Limits **1603**

DuckDB Docker Container **1604**

Development **1605**

Overview **1606**

DuckDB Repositories **1607**

Release Cycle **1608**

Metrics **1614**

Profiling **1619**

Building DuckDB **1622**
Building DuckDB from Source . . . 1622
Building Configuration . . . 1624
Building Extensions . . . 1626
Statically Linking Extensions . . . 1628
Android . . . 1631
OpenHarmony . . . 1632
Linux . . . 1633
macOS . . . 1635
Raspberry Pi . . . 1637
Windows . . . 1637
Python . . . 1638
R . . . 1642
Troubleshooting . . . 1643
Unofficial and Unsupported Platforms . . . 1643

Benchmark Suite **1646**

Testing **1649**
Overview . . . 1649
sqllogictest Introduction . . . 1649
Writing Tests . . . 1651
Unit Tester Configuration . . . 1652
Debugging . . . 1656
Result Verification . . . 1657
Persistent Testing . . . 1659
Loops . . . 1660
Multiple Connections . . . 1661
Catch C/C++ Tests . . . 1662

Internal Errors **1664**

Internals **1665**

Overview of DuckDB Internals **1666**

Storage Versions and Format **1668**

Execution Format **1673**

jemalloc **1680**

Pivot Internals **1681**

Acknowledgments **1684**

Summary

This document contains DuckDB's official documentation and guides in a single-file easy-to-search form. If you find any issues, please report them as a GitHub issue. Contributions are very welcome in the form of pull requests. If you are considering submitting a contribution to the documentation, please consult our contributor guide.

Code repositories:

- DuckDB source code: github.com/duckdb/duckdb
- DuckDB documentation source code: github.com/duckdb/duckdb-web

What's New in the Docs for 2.0

This page tracks the documentation for the features introduced in DuckDB 2.0, organized around the themes of the DuckDB 2.0.0 release announcement. Each section links to the relevant documentation.

DuckDB as a Server: Quack and CONNECT

Since v1.5.3, DuckDB can run as a server via the Quack protocol, letting multiple clients connect to a shared instance over the network. DuckDB 2.0 ships a polished version of the `quack` extension, which provides the remote protocol, and adds the new `CONNECT` and `DISCONNECT` statements to open and close connections from SQL. The command line client can start a server and attach to one directly.

- Quack extension
- `CONNECT` and `DISCONNECT` statements
- CLI `-serve` and `-connect` arguments
- Securing Quack with a reverse proxy
- Remote pushdown to PostgreSQL and MySQL

VARIANT Becomes a First-Class Citizen

The `VARIANT` type stores semi-structured, schema-less data such as JSON while keeping the performance of columnar execution. A family of `variant_*` functions constructs and inspects variant values.

- Variant type
- Variant functions

Triggers

DuckDB 2.0 added trigger support, so you can run actions automatically in response to table changes. This is useful for building audit tables and enforcing invariants.

- `CREATE TRIGGER` statement

SQL Dialect Additions

DuckDB 2.0 extended the SQL dialect in several directions, from new join types to data-modifying CTEs and a friendlier variable syntax.

- Explicit join type syntax, including nearest (`NEAREST`) joins
- Data-modifying CTEs, including `COPY ...` blocks in CTEs
- Nested schemas
- The `$<variable>` variable shorthand, including session variables as default parameter values
- JSON mutation functions
- Grouping by `UNNEST`
- The SQL-standard `FETCH FIRST` syntax
- Well-defined semantics for multiple matching source rows in `MERGE INTO` and `UPDATE ... FROM`
- `SECURE VIEW`
- `SHOW` statement
- Setting a sequence value
- `regexp_extract` additions

Asynchronous I/O

DuckDB 2.0 issues reads in the background and keeps multiple requests in flight, which speeds up queries that read from object storage such as S3. The behavior is controlled by the `async_threads` and read-ahead settings.

- Asynchronous I/O overview
- Thread pools and read-ahead
- Asynchronous I/O configuration settings

Faster Queries Across the Board

DuckDB 2.0 included optimizer and engine improvements, including faster recursive CTEs and better row-group pruning.

- Recursive CTEs and the blog post on how DuckDB runs them faster
- Min-max indexes (zonemaps) used for row-group pruning

Storage Format v2.0

New database files use the v2.0.0 storage format by default. The format adds lazy metadata loading, better compression and corruption validation. DuckDB versions before 2.0 cannot read files in the new format, so a migration guide covers upgrading existing deployments.

- Default storage version
- Storage version table
- Migrating existing deployments: see the DuckDB 2.0 migration guide

A Brand New SQL Parser

DuckDB 2.0 introduced a PEG-based SQL parser that replaces the PostgreSQL-derived one and allows extensions to contribute grammar.

- PEG parser
- Legacy parser extension for switching back to the old parser

Timezones, Calendars and Collations Without ICU

DuckDB 2.0 replaces the ICU library with a smaller, faster native implementation of timezone, calendar and collation support in the `icu` extension.

- ICU extension

Write Extensions Once, Host Them Yourself

DuckDB 2.0 shipped a revamped C API (C API v2), which is not yet stable. You can also register and host your own extension repositories or statically link extensions into custom builds.

- C API v2
- Extension repositories
- Statically linking extensions: see Statically Linking Extensions

Core Extension Updates

Several core extensions shipped new features alongside DuckDB 2.0.

- DuckLake extension, the reference implementation of DuckLake 1.1
- Iceberg extension `ATTACH` options for REST catalogs
- Creating Delta tables, time travel and `VARIANT` columns in the Delta extension

Deprecations

DuckDB 2.0 disables or phases out a few features, which will be removed in future releases.

- The single-arrow lambda syntax, now disabled by default
- macOS x86_64 (`osx_amd64`), for which community extensions are no longer built

Other Documentation Updates

Additional 2.0 features documented across the site, grouped by area.

Data Import and Export

- Writing GeoJSON and loading GeoJSON
- The Parquet `schema` parameter and new Parquet writing options
- The `PARTITION_PATH` copy option for computing custom partition subdirectories
- The `ORDER_BY` copy option for sorting rows before writing

Clients

- CLI agent mode, which produces compact, machine-readable output when the CLI is driven by an AI coding agent
- CLI command history: the `.history` command and the `shell_history()` table function
- Pretty-printed, highlighted `.schema` output with `--no-indent` to restore the single-line form
- The Python appender for inserting rows that Python code produces

Building

- A C++17 compiler is now required to build DuckDB

Connect

Connect

Connect or Create a Database

To use DuckDB, you must first create a connection to a database. The exact syntax varies between the client APIs but it typically involves passing an argument to configure persistence.

Since DuckDB 1.5.3, you can use the Quack protocol to run DuckDB as a server and set up a client-server connection. DuckDB 2.0 adds the CONNECT statement, which directs queries to a remote instance.

Persistence

DuckDB can operate in both persistent mode, where the data is saved to disk, and in in-memory mode, where the entire dataset is stored in the main memory.

> **Tip.** Both persistent and in-memory databases use spilling to disk to facilitate larger-than-memory workloads (i.e., out-of-core-processing).

Persistent Database

To create or open a persistent database, set the path of the database file, e.g., `my_database.duckdb`, when creating the connection. This path can point to an existing database or to a file that does not yet exist and DuckDB will open or create a database at that location as needed. The file may have an arbitrary extension, but `.db` or `.duckdb` are two common choices with `.ddb` also used sometimes.

Starting with v0.10, DuckDB's storage format is backwards-compatible, i.e., DuckDB is able to read database files produced by an older version of DuckDB. For example, DuckDB 0.10 can read and operate on files created by the previous DuckDB version, v0.9. Note that DuckDB versions before 2.0 cannot read database files created by DuckDB 2.0. For details, see the migration guide. For more details on DuckDB's storage format, see the storage page.

In-Memory Database

DuckDB can operate in in-memory mode. In most clients, this can be activated by passing the special value `:memory:` as the database file or omitting the database file argument. In in-memory mode, no data is persisted to disk, therefore, all data is lost when the process finishes.

Concurrency

Handling Concurrency

Single Process

In in-process mode, DuckDB has two configurable options for concurrency:

1. **Read-write mode:** one process can both read and write to the database.
2. **Read-only mode:** multiple processes can read from the database, but no processes can write (`access_mode = 'READ_ONLY'`).

When using read-write mode, DuckDB supports multiple writer threads using a combination of MVCC (Multi-Version Concurrency Control) and optimistic concurrency control (see Concurrency within a Single Process), but all within that single writer process. The reason for this concurrency model is to allow for the caching of data in RAM for faster analytical queries, rather than going back and forth to disk during each query. It also allows the caching of function pointers, the database catalog, and other items so that subsequent queries on the same connection are faster.

Concurrency Model within a Single Process DuckDB supports concurrency within a single process according to the following rules. As long as there are no write conflicts, multiple concurrent writes will succeed. Appends will never conflict, even on the same table. Multiple threads can also simultaneously update separate tables or separate subsets of the same table. Optimistic concurrency control comes into play when two threads attempt to edit (update or delete) the same row at the same time. In that situation, the second thread to attempt the edit will fail with a conflict error.

Multiple Processes

Writing to DuckDB's native database format from multiple processes is supported through the Quack remote protocol, which turns DuckDB into a client-server database. DuckDB 2.0 ships a polished version of Quack (first released in DuckDB 1.5.3) with substantial performance and stability improvements.

Alternatively, consider using the DuckLake format with PostgreSQL as the catalog database. By coordinating through a central PostgreSQL catalog, DuckDB instances can achieve concurrent read-writes on the same database. DuckLake 1.1, intended for production use, was released alongside DuckDB 2.0.

Optimistic Concurrency Control

DuckDB uses optimistic concurrency control, an approach generally considered to be the best fit for read-intensive analytical database systems as it speeds up read query processing. As a result any transactions that modify the same rows at the same time will cause a transaction conflict error:

```
Transaction conflict: cannot update a table that has been altered!
```

> **Tip.** A common workaround when a transaction conflict is encountered is to rerun the transaction.

Troubleshooting

File locks. DuckDB handles concurrent database access requests using file locks. Exercise extra caution when accessing a DuckDB database file in a shared directory (e.g., from different operating systems using different file systems or on network attached storage).

Data Import and Export

Importing Data

The first step to using a database system is to insert data into that system. DuckDB can directly connect to many popular data sources and offers several data ingestion methods that allow you to easily and efficiently fill up the database. On this page, we provide an overview of these methods so you can select which one is best suited for your use case.

`INSERT` Statements

`INSERT` statements are the standard way of loading data into a database system. They are suitable for quick prototyping, but should be avoided for bulk loading as they have significant per-row overhead.

```
INSERT INTO people VALUES (1, 'Mark');
```

For a more detailed description, see the page on the `INSERT` statement.

File Loading: Relative Paths

Use the configuration option `file_search_path` to configure to which "root directories" relative paths are expanded on. If `file_search_path` is not set, the working directory is used as the basis for relative paths.

File Formats

CSV Loading

Data can be efficiently loaded from CSV files using several methods. The simplest is to use the CSV file's name:

```
SELECT * FROM 'test.csv';
```

Alternatively, use the `read_csv` function to pass along options:

```
SELECT * FROM read_csv('test.csv', header = false);
```

Or use the COPY statement:

```
COPY tbl FROM 'test.csv' (HEADER false);
```

It is also possible to read data directly from **compressed CSV files** (e.g., compressed with gzip):

```
SELECT * FROM 'test.csv.gz';
```

DuckDB can create a table from the loaded data using the `CREATE TABLE ... AS SELECT` statement:

```
CREATE TABLE test AS
    SELECT * FROM 'test.csv';
```

For more details, see the page on CSV loading.

Parquet Loading

Parquet files can be efficiently loaded and queried using their filename:

```
SELECT * FROM 'test.parquet';
```

Alternatively, use the `read_parquet` function:

```
SELECT * FROM read_parquet('test.parquet');
```

Or use the COPY statement:

```
COPY tbl FROM 'test.parquet';
```

For more details, see the page on Parquet loading.

JSON Loading

JSON files can be efficiently loaded and queried using their filename:

```
SELECT * FROM 'test.json';
```

Alternatively, use the `read_json` function:

```
SELECT * FROM read_json('test.json');
```

Or use the COPY statement:

```
COPY tbl FROM 'test.json';
```

For more details, see the page on JSON loading.

Returning the Filename

Since DuckDB 1.3.0, the CSV, JSON and Parquet readers support the `filename` virtual column:

```
COPY (FROM (VALUES (42), (43)) t(x)) TO 'test.parquet';
SELECT *, filename FROM 'test.parquet';
```

Returning the File Index

The CSV, JSON and Parquet readers also support the `file_index` virtual column, which returns the zero-based index of the file that each row came from, in the order the files were read:

```
COPY (FROM (VALUES (42)) t(x)) TO 'a.parquet';
COPY (FROM (VALUES (43)) t(x)) TO 'b.parquet';
SELECT x, file_index FROM read_parquet(['a.parquet', 'b.parquet']);
```

x	file_index
42	0
43	1

Appender

In several APIs (C, C++, Go, Java, Julia, Node.js, Python and Rust), the Appender can be used as an alternative for bulk data loading. This class can be used to efficiently add rows to the database system without using SQL statements.

Data Sources

DuckDB supports several data sources, including file formats, network protocols, and database systems:

- AWS S3 buckets and storage with S3-compatible API
- Azure Blob Storage
- Blob files
- Cloudflare R2
- CSV
- Delta Lake
- DuckLake
- Excel
- httpfs
- Iceberg
- JSON
- Lance tables
- MySQL
- Parquet
- PostgreSQL
- SQLite
- Text files
- Vortex files

CSV Files

CSV Import

Examples

The following examples use the `flights.csv` file.

Read a CSV file from disk, auto-infer options:

```
SELECT * FROM 'flights.csv';
```

Use the `read_csv` function with custom options:

```
SELECT *
FROM read_csv('flights.csv',
    delim = '|',
    header = true,
    columns = {
        'FlightDate': 'DATE',
        'UniqueCarrier': 'VARCHAR',
        'OriginCityName': 'VARCHAR',
        'DestCityName': 'VARCHAR'
    });
```

Read a CSV from stdin, auto-infer options:

```
cat flights.csv | duckdb -c "SELECT * FROM read_csv('/dev/stdin')"
```

Read a CSV file into a table:

```
CREATE TABLE ontime (
    FlightDate DATE,
    UniqueCarrier VARCHAR,
    OriginCityName VARCHAR,
    DestCityName VARCHAR
);
COPY ontime FROM 'flights.csv';
```

Alternatively, create a table without specifying the schema manually using a `CREATE TABLE ... AS SELECT` statement:

```
CREATE TABLE ontime AS
    SELECT * FROM 'flights.csv';
```

We can use the FROM-first syntax to omit `SELECT *`.

```
CREATE TABLE ontime AS
    FROM 'flights.csv';
```

CSV Loading

CSV loading, i.e., importing CSV files to the database, is a very common, and yet surprisingly tricky, task. While CSVs seem simple on the surface, there are a lot of inconsistencies found within CSV files that can make loading them a challenge. CSV files come in many different varieties, are often corrupt, and do not have a schema. The CSV reader needs to cope with all of these different situations.

The DuckDB CSV reader can automatically infer which configuration flags to use by analyzing the CSV file using the CSV sniffer. This will work correctly in most situations, and should be the first option attempted. In rare situations where the CSV reader cannot figure out the

correct configuration it is possible to manually configure the CSV reader to correctly parse the CSV file. See the auto detection page for more information.

> Auto-detection (`auto_detect`) is enabled by default. Setting an option explicitly does not turn auto-detection off: the sniffer still detects every option you did not specify, and your explicit value overrides the sniffer only for that option. To disable auto-detection completely, set `auto_detect = false`.

Parameters

Below are parameters that can be passed to the `read_csv` function. Where meaningfully applicable, these parameters can also be passed to the COPY statement.

Name	Description	Type	Default
`all_varchar`	Skip type detection and assume all columns are of type VARCHAR. This option is only supported by the `read_csv` function.	BOOL	`false`
`allow_empty`	If the file pattern or file list matches no files, return an empty result (with a single `empty` column of type BOOL) instead of throwing an error. Not available in the COPY statement. Added in DuckDB 2.0.	BOOL	`false`
`allow_quoted_nulls`	Allow the conversion of quoted values to `NULL` values	BOOL	`true`
`auto_detect`	Auto detect CSV parameters.	BOOL	`true`
`auto_type_candidates`	Types that the sniffer uses when detecting column types. The VARCHAR type is always included as a fallback option. See example.	`TYPE[]`	default types
`buffer_size`	Size of the buffers used to read files, in bytes. Must be large enough to hold four lines and can significantly impact performance.	BIGINT	`16 * max_line_size`
`columns`	Column names and types, as a struct (e.g., `{'col1': 'INTEGER', 'col2': 'VARCHAR'}`). Using this option disables auto detection of the schema.	STRUCT	(empty)
`comment`	Character used to initiate comments. Lines starting with a comment character (optionally preceded by space characters) are completely ignored; other lines containing a comment character are parsed only up to that point.	VARCHAR	(empty)
`compression`	Method used to compress CSV files. By default this is detected automatically from the file extension (e.g., `t.csv.gz` will use gzip, `t.csv` will use `none`). Options are `none`, `gzip`, `zstd`.	VARCHAR	`auto`
`dateformat`	Date format used when parsing and writing dates.	VARCHAR	(empty)
`date_format`	Alias for `dateformat`; only available in the COPY statement.	VARCHAR	(empty)
`decimal_separator`	Decimal separator for numbers.	VARCHAR	`.`
`delim`	Delimiter character used to separate columns within each line, e.g., `,` `;` `\t`. The delimiter character can be up to 4 bytes, e.g., 🦆. Alias for `sep`.	VARCHAR	`,`
`delimiter`	Alias for `delim`; only available in the COPY statement.	VARCHAR	`,`

Name	Description	Type	Default
`escape`	String used to escape the `quote` character within quoted values.	`VARCHAR`	`"`
`encoding`	Encoding used by the CSV file. Options are `utf-8`, `utf-16`, `latin-1`. Not available in the `COPY` statement (which always uses `utf-8`).	`VARCHAR`	`utf-8`
`filename`	Add path of the containing file to each row, as a string column named `filename`. Relative or absolute paths are returned depending on the path or glob pattern provided to `read_csv`, not just filenames. Since DuckDB 1.3.0, the `filename` column is added automatically as a virtual column and this option is only kept for compatibility reasons.	`BOOL`	`false`
`files_to_sniff`	Number of files used by the CSV sniffer to detect the schema when reading multiple files. Set to `-1` to sniff all files.	`BIGINT`	`10`
`force_not_null`	A list of column names whose values are not matched against the `NULL` string (see `nullstr` or `null`). In the default case where the `NULL` string is empty, empty values in these columns are read as zero-length strings instead of `NULL`s. If a value in one of these columns matches the `NULL` string but the column is not of type `VARCHAR`, an error is raised.	`VARCHAR[]`	`[]`
`header`	First line of each file contains the column names.	`BOOL`	`false`
`hive_partitioning`	Interpret the path as a Hive partitioned path.	`BOOL`	(auto-detected)
`ignore_errors`	Ignore any parsing errors encountered.	`BOOL`	`false`
`max_line_size` or `maximum_line_size`. Not available in the `COPY` statement.	Maximum line size, in bytes.	`BIGINT`	2000000
`names` or `column_names`	Column names, as a list. See example.	`VARCHAR[]`	(empty)
`new_line`	New line character(s). Options are `'\r'`,`'\n'`, or `'\r\n'`. The CSV parser only distinguishes between single-character and double-character line delimiters. Therefore, it does not differentiate between `'\r'` and `'\n'`.	`VARCHAR`	(empty)
`normalize_names`	Normalize column names. This removes any non-alphanumeric characters from them. Column names that are reserved SQL keywords are prefixed with an underscore character (_).	`BOOL`	`false`
`null_padding`	Pad the remaining columns on the right with `NULL` values when a line lacks columns.	`BOOL`	`false`
`nullstr` or `null`	Strings that represent a `NULL` value. Accepts a single string or a list of strings. Matching against these strings can be suppressed for specific columns with `force_not_null`.	`VARCHAR` or `VARCHAR[]`	(empty)
`parallel`	Use the parallel CSV reader.	`BOOL`	`true`
`quote`	String used to quote values.	`VARCHAR`	`"`

Name	Description	Type	Default
`rejects_scan`	Name of the temporary table where information on faulty scans is stored.	`VARCHAR`	`reject_scans`
`rejects_table`	Name of the temporary table where information on faulty lines is stored.	`VARCHAR`	`reject_errors`
`rejects_limit`	Upper limit on the number of faulty lines per file that are recorded in the rejects table. Setting this to `0` means that no limit is applied.	`BIGINT`	`0`
`rejects_line_size_limit`	Maximum number of characters of a rejected line that are stored in the `csv_line` column of the reject errors table. Longer lines are truncated. Setting this to `0` disables truncation and stores the full line. Added in DuckDB 2.0.	`BIGINT`	`10000`
`sample_size`	Number of sample lines for auto detection of parameters.	`BIGINT`	20480
`sep`	Delimiter character used to separate columns within each line, e.g., `,` `;` `\t`. The delimiter character can be up to 4 bytes, e.g., 🪿. Alias for `delim`.	`VARCHAR`	`,`
`skip`	Number of lines to skip at the start of each file.	`BIGINT`	0
`store_rejects`	Skip any lines with errors and store them in the rejects table.	`BOOL`	`false`
`strict_mode`	Enforces the strictness level of the CSV Reader. When set to `true`, the parser will throw an error upon encountering any issues. When set to `false`, the parser will attempt to read structurally incorrect files. It is important to note that reading structurally incorrect files can cause ambiguity; therefore, this option should be used with caution.	`BOOL`	`true`
`thousands`	Character used to identify thousands separators in numeric values. It must be a single character and different from the `decimal_separator` option.	`VARCHAR`	(empty)
`timestampformat`	Timestamp format used when parsing and writing timestamps.	`VARCHAR`	(empty)
`timestamp_format`	Alias for `timestampformat`; only available in the COPY statement.	`VARCHAR`	(empty)
`types` or `dtypes` or `column_types`	Column types, as either a list (by position) or a struct (by name). See example.	`VARCHAR[]` or `STRUCT`	(empty)
`union_by_name`	Align columns from different files by column name instead of position. Using this option increases memory consumption.	`BOOL`	`false`

> **Tip.** DuckDB's CSV reader supports `UTF-8` (default), `UTF-16` and `Latin-1` encodings. For other encodings, you can either use the `encodings` extension or convert them e.g. using the `iconv` command-line tool:
>
> ```
> iconv -f ISO-8859-2 -t UTF-8 input.csv > input-utf-8.csv
> ```

`auto_type_candidates` Details

The `auto_type_candidates` option lets you specify the data types that should be considered by the CSV reader for column data type detection. Usage example:

```
SELECT * FROM read_csv('csv_file.csv', auto_type_candidates = ['BIGINT', 'DATE']);
```

The default value for the `auto_type_candidates` option is `['NULL', 'BOOLEAN', 'BIGINT', 'HUGEINT', 'BIGNUM', 'DOUBLE', 'TIME', 'DATE', 'TIMESTAMP', 'TIMESTAMPTZ', 'VARCHAR']`.

CSV Functions

The `read_csv` automatically attempts to figure out the correct configuration of the CSV reader using the CSV sniffer. It also automatically deduces types of columns. If the CSV file has a header, it will use the names found in that header to name the columns. Otherwise, the columns will be named `column0, column1, column2, ...`. An example with the `flights.csv` file:

```
SELECT * FROM read_csv('flights.csv');
```

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

The path can either be a relative path (relative to the current working directory) or an absolute path.

We can use `read_csv` to create a persistent table as well:

```
CREATE TABLE ontime AS
    SELECT * FROM read_csv('flights.csv');
DESCRIBE ontime;
```

column_name	column_type	null	key	default	extra
FlightDate	DATE	YES	NULL	NULL	NULL
UniqueCarrier	VARCHAR	YES	NULL	NULL	NULL
OriginCityName	VARCHAR	YES	NULL	NULL	NULL
DestCityName	VARCHAR	YES	NULL	NULL	NULL

```
SELECT * FROM read_csv('flights.csv', sample_size = 20_000);
```

If we set `delim` / `sep`, `quote`, `escape`, or `header` explicitly, we can bypass the automatic detection of this particular parameter:

```
SELECT * FROM read_csv('flights.csv', header = true);
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Writing Using the COPY Statement

The COPY statement can be used to load data from a CSV file into a table. This statement has the same syntax as the one used in PostgreSQL. To load the data using the COPY statement, we must first create a table with the correct schema (which matches the order of the columns in the CSV file and uses types that fit the values in the CSV file). COPY detects the CSV's configuration options automatically.

```
CREATE TABLE ontime (
    flightdate DATE,
    uniquecarrier VARCHAR,
    origincityname VARCHAR,
    destcityname VARCHAR
);
COPY ontime FROM 'flights.csv';
SELECT * FROM ontime;
```

flightdate	uniquecarrier	origincityname	destcityname
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

If we want to manually specify the CSV format, we can do so using the configuration options of COPY.

```
CREATE TABLE ontime (flightdate DATE, uniquecarrier VARCHAR, origincityname VARCHAR, destcityname VARCHAR);
COPY ontime FROM 'flights.csv' (DELIMITER '|', HEADER);
SELECT * FROM ontime;
```

Reading Faulty CSV Files

DuckDB supports reading erroneous CSV files. For details, see the Reading Faulty CSV Files page.

Order Preservation

The CSV reader respects the `preserve_insertion_order` configuration option to preserve insertion order. When `true` (the default), the order of the rows in the result set returned by the CSV reader is the same as the order of the corresponding lines read from the file(s). When `false`, there is no guarantee that the order is preserved.

Writing CSV Files

DuckDB can write CSV files using the `COPY ... TO` statement.

CSV Auto Detection

When using `read_csv`, the system tries to automatically infer how to read the CSV file using the CSV sniffer. This step is necessary because CSV files are not self-describing and come in many different dialects. The auto-detection works roughly as follows:

- Detect the dialect of the CSV file (delimiter, quoting rule, escape).
- Detect the types of each of the columns.
- Detect whether or not the file has a header row.

By default the system will try to auto-detect all options. However, options can be individually overridden by the user. This can be useful in case the system makes a mistake. For example, if the delimiter is chosen incorrectly, we can override it by calling the `read_csv` with an explicit delimiter (e.g., `read_csv('file.csv', delim = '|')`).

Sample Size

The type detection works by operating on a sample of the file. The size of the sample can be modified by setting the `sample_size` parameter. The default sample size is 20,480 rows. Setting the `sample_size` parameter to `-1` means the entire file is read for sampling:

```
SELECT * FROM read_csv('my_csv_file.csv', sample_size = -1);
```

The way sampling is performed depends on the type of file. If we are reading from a regular file on disk, we will jump into the file and try to sample from different locations in the file. If we are reading from a file in which we cannot jump – such as a `.gz` compressed CSV file or `stdin` – samples are taken only from the beginning of the file.

`sniff_csv` Function

It is possible to run the CSV sniffer as a separate step using the `sniff_csv(filename)` function, which returns the detected CSV properties as a table with a single row.

Besides the commonly used `sample_size` parameter, which configures the number of rows sampled, `sniff_csv` accepts the same named parameters as the `read_csv` function. Any parameter you pass is treated as fixed: the sniffer uses the value you provide and only detects the properties you did not specify. For example, passing `ignore_errors = true` lets the sniffer skip malformed lines while it detects the dialect. The `auto_detect` parameter must be `true` (its default) for `sniff_csv`; passing `auto_detect = false` results in an error. In addition, `sniff_csv` accepts a `force_match` (BOOLEAN) parameter that is not available in `read_csv`.

```
FROM sniff_csv('my_file.csv');
FROM sniff_csv('my_file.csv', sample_size = 1000);
```

The `read_csv` parameter column lists the corresponding `read_csv` parameter, so the sniffed values can be passed directly to `read_csv`.

Column name	Description	`read_csv` parameter	Example
`Delimiter`	Delimiter	`delim`	`,`
`Quote`	Quote character	`quote`	`"`
`Escape`	Escape	`escape`	`\`
`NewLineDelimiter`	New-line delimiter	`new_line`	`\r\n`
`Comment`	Comment character	`comment`	`#`
`SkipRows`	Number of rows skipped	`skip`	`1`
`HasHeader`	Whether the CSV has a header	`header`	`true`
`Columns`	Column types encoded as a `LIST` of `STRUCT`s	`columns`	`({'name': 'VARCHAR', 'age': 'BIGINT'})`
`DateFormat`	Date format	`dateformat`	`%d/%m/%Y`
`TimestampFormat`	Timestamp Format	`timestampformat`	`%Y-%m-%dT%H:%M:%S.%f`
`UserArguments`	Arguments used to invoke `sniff_csv`	(none)	`sample_size = 1000`
`Prompt`	Prompt ready to be used to read the CSV	(none)	`FROM read_csv('my_file.csv', auto_detect=false, delim=',', ...)`

Prompt

The `Prompt` column contains a SQL command with the configurations detected by the sniffer.

```
-- use line mode in CLI to get the full command
.mode line
SELECT Prompt FROM sniff_csv('my_file.csv');
```

```
Prompt = FROM read_csv('my_file.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n',
skip=0, header=true, columns={...});
```

Detection Steps

Dialect Detection

Dialect detection works by attempting to parse the samples using the set of considered values. The detected dialect is the dialect that has (1) a consistent number of columns for each row, and (2) the highest number of columns for each row.

The following dialects are considered for automatic dialect detection.

Parameters	Considered values
`delim`	`,` `|` `;` `\t`
`quote`	`"` `'` (empty)
`escape`	`"` `'` `\` (empty)

Consider the example file `flights.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-01|AA|New York, NY|Los Angeles, CA
1988-01-02|AA|New York, NY|Los Angeles, CA
1988-01-03|AA|New York, NY|Los Angeles, CA
```

In this file, the dialect detection works as follows:

- If we split by a `|` every row is split into 4 columns.
- If we split by a `,` rows 2-4 are split into 3 columns, while the first row is split into 1 column.
- If we split by `;`, every row is split into 1 column.
- If we split by `\t`, every row is split into 1 column.

In this example – the system selects the `|` as the delimiter. All rows are split into the same amount of columns, and there is more than one column per row meaning the delimiter was actually found in the CSV file.

Type Detection

After detecting the dialect, the system will attempt to figure out the types of each of the columns. Note that this step is only performed if we are calling `read_csv`. In case of the COPY statement the types of the table that we are copying into will be used instead.

The type detection works by attempting to convert the values in each column to the candidate types. If the conversion is unsuccessful, the candidate type is removed from the set of candidate types for that column. After all samples have been handled – the remaining candidate type with the highest priority is chosen. The default set of candidate types is given below, in order of priority:

Types
NULL
BOOLEAN
TIME
DATE
TIMESTAMP
TIMESTAMPTZ
BIGINT
HUGEINT
BIGNUM
DOUBLE
VARCHAR

Everything can be cast to VARCHAR, therefore, this type has the lowest priority meaning that all columns are converted to VARCHAR as a fallback if they cannot be cast to anything else. In `flights.csv` the `FlightDate` column will be cast to a DATE, while the other columns will be cast to VARCHAR.

The set of candidate types that should be considered by the CSV reader can be specified explicitly using the `auto_type_candidates` option. VARCHAR as the fallback type will always be considered as a candidate type whether you specify it or not.

Here are all additional candidate types that may be specified using the `auto_type_candidates` option, in order of priority:

Types
TINYINT
SMALLINT
INTEGER
DECIMAL
FLOAT

Even though the set of data types that can be automatically detected may appear quite limited, the CSV reader can be configured to read arbitrarily complex types by using the `types`-option described in the next section.

Type detection can be entirely disabled by using the `all_varchar` option. If this is set all columns will remain as VARCHAR (as they originally occur in the CSV file).

Note that using quote characters vs. no quote characters (e.g., `"42"` and `42`) does not make a difference for type detection. Quoted fields will not be converted to VARCHAR, instead, the sniffer will try to find the type candidate with the highest priority.

Overriding Type Detection The detected types can be individually overridden using the `types` option. This option takes either of two options:

- A list of type definitions (e.g., `types = ['INTEGER', 'VARCHAR', 'DATE']`). This overrides the types of the columns in-order of occurrence in the CSV file.
- Alternatively, `types` takes a `name → type` map which overrides options of individual columns (e.g., `types = {'quarter': 'INTEGER'}`).

The set of column types that may be specified using the `types` option is not as limited as the types available for the `auto_type_candidates` option: any valid type definition is acceptable to the `types`-option. (To get a valid type definition, use the `typeof()` function, or use the `column_type` column of the `DESCRIBE` result.)

The `sniff_csv()` function's `Column` field returns a struct with column names and types that can be used as a basis for overriding types.

Header Detection

Header detection works by checking if the candidate header row deviates from the other rows in the file in terms of types. For example, in `flights.csv`, we can see that the header row consists of only VARCHAR columns – whereas the values contain a DATE value for the `FlightDate` column. As such – the system defines the first row as the header row and extracts the column names from the header row.

In files that do not have a header row, the column names are generated as `column0`, `column1`, etc.

Note that headers cannot be detected correctly if all columns are of type VARCHAR – as in this case the system cannot distinguish the header row from the other rows in the file. In this case, the system assumes the file has a header. This can be overridden by setting the `header` option to `false`.

Dates and Timestamps

DuckDB supports the ISO 8601 format by default for timestamps, dates and times. Unfortunately, not all dates and times are formatted using this standard. For that reason, the CSV reader also supports the `dateformat` and `timestampformat` options. Using this format the user can specify a format string that specifies how the date or timestamp should be read.

As part of the auto-detection, the system tries to figure out if dates and times are stored in a different representation. This is not always possible – as there are ambiguities in the representation. For example, the date `01-02-2000` can be parsed as either January 2nd or February 1st. Often these ambiguities can be resolved. For example, if we later encounter the date `21-02-2000` then we know that the format must have been `DD-MM-YYYY`. `MM-DD-YYYY` is no longer possible as there is no 21st month.

If the ambiguities cannot be resolved by looking at the data the system has a list of preferences for which date format to use. If the system chooses incorrectly, the user can specify the `dateformat` and `timestampformat` options manually.

The system considers the following formats for dates (`dateformat`). Higher entries are chosen over lower entries in case of ambiguities (i.e., ISO 8601 is preferred over `MM-DD-YYYY`).

dateformat
ISO 8601
%y-%m-%d
%Y-%m-%d
%d-%m-%y
%d-%m-%Y
%m-%d-%y
%m-%d-%Y

The system considers the following formats for timestamps (`timestampformat`). Higher entries are chosen over lower entries in case of ambiguities.

timestampformat
ISO 8601
%y-%m-%d %H:%M:%S
%Y-%m-%d %H:%M:%S
%d-%m-%y %H:%M:%S
%d-%m-%Y %H:%M:%S
%m-%d-%y %I:%M:%S %p
%m-%d-%Y %I:%M:%S %p
%Y-%m-%d %H:%M:%S.%f

Reading Faulty CSV Files

CSV files can come in all shapes and forms, with some presenting many errors that make the process of cleanly reading them inherently difficult. To help users read these files, DuckDB supports detailed error messages, the ability to skip faulty lines and the possibility of storing faulty lines in a temporary table to assist users with a data cleaning step.

Structural Errors

DuckDB supports the detection and skipping of several different structural errors. In this section, we will go over each error with an example. For the examples, consider the following table:

```
CREATE TABLE people (name VARCHAR, birth_date DATE);
```

DuckDB detects the following error types:

- `CAST`: Casting errors occur when a column in the CSV file cannot be cast to the expected schema value. For example, the line `Pedro,The 90s` would cause an error since the string `The 90s` cannot be cast to a date.
- `MISSING COLUMNS`: This error occurs if a line in the CSV file has fewer columns than expected. In our example, we expect two columns; therefore, a row with just one value, e.g., `Pedro`, would cause this error.
- `TOO MANY COLUMNS`: This error occurs if a line in the CSV has more columns than expected. In our example, any line with more than two columns would cause this error, e.g., `Pedro,01-01-1992,pdet`.
- `UNQUOTED VALUE`: Quoted values in CSV lines must always be unquoted at the end; if a quoted value remains quoted throughout, it will cause an error. For example, assuming our scanner uses `quote='"'`, the line `"pedro"holanda, 01-01-1992` would present an unquoted value error.
- `LINE SIZE OVER MAXIMUM`: DuckDB has a parameter that sets the maximum line size a CSV file can have, which by default is set to 2,000,000 bytes. Assuming our scanner is set to `max_line_size = 25`, the line `Pedro Holanda, 01-01-1992` would produce an error, as it exceeds 25 bytes.
- `INVALID ENCODING`: DuckDB supports UTF-8 strings, UTF-16 and Latin-1 encodings. Lines containing other characters will produce an error. For example, the line `pedro\xff\xff, 01-01-1992` would be problematic.

Anatomy of a CSV Error

By default, when performing a CSV read, if any structural errors are encountered, the scanner will immediately stop the scanning process and throw the error to the user. These errors are designed to provide as much information as possible to allow users to evaluate them directly in their CSV file.

This is an example for a full error message:

```
Conversion Error:
CSV Error on Line: 5648
Original Line: Pedro,The 90s
Error when converting column "birth_date". date field value out of range: "The 90s", expected format is
(DD-MM-YYYY)

Column date is being converted as type DATE
This type was auto-detected from the CSV file.
Possible solutions:
* Override the type for this column manually by setting the type explicitly, e.g., types={'birth_date':
'VARCHAR'}
* Set the sample size to a larger value to enable the auto-detection to scan more values, e.g., sample_
size=-1
* Use a COPY statement to automatically derive types from an existing table.

  file= people.csv
  delimiter = , (Auto-Detected)
  quote = " (Auto-Detected)
  escape = " (Auto-Detected)
  new_line = \r\n (Auto-Detected)
  header = true (Auto-Detected)
  skip_rows = 0 (Auto-Detected)
  date_format = (DD-MM-YYYY) (Auto-Detected)
  timestamp_format =  (Auto-Detected)
  null_padding=0
  sample_size=20480
  ignore_errors=false
  all_varchar=0
```

The first block provides us with information regarding where the error occurred, including the line number, the original CSV line and which field was problematic:

```
Conversion Error:
CSV Error on Line: 5648
Original Line: Pedro,The 90s
```

```
Error when converting column "birth_date". date field value out of range: "The 90s", expected format is
(DD-MM-YYYY)
```

The second block provides us with potential solutions:

```
Column date is being converted as type DATE
This type was auto-detected from the CSV file.
Possible solutions:
* Override the type for this column manually by setting the type explicitly, e.g., types={'birth_date':
'VARCHAR'}
* Set the sample size to a larger value to enable the auto-detection to scan more values, e.g., sample_
size=-1
* Use a COPY statement to automatically derive types from an existing table.
```

Since the type of this field was auto-detected, it suggests defining the field as a VARCHAR or fully utilizing the dataset for type detection.

Finally, the last block presents some of the options used in the scanner that can cause errors, indicating whether they were auto-detected or manually set by the user.

Using the `ignore_errors` Option

There are cases where CSV files may have multiple structural errors, and users simply wish to skip these and read the correct data. Reading erroneous CSV files is possible by utilizing the `ignore_errors` option. With this option set, rows containing data that would otherwise cause the CSV parser to generate an error will be ignored. In our example, we will demonstrate a CAST error, but note that any of the errors described in our Structural Error section would cause the faulty line to be skipped.

For example, consider the following CSV file, `faulty.csv`:

```
Pedro,31
Oogie Boogie, three
```

If you read the CSV file, specifying that the first column is a VARCHAR and the second column is an `INTEGER`, loading the file would fail, as the string `three` cannot be converted to an `INTEGER`.

For example, the following query will throw a casting error.

```
FROM read_csv('faulty.csv', columns = {'name': 'VARCHAR', 'age': 'INTEGER'});
```

However, with `ignore_errors` set, the second row of the file is skipped, outputting only the complete first row. For example:

```
FROM read_csv(
    'faulty.csv',
    columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
    ignore_errors = true
);
```

Outputs:

name	age
Pedro	31

One should note that the CSV Parser is affected by the projection pushdown optimization. Hence, if we were to select only the name column, both rows would be considered valid, as the casting error on the age would never occur. For example:

```
SELECT name
FROM read_csv('faulty.csv', columns = {'name': 'VARCHAR', 'age': 'INTEGER'});
```

Outputs:

name
Pedro
Oogie Boogie

Retrieving Faulty CSV Lines

Being able to read faulty CSV files is important, but for many data cleaning operations, it is also necessary to know exactly which lines are corrupted and what errors the parser discovered on them. For scenarios like these, it is possible to use DuckDB's CSV Rejects Table feature. By default, this feature creates two temporary tables.

1. `reject_scans`: Stores information regarding the parameters of the CSV Scanner.
2. `reject_errors`: Stores information regarding each CSV faulty line and in which CSV Scanner they happened.

Note that any of the errors described in our Structural Error section will be stored in the rejects tables. Also, if a line has multiple errors, multiple entries will be stored for the same line, one for each error.

Reject Scans

The CSV Reject Scans Table returns the following information:

Column name	Description	Type
`scan_id`	The internal ID used in DuckDB to represent that scanner	`UBIGINT`
`file_id`	A scanner might happen over multiple files, so the file_id represents a unique file in a scanner	`UBIGINT`
`file_path`	The file path	`VARCHAR`
`delimiter`	The delimiter used e.g., ;	`VARCHAR`
`quote`	The quote used e.g., ”	`VARCHAR`
`escape`	The escape used e.g., ”	`VARCHAR`
`newline_delimiter`	The newline delimiter used e.g., \r\n	`VARCHAR`
`skip_rows`	If any rows were skipped from the top of the file	`UINTEGER`
`has_header`	If the file has a header	`BOOLEAN`
`columns`	The schema of the file (i.e., all column names and types)	`VARCHAR`
`date_format`	The format used for date types	`VARCHAR`
`timestamp_format`	The format used for timestamp types	`VARCHAR`
`user_arguments`	Any extra scanner parameters manually set by the user	`VARCHAR`

Reject Errors

The CSV Reject Errors Table returns the following information:

Column name	Description	Type
`scan_id`	The internal ID used in DuckDB to represent that scanner, used to join with reject scans tables	`UBIGINT`

Column name	Description	Type
`file_id`	The file_id represents a unique file in a scanner, used to join with reject scans tables	`UBIGINT`
`line`	Line number, from the CSV File, where the error occurred.	`UBIGINT`
`line_byte_position`	Byte Position of the start of the line, where the error occurred.	`UBIGINT`
`byte_position`	Byte Position where the error occurred.	`UBIGINT`
`column_idx`	If the error happens in a specific column, the index of the column.	`UBIGINT`
`column_name`	If the error happens in a specific column, the name of the column.	`VARCHAR`
`error_type`	The type of the error that happened.	`ENUM`
`csv_line`	The original CSV line, truncated to `rejects_line_size_limit` characters.	`VARCHAR`
`error_message`	The error message produced by DuckDB.	`VARCHAR`

Parameters

The parameters listed below are used in the `read_csv` function to configure the CSV Rejects Table.

Name	Description	Type	Default
`store_rejects`	If set to true, any errors in the file will be skipped and stored in the default rejects temporary tables.	`BOOLEAN`	False
`rejects_scan`	Name of a temporary table where the information of the scan information of faulty CSV file are stored.	`VARCHAR`	reject_scans
`rejects_table`	Name of a temporary table where the information of the faulty lines of a CSV file are stored.	`VARCHAR`	reject_errors
`rejects_limit`	Upper limit on the number of faulty records from a CSV file that will be recorded in the rejects table. 0 is used when no limit should be applied.	`BIGINT`	0
`rejects_line_size_limit`	Maximum number of characters of a rejected line stored in the `csv_line` column. Longer lines are truncated, and `0` disables truncation, storing the full line.	`BIGINT`	10000

> New As of DuckDB 2.0, the `rejects_line_size_limit` parameter bounds how many characters of each rejected line are stored in the `csv_line` column of the reject errors table.

To store the information of the faulty CSV lines in a rejects table, the user must simply set the `store_rejects` option to true. For example:

```
FROM read_csv(
    'faulty.csv',
    columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
    store_rejects = true
);
```

You can then query both the `reject_scans` and `reject_errors` tables, to retrieve information about the rejected tuples. For example:

```
FROM reject_scans;
```

Outputs:

scan_id	file_id	file_path	delimiter	quote	escape	newline_delimiter	skip_rows	has_header	columns	date_format	timestamp_format	user_arguments
5	0	faulty.csv	,	”	”	\n	0	false	{'name': 'VARCHAR','age': 'INTEGER'}			store_rejects=true

```
FROM reject_errors;
```

Outputs:

scan_id	file_id	line	line_byte_position	byte_position	column_idx	column_name	error_type	csv_line	error_message
5	0	2	10	23	2	age	CAST	Oogie Boogie, three	Error when converting column ”age”. Could not convert string ” three” to 'INTEGER'

CSV Import Tips

Below is a collection of tips to help when attempting to import complex CSV files. In the examples, we use the `flights.csv` file.

Override the Header Flag if the Header Is Not Correctly Detected

If a file contains only string columns the `header` auto-detection might fail. Provide the `header` option to override this behavior.

```
SELECT * FROM read_csv('flights.csv', header = true);
```

Provide Names if the File Does Not Contain a Header

If the file does not contain a header, names will be auto-generated by default. You can provide your own names with the `names` option.

```
SELECT * FROM read_csv('flights.csv', names = ['DateOfFlight', 'CarrierName']);
```

Override the Types of Specific Columns

The `types` flag can be used to override types of only certain columns by providing a struct of `name` → `type` mappings.

```
SELECT * FROM read_csv('flights.csv', types = {'FlightDate': 'DATE'});
```

Use COPY When Loading Data into a Table

The COPY statement copies data directly into a table. The CSV reader uses the schema of the table instead of auto-detecting types from the file. This speeds up the auto-detection, and prevents mistakes from being made during auto-detection.

```
COPY tbl FROM 'test.csv';
```

Use `union_by_name` When Loading Files with Different Schemas

The `union_by_name` option can be used to unify the schema of files that have different or missing columns. For files that do not have certain columns, `NULL` values are filled in.

```
SELECT * FROM read_csv('flights*.csv', union_by_name = true);
```

To load data into *an existing table* where the table has more columns than the CSV file, you can use the `INSERT INTO ... BY NAME` clause:

```
INSERT INTO tbl BY NAME
    SELECT * FROM read_csv('input.csv');
```

Sample Size

If the CSV sniffer is not detecting the correct type, try increasing the sample size. The option `sample_size = -1` forces the sniffer to read the entire file:

```
SELECT * FROM read_csv('my_csv_file.csv', sample_size = -1);
```

I/O

> **Warning.** DuckDB ships with safe defaults. The configuration options described on this page are advanced options, so proceed with caution when changing them.

Asynchronous I/O

> New Asynchronous I/O was introduced in DuckDB 2.0.

Starting with v2.0, DuckDB supports asynchronous I/O. Instead of blocking a worker thread until requested data arrives, DuckDB issues reads in the background and keeps multiple requests in flight while worker threads process data that has already arrived. This can significantly speed up queries when synchronous I/O does not saturate the available bandwidth, e.g., when reading data from object storage such as S3.

For a detailed explanation of the design and benchmark results, see the "Asynchronous I/O in DuckDB" blog post.

Supported Formats

Asynchronous I/O is currently supported for the following formats:

- Parquet files
- uncompressed, seekable CSV files encoded in UTF-8
- DuckDB's native storage format

Other formats, such as JSON files, use synchronous I/O. Asynchronous I/O is used automatically for the supported formats, including when they are read through data lake formats such as DuckLake.

Thread Pools

DuckDB uses two thread pools:

- The regular pool contains the worker threads, which perform the actual query processing (e.g., decoding, joins and aggregations). Its size is controlled by the `threads` setting and defaults to the number of CPU cores. Regular threads prioritize query processing but can also perform I/O tasks when idle.
- The asynchronous pool contains threads dedicated to blocking I/O. As these threads spend most of their time waiting for responses (e.g., HTTP requests), there are more of them than CPU cores: by default, four times the number of system threads, capped at 256. Its size is controlled by the `async_threads` setting.

For example, to set the number of asynchronous I/O threads to 48, run:

```
SET async_threads = 48;
```

> **Warning.** The asynchronous pool increases the total number of threads per DuckDB instance. If you run many DuckDB instances in a single process, consider setting `threads` and `async_threads` explicitly.

Read-Ahead

To keep the asynchronous threads busy, DuckDB reads ahead: it schedules the reads for upcoming scan jobs (e.g., row groups in Parquet files or byte ranges in CSV files) before the worker threads need them. While a worker thread processes the current job, the asynchronous threads fetch the data for the next jobs. If the data for a job has not arrived yet, the worker thread is free to run other tasks in the meantime.

Keeping more fetch tasks in flight consumes more memory. To determine a budget and avoid out-of-memory issues, DuckDB provides the `read_ahead_depth` configuration option. It can have three types of values:

- `-1` (default): unlimited depth, bounded by memory.
- `N > 0`: at most N jobs ahead, with no memory budget.
- `0`: read-ahead is off, each scan task schedules I/O only for its own job.

To configure it, use the SET clause, e.g.:

```
SET read_ahead_depth = 5;
```

In the default mode, the read-ahead budget is negotiated with the same memory manager that distributes memory between operators such as joins, sorts and window functions. Under high memory pressure, the read-ahead queue shrinks to a single job and the scan behaves similarly to a synchronous scan. Once memory frees up, the queue fills up again. To limit the total memory used by DuckDB, use the `memory_limit` setting.

Tuning

The default settings work well in most cases. On machines with high network bandwidth, you can further increase throughput by fixing the read-ahead depth and adjusting the number of asynchronous threads and the HTTP retry settings. For example, the following configuration saturated a 25 Gbit/s network on a 64-core EC2 instance reading Parquet files from S3:

```
SET read_ahead_depth = 64;
SET async_threads = 48;
SET http_retries = 8;
SET http_retry_wait_ms = 50;
SET http_retry_backoff = 2;
```

As the row group is the unit of parallelism for Parquet scans, asynchronous I/O works best if Parquet files have at least as many row groups as the number of threads. Files with only a few very large row groups cannot keep enough requests in flight to saturate the network.

> New As of DuckDB 2.0, the size of the HTTP client pool can be tuned with the `http_client_pool_capacity` setting. By default the capacity is derived automatically from the number of CPU threads and asynchronous threads, bounded by the operating system file-descriptor limit, and capped at 256. Resizing `async_threads` also resizes the automatic pool so that it tracks the asynchronous pool. To pin the pool to a fixed size, set a positive integer; use `RESET http_client_pool_capacity` to return to the automatic value.

```
SET http_client_pool_capacity = 8;
```

Synchronizing Writes to Disk

When DuckDB persists changes to a database on the local file system, it asks the operating system to flush the written data to stable storage. The `fsync_mode` configuration option controls how this is done. It can have three values:

- STANDARD (default): uses the regular sync call of the platform, i.e., `fdatasync` or `fsync` on Unix-like systems and `FlushFileBuffers` on Windows.
- FULL: on macOS, uses `F_FULLFSYNC`, which also instructs the drive to flush its write cache and thus guarantees durability in case of a power failure. If the file system does not support `F_FULLFSYNC`, DuckDB falls back to the STANDARD behavior. On other platforms, FULL is equivalent to STANDARD.
- NONE: skips the sync call. Writes are handed over to the operating system, which flushes them to disk at its own pace.

For example, to disable syncing for a performance-critical workload, run:

```
SET fsync_mode = 'NONE';
```

The option is global, i.e., it applies to the whole DuckDB instance.

> **Warning.** Setting `fsync_mode` to `NONE` may cause data loss or database corruption if the operating system crashes or the machine loses power. Only use it if the database can be recreated from other sources.

JSON Files

JSON Overview

DuckDB supports SQL functions that are useful for reading values from existing JSON and creating new JSON data. JSON is supported with the `json` extension which is shipped with most DuckDB distributions and is auto-loaded on first use. If you would like to install or load it manually, please consult the "Installing and Loading" page.

About JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). While it is not a very efficient format for tabular data, it is very commonly used, especially as a data interchange format.

JSONPath and JSON Pointer Syntax

DuckDB implements multiple interfaces for JSON extraction: JSONPath and JSON Pointer. Both of them work with the arrow operator (`->`) and the `json_extract` function call.

Note that DuckDB only supports lookups in JSONPath, i.e., extracting fields with `.<key>` or array elements with `[<index>]`. Arrays can be indexed from the back and both approaches support the wildcard `*`. DuckDB does *not* support the full JSONPath syntax because SQL is readily available for any further transformations.

> It's best to pick either the JSONPath or the JSON Pointer syntax and use it in your entire application.

Indexing

> **Warning.** Following PostgreSQL's conventions, DuckDB uses 1-based indexing for its `ARRAY` and `LIST` data types but 0-based indexing for the JSON data type.

Examples

Loading JSON

Read a JSON file from disk, auto-infer options:

```
SELECT * FROM 'todos.json';
```

Use the `read_json` function with custom options:

```
SELECT *
FROM read_json('todos.json',
               format = 'array',
               columns = {userId: 'UBIGINT',
                          id: 'UBIGINT',
                          title: 'VARCHAR',
                          completed: 'BOOLEAN'});
```

Read a JSON file from stdin, auto-infer options:

```
cat data/json/todos.json | duckdb -c "SELECT * FROM read_json('/dev/stdin')"
```

Read a JSON file into a table:

```
CREATE TABLE todos (userId UBIGINT, id UBIGINT, title VARCHAR, completed BOOLEAN);
COPY todos FROM 'todos.json' (AUTO_DETECT true);
```

Alternatively, create a table without specifying the schema manually with a `CREATE TABLE ... AS SELECT` clause:

```
CREATE TABLE todos AS
    SELECT * FROM 'todos.json';
```

Since DuckDB 1.3.0, the JSON reader returns the `filename` virtual column:

```
SELECT filename, *
FROM 'todos-*.json';
```

Writing JSON

Write the result of a query to a JSON file:

```
COPY (SELECT * FROM todos) TO 'todos.json';
```

JSON Data Type

Create a table with a column for storing JSON data and insert data into it:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');
```

Retrieving JSON Data

Retrieve the family key's value:

```
SELECT j.family FROM example;
```

```
"anatidae"
```

Extract the family key's value with a JSONPath expression as JSON:

```
SELECT j->'$.family' FROM example;
```

```
"anatidae"
```

Extract the family key's value with a JSONPath expression as a VARCHAR:

```
SELECT j->>'$.family' FROM example;
```

```
anatidae
```

Using Quotes for Special Characters

JSON object keys that contain the special `[` and `.` characters can be used by surrounding them with double quotes (`"`):

```
SELECT '{"d[u]._\"ck":42}'->'$."d[u]._\"ck"' AS v;
```

```
42
```

Creating JSON

JSON Creation Functions

The following functions are used to create JSON.

Function	Description
`to_json(any)`	Create JSON from a value of any type. Our `LIST` is converted to a JSON array, and our `STRUCT` and `MAP` are converted to a JSON object.
`json_quote(any)`	Alias for `to_json`.
`array_to_json(list)`	Alias for `to_json` that only accepts `LIST`.
`row_to_json(list)`	Alias for `to_json` that only accepts `STRUCT`.
`json_array(any, ...)`	Create a JSON array from the values in the argument lists.
`json_object(key, value, ...)`	Create a JSON object from `key`, `value` pairs in the argument list. Requires an even number of arguments.
`json_merge_patch(json, json)`	Merge two JSON documents together.

Examples:

```
SELECT to_json('duck');
```

```
"duck"
```

```
SELECT to_json([1, 2, 3]);
```

```
[1,2,3]
```

```
SELECT to_json({duck : 42});
```

```
{"duck":42}
```

```
SELECT to_json(MAP(['duck'], [42]));
```

```
{"duck":42}
```

```
SELECT json_array('duck', 42, 'goose', 123);
```

```
["duck",42,"goose",123]
```

```
SELECT json_object('duck', 42, 'goose', 123);
```

```
{"duck":42,"goose":123}
```

```
SELECT json_merge_patch('{"duck": 42}', '{"goose": 123}');
```

```
{"duck":42,"goose":123}
```

Loading JSON

The DuckDB JSON reader can automatically infer which configuration flags to use by analyzing the JSON file. This will work correctly in most situations, and should be the first option attempted. In rare situations where the JSON reader cannot figure out the correct configuration, it is possible to manually configure the JSON reader to correctly parse the JSON file.

The `read_json` Function

The `read_json` is the simplest method of loading JSON files: it automatically attempts to figure out the correct configuration of the JSON reader. It also automatically deduces types of columns. In the following example, we use the `todos.json` file,

```
SELECT *
FROM read_json('todos.json')
LIMIT 5;
```

userId	id	title	completed
1	1	delectus aut autem	false
1	2	quis ut nam facilis et officia qui	false
1	3	fugiat veniam minus	false
1	4	et porro tempora	true
1	5	laboriosam mollitia et enim quasi adipisci quia provident illum	false

We can use `read_json` to create a persistent table as well:

```
CREATE TABLE todos AS
    SELECT *
    FROM read_json('todos.json');
DESCRIBE todos;
```

column_name	column_type	null	key	default	extra
userId	BIGINT	YES	NULL	NULL	NULL
id	BIGINT	YES	NULL	NULL	NULL
title	VARCHAR	YES	NULL	NULL	NULL
completed	BOOLEAN	YES	NULL	NULL	NULL

If we specify types for a subset of columns, `read_json` excludes columns that we don't specify:

```
SELECT *
FROM read_json(
        'todos.json',
        columns = {userId: 'UBIGINT', completed: 'BOOLEAN'}
    )
LIMIT 5;
```

Note that only the `userId` and `completed` columns are shown:

userId	completed
1	false
1	false
1	false
1	true
1	false

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Functions for Reading JSON Objects

The following table functions are used to read JSON:

Function	Description
`read_json_objects(filename)`	Read a JSON object from `filename`, where `filename` can also be a list of files or a glob pattern.
`read_ndjson_objects(filename)`	Alias for `read_json_objects` with the parameter `format` set to `newline_delimited`.
`read_json_objects_auto(filename)`	Alias for `read_json_objects` with the parameter `format` set to `auto`.

Parameters

These functions have the following parameters:

Name	Description	Type	Default
`allow_empty`	If the file pattern or file list matches no files, return an empty result (with a single `empty` column of type `BOOL`) instead of throwing an error. Added in DuckDB 2.0.	`BOOL`	`false`
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `t.json.gz` will use gzip, `t.json` will use none). Options are `none`, `gzip`, `zstd` and `auto_detect`.	`VARCHAR`	`auto_detect`
`filename`	Whether or not an extra `filename` column should be included in the result. Since DuckDB 1.3.0, the `filename` column is added automatically as a virtual column and this option is only kept for compatibility reasons.	`BOOL`	`false`
`format`	Can be one of `auto`, `unstructured`, `newline_delimited` and `array`.	`VARCHAR`	`auto`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	`BOOL`	(auto-detected)
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `newline_delimited`).	`BOOL`	`false`
`maximum_sample_files`	The maximum number of JSON files sampled for auto-detection.	`BIGINT`	`32`
`maximum_object_size`	The maximum size of a JSON object (in bytes).	`UINTEGER`	`16777216`

The `format` parameter specifies how to read the JSON from a file. With `unstructured`, the top-level JSON is read, e.g., for `birds.json`:

```
{
  "duck": 42
}
{
  "goose": [1, 2, 3]
}
```

```
FROM read_json_objects('birds.json', format = 'unstructured');
```

will result in two objects being read:

```
┌──────────────────────────────┐
│             json             │
│             json             │
├──────────────────────────────┤
│ {\n    "duck": 42\n}         │
│ {\n    "goose": [1, 2, 3]\n} │
└──────────────────────────────┘
```

With `newline_delimited`, NDJSON is read, where each JSON is separated by a newline (`\n`), e.g., for `birds-nd.json`:

```
{"duck": 42}
{"goose": [1, 2, 3]}
```

```
FROM read_json_objects('birds-nd.json', format = 'newline_delimited');
```

will also result in two objects being read:

```
┌──────────────────────┐
│         json         │
│         json         │
├──────────────────────┤
│ {"duck": 42}         │
│ {"goose": [1, 2, 3]} │
└──────────────────────┘
```

With `array`, each array element is read, e.g., for `birds-array.json`:

```
[
    {
        "duck": 42
    },
    {
        "goose": [1, 2, 3]
    }
]
```

```
FROM read_json_objects('birds-array.json', format = 'array');
```

will again result in two objects being read:

```
┌──────────────────────────────────────┐
│                 json                 │
│                 json                 │
├──────────────────────────────────────┤
│ {\n        "duck": 42\n    }         │
│ {\n        "goose": [1, 2, 3]\n    } │
└──────────────────────────────────────┘
```

Functions for Reading JSON as a Table

DuckDB also supports reading JSON as a table, using the following functions:

Function	Description
`read_json(filename)`	Read JSON from `filename`, where `filename` can also be a list of files, or a glob pattern.
`read_json_auto(filename)`	Alias for `read_json`.
`read_ndjson(filename)`	Alias for `read_json` with parameter `format` set to `newline_delimited`.

Function	Description
`read_ndjson_auto(filename)`	Alias for `read_json` with parameter `format` set to `newline_delimited`.

Parameters

Besides the `maximum_object_size`, `format`, `ignore_errors` and `compression`, these functions have additional parameters:

Name	Description	Type	Default
`array`	Shorthand for the `format` parameter: `true` is equivalent to `format = 'array'` and `false` is equivalent to `format = 'newline_delimited'`. As `array` is a keyword, the parameter name must be quoted, e.g., `read_json('my_file.json', "array" = true)`. Added in DuckDB 2.0.	`BOOL`	-
`auto_detect`	Whether to auto-detect the names of the keys and data types of the values automatically	`BOOL`	`true`
`columns`	A struct that specifies the key names and value types contained within the JSON file (e.g., `{key1: 'INTEGER', key2: 'VARCHAR'}`). If `auto_detect` is enabled these will be inferred	`STRUCT`	`(empty)`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`iso`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`records`	Can be one of `auto`, `true`, `false`	`VARCHAR`	`auto`
`sample_size`	Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file	`UBIGINT`	`20480`
`timestampformat`	Specifies the date format to use when parsing timestamps. See Date Format. When set to `iso` (the default), ISO 8601 timestamps with timezone offsets (e.g., `2024-01-01T12:00:00+05:00`) and fractional seconds (e.g., `2024-01-01T12:00:00.123Z`) are automatically inferred as `TIMESTAMPTZ`.	`VARCHAR`	`iso`
`union_by_name`	Whether the schemas of multiple JSON files should be unified	`BOOL`	`false`
`map_inference_threshold`	Controls the threshold for number of columns whose schema will be auto-detected; if JSON schema auto-detection would infer a `STRUCT` type for a field that has *more* than this threshold number of subfields, it infers a `MAP` type instead. Set to `-1` to disable `MAP` inference.	`BIGINT`	`200`
`field_appearance_threshold`	The JSON reader divides the number of appearances of each JSON field by the auto-detection sample size. If the average over the fields of an object is less than this threshold, it will default to using a `MAP` type with value type of merged field types.	`DOUBLE`	`0.1`

Name	Description	Type	Default
`geojson`	When `true`, parse GeoJSON geometry objects as `GEOMETRY` instead of `JSON`/`STRUCT`. Stray GeoJSON fragments in ordinary JSON are not inferred unless this is set.	`BOOL`	`false`

Note that DuckDB can convert JSON arrays directly to its internal `LIST` type, and missing keys become `NULL`:

```
SELECT *
FROM read_json(
    ['birds1.json', 'birds2.json'],
    columns = {duck: 'INTEGER', goose: 'INTEGER[]', swan: 'DOUBLE'}
);
```

duck	goose	swan
42	[1, 2, 3]	NULL
43	[4, 5, 6]	3.3

DuckDB can automatically detect the types like so:

```
SELECT goose, duck FROM read_json('*.json.gz');
SELECT goose, duck FROM '*.json.gz'; -- equivalent
```

DuckDB can read (and auto-detect) a variety of formats, specified with the `format` parameter. Querying a JSON file that contains an `array`, e.g.:

```
[
  {
    "duck": 42,
    "goose": 4.2
  },
  {
    "duck": 43,
    "goose": 4.3
  }
]
```

Can be queried exactly the same as a JSON file that contains `unstructured` JSON, e.g.:

```
{
    "duck": 42,
    "goose": 4.2
}
{
    "duck": 43,
    "goose": 4.3
}
```

Both can be read as the table:

```
SELECT *
FROM read_json('birds.json');
```

duck	goose
42	4.2
43	4.3

If your JSON file does not contain "records", i.e., any other type of JSON than objects, DuckDB can still read it. This is specified with the `records` parameter. The `records` parameter specifies whether the JSON contains records that should be unpacked into individual columns. DuckDB also attempts to auto-detect this. For example, take the following file, `birds-records.json`:

```
{"duck": 42, "goose": [1, 2, 3]}
{"duck": 43, "goose": [4, 5, 6]}
```

```
SELECT *
FROM read_json('birds-records.json');
```

The query results in two columns:

duck	goose
42	[1,2,3]
43	[4,5,6]

You can read the same file with `records` set to `false`, to get a single column, which is a STRUCT containing the data:

json
{'duck': 42, 'goose': [1,2,3]}
{'duck': 43, 'goose': [4,5,6]}

For additional examples reading more complex data, please see the "Shredding Deeply Nested JSON, One Vector at a Time" blog post.

Loading with the COPY Statement Using `FORMAT json`

When the `json` extension is installed, `FORMAT json` is supported for `COPY FROM`, `IMPORT DATABASE`, as well as `COPY TO` and `EXPORT DATABASE`. See the COPY statement and the IMPORT / EXPORT clauses.

By default, COPY expects newline-delimited JSON. If you prefer copying data to/from a JSON array, you can specify `ARRAY true`, e.g.,

```
COPY (SELECT * FROM range(5) r(i))
TO 'numbers.json' (ARRAY true);
```

will create the following file:

```
[
    {"i":0},
    {"i":1},
    {"i":2},
    {"i":3},
    {"i":4}
]
```

This can be read back to DuckDB as follows:

```
CREATE TABLE numbers (i BIGINT);
COPY numbers FROM 'numbers.json' (ARRAY true);
```

The format can be detected automatically like so:

```
CREATE TABLE numbers (i BIGINT);
COPY numbers FROM 'numbers.json' (AUTO_DETECT true);
```

We can also create a table from the auto-detected schema:

```
CREATE TABLE numbers AS
    FROM 'numbers.json';
```

GeoJSON

> New This feature was introduced in DuckDB 2.0.

The `json` extension can read GeoJSON when you opt in. Pass `geojson := true` to `read_json` so geometry objects become `GEOMETRY` (the `spatial` type) instead of generic JSON:

```
SELECT *
FROM read_json('features.geojson', geojson := true);
```

`.geojson` / `.geojsonl` files are also recognized as JSON replacements. Convert with `st_geomfromgeojson` / `st_asgeojson`. Writing FeatureCollections is documented on the Writing JSON page.

Writing JSON

The contents of tables or the result of queries can be written directly to a JSON file using the COPY statement. For example:

```
CREATE TABLE cities AS
    FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
COPY cities TO 'cities.json';
```

This will result in `cities.json` with the following content:

```
{"name":"Amsterdam","id":1}
{"name":"London","id":2}
```

See the COPY statement for more information.

GeoJSON

> New This feature was introduced in DuckDB 2.0.

`COPY ... (FORMAT GEOJSON)` writes a GeoJSON `FeatureCollection`. `FORMAT GEOJSONL` (or `ARRAY true` with `FORMAT GEOJSON`) writes newline-delimited `Feature` objects. You do not need `json_geometry_format` for these copy formats.

```
COPY (SELECT 'POINT(0 1)'::GEOMETRY AS geom, 10 AS my_field)
    TO 'test.json' (FORMAT GEOJSON);
COPY (SELECT 'POINT(0 1)'::GEOMETRY AS geom, 10 AS my_field)
    TO 'test.jsonl' (FORMAT GEOJSONL);
```

Optional `ID_COLUMN` and `GEOMETRY_COLUMN` pick which columns become the Feature `id` and geometry. `st_asgeojson` / `st_geomfromgeojson` convert a single value. Reading GeoJSON is documented on the Loading JSON page.

JSON Type

DuckDB supports `json` via the JSON logical type. For example:

```
SELECT '[1, null, {"key": "value"}]'::JSON;
```

```
[1, null, {"key": "value"}]
```

Logically, the JSON type is similar to a VARCHAR, but with the restriction that it must be valid JSON. Physically, the data is stored as a VARCHAR.

For example, you can't parse invalid JSON:

```
SELECT 'unquoted'::JSON;
```

```
Conversion Error: Malformed JSON at byte 0 of input: unexpected character.  Input: "unquoted"
```

Instead, what you probably want here is `SELECT '"quoted"'::JSON`.

Since the data is stored physically as a VARCHAR, whitespace is significant:

```
SELECT '{ "a": 5 }'::JSON = '{"a":5}'::JSON;
```

```
false
```

Please note that whitespaces are kept in roundtrips:

```
SELECT '{  "a":5 }'::JSON::VARCHAR
```

```
{  "a":5 }
```

The order of keys in objects is significant:

```
 SELECT '{"a":1,"b":2}'::JSON = '{"b":2,"a":1}'::JSON;
```

```
false
```

Duplicate keys are allowed in JSON objects:

```
SELECT '{"a":1,"a":2}'::JSON;
```

```
{"a":1,"a":2}
```

We allow any of DuckDB's types to be cast to JSON, and JSON to be cast back to any of DuckDB's types, for example, to cast `JSON` to DuckDB's STRUCT type, run:

```
SELECT '{"duck": 42}'::JSON::STRUCT(duck INTEGER);
```

```
{'duck': 42}
```

And back:

```
SELECT {duck: 42}::JSON;
```

```
{"duck":42}
```

This works for our nested types as shown in the example, but also for non-nested types:

```
SELECT '2023-05-12'::DATE::JSON;
```

```
"2023-05-12"
```

The only exception to this behavior is the cast from VARCHAR to `JSON`, which does not alter the data, but instead parses and validates the contents of the VARCHAR as JSON.

JSON Processing Functions

JSON Extraction Functions

There are two extraction functions, which have their respective operators. The operators can only be used if the string is stored as the `JSON` logical type. These functions support the same two location notations as JSON Scalar functions.

Function	Alias	Operator	Description
`json_exists(json, path)`			Returns `true` if the supplied path exists in the `json`, and `false` otherwise.
`json_extract(json, path)`	`json_extract_path`	`->`	Extracts JSON from `json` at the given `path`. If `path` is a `LIST`, the result will be a `LIST` of `JSON`.

Function	Alias	Operator	Description
`json_extract_string(json, path)`	`json_extract_path_text`	`->>`	Extracts VARCHAR from `json` at the given `path`. If `path` is a `LIST`, the result will be a `LIST` of VARCHAR.
`json_value(json, path)`			Extracts JSON from `json` at the given `path`. If the `json` at the supplied path is not a scalar value, it will return `NULL`.

Note that the arrow operator `->`, which is used for JSON extracts, has a low precedence as it is also used in lambda functions. Therefore, you need to surround the `->` operator with parentheses when expressing operations such as equality comparisons (=). For example:

```
SELECT ((JSON '{"field": 42}')->'field') = 42;
```

Unlike in DuckDB 1.x, the arrow operators bind more tightly than `AND` and `OR`, so an extraction can be combined with other conditions in a `WHERE` clause without parentheses:

```
SELECT count(*) FROM (VALUES ('{}'::JSON)) t(j)
WHERE 1 = 1 AND j->>'field' IS NOT NULL;
```

> **Warning.** DuckDB's JSON data type uses 0-based indexing.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');

SELECT json_extract(j, '$.family') FROM example;
```

```
"anatidae"
```

```
SELECT j->'$.family' FROM example;
```

```
"anatidae"
```

```
SELECT j->'$.species[0]' FROM example;
```

```
"duck"
```

```
SELECT j->'$.species[*]' FROM example;
```

```
["duck", "goose", "swan", null]
```

```
SELECT j->>'$.species[*]' FROM example;
```

```
[duck, goose, swan, null]
```

```
SELECT j->'$.species'->0 FROM example;
```

```
"duck"
```

```
SELECT j->'species'->['/0', '/1'] FROM example;
```

```
['"duck"', '"goose"']
```

```
SELECT json_extract_string(j, '$.family') FROM example;
```

```
anatidae
```

```
SELECT j->>'$.family' FROM example;
```

```
anatidae
```

```
SELECT j->>'$.species[0]' FROM example;
```

```
duck
```

```
SELECT j->'species'->>0 FROM example;
```

```
duck
```

```
SELECT j->'species'->>['/0', '/1'] FROM example;
```

```
[duck, goose]
```

Note that DuckDB's JSON data type uses 0-based indexing.

If multiple values need to be extracted from the same JSON, it is more efficient to extract a list of paths:

The following will cause the JSON to be parsed twice:

Resulting in a slower query that uses more memory:

```
SELECT
    json_extract(j, 'family') AS family,
    json_extract(j, 'species') AS species
FROM example;
```

family	species
”anatidae”	[”duck”,”goose”,”swan”,null]

The following produces the same result but is faster and more memory-efficient:

```
WITH extracted AS (
    SELECT json_extract(j, ['family', 'species']) AS extracted_list
    FROM example
)
SELECT
    extracted_list[1] AS family,
    extracted_list[2] AS species
FROM extracted;
```

Processing Extracted Arrays with Lambda Functions

When a path with a wildcard such as `[*]` extracts a JSON array into a `LIST`, the result can be processed further with lambda functions. For example, use `list_transform` to uppercase each extracted species name:

```
SELECT list_transform(j->>'$.species[*]', lambda s: upper(s)) AS names
FROM example;
```

names
[DUCK, GOOSE, SWAN, NULL]

To drop the `NULL` entries and combine the remaining names into a single string, chain `list_filter` and `list_reduce`:

```
SELECT list_reduce(
    list_filter(j->>'$.species[*]', lambda s: s IS NOT NULL),
    lambda acc, s: acc || ', ' || s
) AS names
FROM example;
```

names
duck, goose, swan

JSON Scalar Functions

The following scalar JSON functions can be used to gain information about the stored JSON values. With the exception of `json_valid(json)`, all JSON functions produce an error when invalid JSON is supplied.

We support two kinds of notations to describe locations within JSON: JSON Pointer and JSONPath.

Function	Description
`json_array_length(json[, path])`	Return the number of elements in the JSON array `json`, or `0` if it is not a JSON array. If `path` is specified, return the number of elements in the JSON array at the given `path`. If `path` is a `LIST`, the result will be `LIST` of array lengths.
`json_contains(json_haystack, json_needle)`	Returns `true` if `json_needle` is contained in `json_haystack`. Both parameters are of JSON type, but `json_needle` can also be a numeric value or a string, however the string must be wrapped in double quotes.
`json_keys(json[, path])`	Returns the keys of `json` as a `LIST` of `VARCHAR`, if `json` is a JSON object. If `path` is specified, return the keys of the JSON object at the given `path`. If `path` is a `LIST`, the result will be `LIST` of `LIST` of `VARCHAR`.
`json_pretty(json)`	Return `json` in a pretty-printed, indented format.
`json_structure(json)`	Return the structure of `json`. Defaults to `JSON` if the structure is inconsistent (e.g., incompatible types in an array).
`json_type(json[, path])`	Return the type of the supplied `json`, which is one of `ARRAY`, `BIGINT`, `BOOLEAN`, `DOUBLE`, `OBJECT`, `UBIGINT`, `VARCHAR` and `NULL`. If `path` is specified, return the type of the element at the given `path`. If `path` is a `LIST`, the result will be `LIST` of types.
`json_valid(json)`	Return whether `json` is valid JSON.
`json(json)`	Parse and minify `json`.

The JSONPointer syntax separates each field with a `/`. For example, to extract the first element of the array with key `duck`, you can do:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '/duck/0');
```

```
1
```

The JSONPath syntax separates fields with a `.`, and accesses array elements with `[i]`, and always starts with `$`. Using the same example, we can do the following:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[0]');
```

```
1
```

Note that DuckDB's JSON data type uses 0-based indexing.

JSONPath is more expressive, and can also access from the back of lists:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[#-1]');
```

```
3
```

JSONPath also allows escaping syntax tokens, using double quotes:

```
SELECT json_extract('{"duck.goose": [1, 2, 3]}', '$."duck.goose"[1]');
```

```
2
```

Examples using the anatidae biological family:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');
```

```
SELECT json(j) FROM example;
```

```
{"family":"anatidae","species":["duck","goose","swan",null]}
```

```
SELECT j.family FROM example;
```

```
"anatidae"
```

```
SELECT j.species[0] FROM example;
```

```
"duck"
```

```
SELECT json_valid(j) FROM example;
```

```
true
```

```
SELECT json_valid('{');
```

```
false
```

```
SELECT json_array_length('["duck", "goose", "swan", null]');
```

```
4
```

```
SELECT json_array_length(j, 'species') FROM example;
```

```
4
```

```
SELECT json_array_length(j, '/species') FROM example;
```

```
4
```

```
SELECT json_array_length(j, '$.species') FROM example;
```

```
4
```

```
SELECT json_array_length(j, ['$.species']) FROM example;
```

```
[4]
```

```
SELECT json_type(j) FROM example;
```

```
OBJECT
```

```
SELECT json_keys(j) FROM example;
```

```
[family, species]
```

```
SELECT json_structure(j) FROM example;
```

```
{"family":"VARCHAR","species":["VARCHAR"]}
```

```
SELECT json_structure('["duck", {"family": "anatidae"}]');
```

```
["JSON"]
```

> Because JSON has no native temporal or UUID types, `json_structure` reports `DATE`, `TIME`, `TIMESTAMP`, and `UUID` values as `VARCHAR`, since that is how they are represented in JSON. If you feed this structure back into `from_json`, these columns are produced as `VARCHAR`. To recover the original types, replace the relevant `"VARCHAR"` entries in the structure with the desired type name, for example `"DATE"`.

```
SELECT json_contains('{"key": "value"}', '"value"');
```

```
true
```

```
SELECT json_contains('{"key": 1}', '1');
```

```
true
```

```
SELECT json_contains('{"top_key": {"key": "value"}}', '{"key": "value"}');
```

```
true
```

JSON Patching and Normalization Functions

The following scalar functions compute, apply and clean up JSON patches. They complement the `json_merge_patch` function, which applies a RFC 7396 merge patch to a JSON document. For an end-to-end example of using these functions in a data reconciliation pipeline, see the "Reconciling JSON in DuckDB, One Patch at a Time" blog post.

Function	Description
`json_deep_merge(json, patch[, ...])`	Recursively merge one or more `patch` documents into `json`, applied left to right. Works like `json_merge_patch`, except that a JSON `null` value in a patch keeps the original value instead of deleting the key. A SQL `NULL` patch returns `NULL`, while a SQL `NULL` `json` is ignored.
`json_merge_patch_diff(json_orig, json_modified)`	Return the minimal RFC 7396 merge patch such that `json_merge_patch(json_orig, patch)` equals `json_modified`. Deleted keys appear as `null`, changed and added keys appear with their new values, and unchanged keys are omitted. Nested objects are diffed recursively.
`json_normalize(json)`	Return `json` in a canonical form by recursively sorting the keys of every object, including objects nested inside arrays. The order of array elements is preserved and scalar values are returned unchanged.
`json_strip_nulls(json)`	Recursively remove all keys whose value is JSON `null`, including in objects nested inside arrays. `null` array elements are kept and scalar values are returned unchanged.

Compute the patch between two documents:

```
SELECT json_merge_patch_diff('{"a": 1, "b": 2, "c": 3}', '{"a": 1, "b": 99, "d": 4}');
```

```
{"c":null,"b":99,"d":4}
```

Applying the diff using `json_merge_patch` reconstructs the modified document:

```
SELECT json_merge_patch(
    '{"a": 1, "b": 2, "c": 3}',
    json_merge_patch_diff('{"a": 1, "b": 2, "c": 3}', '{"a": 1, "b": 99, "d": 4}')
) = '{"a":1,"b":99,"d":4}' AS round_trips;
```

```
true
```

`json_deep_merge` treats JSON `null` values in the patch as "keep the original value", while `json_merge_patch` deletes the key:

```
SELECT
    json_merge_patch('{"a": {"x": 1, "y": 2}}', '{"a": {"y": null, "z": 3}}') AS merge_patch,
    json_deep_merge('{"a": {"x": 1, "y": 2}}', '{"a": {"y": null, "z": 3}}') AS deep_merge;
```

merge_patch	deep_merge
{”a”:{”x”:1,”z”:3}}	{”a”:{”x”:1,”y”:2,”z”:3}}

`json_deep_merge` accepts any number of patches:

```
SELECT json_deep_merge('{"a": 1}', '{"a": null}', '{"a": 2}');
```

```
{"a":2}
```

Normalize the key order of a document, e.g., to compare or hash documents that only differ in key order:

```
SELECT json_normalize('{"c": {"b": {"z": 1, "a": 2}, "a": 3}, "a": 4}');
```

```
{"a":4,"c":{"a":3,"b":{"a":2,"z":1}}}
```

```
SELECT md5(json_normalize('{"z": 1, "a": 2}')) = md5(json_normalize('{"a": 2, "z": 1}')) AS same_hash;
```

```
true
```

Remove `null`-valued keys recursively:

```
SELECT json_strip_nulls('{"a": [{"x": 1, "y": null}, {"z": null}], "b": null}');
```

```
{"a":[{"x":1},{}]}
```

JSON Mutation Functions

> New Support for the JSON mutation functions was introduced in DuckDB 2.0.

The following scalar functions modify a JSON document at a given path and return the updated document. The `value` argument is a JSON value given as JSON text, e.g., `'2'` for the number 2 or `'"duck"'` for the string `duck`.

Function	Description
`json_set(json, path, value)`	Set the `value` at `path`, inserting it if the path does not exist and overwriting it if it does.
`json_insert(json, path, value)`	Insert the `value` at `path` only if the path does not already exist. Existing values are left unchanged.
`json_replace(json, path, value)`	Replace the `value` at `path` only if the path already exists. Missing paths are left unchanged.
`json_remove(json, path)`	Remove the element at `path`.

Set a key, inserting it because it does not exist yet:

```
SELECT json_set('{"a": 1}', '$.b', '2');
```

```
{"a":1,"b":2}
```

`json_insert` only adds new keys, while `json_replace` only updates existing keys:

```
SELECT
    json_insert('{"a": 1}', '$.a', '9') AS insert,
    json_replace('{"a": 1}', '$.b', '2') AS replace;
```

insert	replace
{”a”:1}	{”a”:1}

Remove a key:

```
SELECT json_remove('{"a": 1, "b": 2}', '$.b');
```

```
{"a":1}
```

JSON Aggregate Functions

There are three JSON aggregate functions.

Function	Description
`json_group_array(any)`	Return a JSON array with all values of `any` in the aggregation.
`json_group_object(key, value)`	Return a JSON object with all `key`, `value` pairs in the aggregation.
`json_group_structure(json)`	Return the combined `json_structure` of all `json` in the aggregation.

Examples:

```
CREATE TABLE example1 (k VARCHAR, v INTEGER);
INSERT INTO example1 VALUES ('duck', 42), ('goose', 7);
```

```
SELECT json_group_array(v) FROM example1;
```

```
[42, 7]
```

```
SELECT json_group_object(k, v) FROM example1;
```

```
{"duck":42,"goose":7}
```

```
CREATE TABLE example2 (j JSON);
INSERT INTO example2 VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');
```

```
SELECT json_group_structure(j) FROM example2;
```

```
{"family":"VARCHAR","species":["VARCHAR"],"coolness":"DOUBLE","hair":"BOOLEAN"}
```

Transforming JSON to Nested Types

In many cases, it is inefficient to extract values from JSON one-by-one. Instead, we can “extract” all values at once, transforming JSON to the nested types `LIST` and `STRUCT`.

Function	Description
`json_transform(json, structure)`	Transform `json` according to the specified `structure`.
`from_json(json, structure)`	Alias for `json_transform`.
`json_transform_strict(json, structure)`	Same as `json_transform`, but throws an error when type casting fails.

Function	Description
`from_json_strict(json, structure)`	Alias for `json_transform_strict`.

The `structure` argument is JSON of the same form as returned by `json_structure`. The `structure` argument can be modified to transform the JSON into the desired structure and types. It is possible to extract fewer key/value pairs than are present in the JSON, and it is also possible to extract more: missing keys become `NULL`.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');

SELECT json_transform(j, '{"family": "VARCHAR", "coolness": "DOUBLE"}') FROM example;

{'family': anatidae, 'coolness': 42.420000}
{'family': canidae, 'coolness': NULL}

SELECT json_transform(j, '{"family": "TINYINT", "coolness": "DECIMAL(4, 2)"}') FROM example;

{'family': NULL, 'coolness': 42.42}
{'family': NULL, 'coolness': NULL}

SELECT json_transform_strict(j, '{"family": "TINYINT", "coolness": "DOUBLE"}') FROM example;

Invalid Input Error:
Failed to cast value: "anatidae"
```

JSON Table Functions

DuckDB implements two JSON table functions that take a JSON value and produce a table from it.

Function	Description
`json_each(json[,path]`	Traverse `json` and return one row for each element in the top-level array or object.
`json_tree(json[,path]`	Traverse `json` in depth-first fashion and return one row for each element in the structure.

If the element is not an array or object, the element itself is returned. If the optional `path` argument is supplied, traversal starts from the element at the given path instead of the root element.

The resulting table has the following columns:

Field	Type	Description
`key`	`VARCHAR`	Key of element relative to its parent
`value`	`JSON`	Value of element
`type`	`VARCHAR`	`json_type` (function) of this element
`atom`	`JSON`	`json_value` (function) of this element
`id`	`UBIGINT`	Element identifier, numbered by parse order
`parent`	`UBIGINT`	`id` of parent element

Field	Type	Description
`fullkey`	VARCHAR	JSON path to element
`path`	VARCHAR	JSON path to parent element
`json`	JSON (Virtual)	The `json` parameter
`root`	TEXT (Virtual)	The `path` parameter
`rowid`	BIGINT (Virtual)	The row identifier

These functions are analogous to SQLite's functions with the same name. Note that, because the `json_each` and `json_tree` functions refer to previous subqueries in the same FROM clause, they are *lateral joins*.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');

SELECT je.*, je.rowid
FROM example AS e, json_each(e.j) AS je;
```

key	value	type	atom	id	parent	fullkey	path	rowid
family	”anatidae”	VARCHAR	”anatidae”	2	NULL	$.family	$	0
species	[”duck”,”goose”]	ARRAY	NULL	4	NULL	$.species	$	1
coolness	42.42	DOUBLE	42.42	8	NULL	$.coolness	$	2
family	”canidae”	VARCHAR	”canidae”	2	NULL	$.family	$	0
species	[”labrador”,”bulldog”]	ARRAY	NULL	4	NULL	$.species	$	1
hair	true	BOOLEAN	true	8	NULL	$.hair	$	2

```
SELECT je.*, je.rowid
FROM example AS e, json_each(e.j, '$.species') AS je;
```

key	value	type	atom	id	parent	fullkey	path	rowid
0	”duck”	VARCHAR	”duck”	5	NULL	$.species[0]	$.species	0
1	”goose”	VARCHAR	”goose”	6	NULL	$.species[1]	$.species	1
0	”labrador”	VARCHAR	”labrador”	5	NULL	$.species[0]	$.species	0
1	”bulldog”	VARCHAR	”bulldog”	6	NULL	$.species[1]	$.species	1

```
SELECT je.key, je.value, je.type, je.id, je.parent, je.fullkey, je.rowid
FROM example AS e, json_tree(e.j) AS je;
```

key	value	type	id	parent	fullkey	rowid
NULL	{”family”:”anatidae”,”species”:[”duck”,”goose”],”coolness”:42.42}	OBJECT	0	NULL	$	0
family	”anatidae”	VARCHAR	2	0	$.family	1
species	[”duck”,”goose”]	ARRAY	4	0	$.species	2
0	”duck”	VARCHAR	5	4	$.species[0]	3

key	value	type	id	parent	fullkey	rowid
1	”goose”	VARCHAR	6	4	$.species[1]	4
coolness	42.42	DOUBLE	8	0	$.coolness	5
NULL	{”family”:”canidae”,”species”:[”labrador”,”bulldog”],”hair”:true}	OBJECT	0	NULL	$	0
family	”canidae”	VARCHAR	2	0	$.family	1
species	[”labrador”,”bulldog”]	ARRAY	4	0	$.species	2
0	”labrador”	VARCHAR	5	4	$.species[0]	3
1	”bulldog”	VARCHAR	6	4	$.species[1]	4
hair	true	BOOLEAN	8	0	$.hair	5

JSON Format Settings

The JSON extension can attempt to determine the format of a JSON file when setting `format` to `auto`. Here are some example JSON files and the corresponding `format` settings that should be used.

In each of the below cases, the `format` setting was not needed, as DuckDB was able to infer it correctly, but it is included for illustrative purposes. A query of this shape would work in each case:

```
SELECT *
FROM filename.json;
```

Format: `newline_delimited`

With `format = 'newline_delimited'` newline-delimited JSON can be parsed. Each line is a JSON.

We use the example file `records.json` with the following content:

```
{"key1":"value1", "key2": "value1"}
{"key1":"value2", "key2": "value2"}
{"key1":"value3", "key2": "value3"}
```

```
SELECT *
FROM read_json('records.json', format = 'newline_delimited');
```

key1	key2
value1	value1
value2	value2
value3	value3

Format: `array`

If the JSON file contains a JSON array of objects (pretty-printed or not), `array_of_objects` may be used. To demonstrate its use, we use the example file `records-in-array.json`:

```
[
    {"key1":"value1", "key2": "value1"},
    {"key1":"value2", "key2": "value2"},
    {"key1":"value3", "key2": "value3"}
]
```

```
SELECT *
FROM read_json('records-in-array.json', format = 'array');
```

key1	key2
value1	value1
value2	value2
value3	value3

Format: `unstructured`

If the JSON file contains JSON that is not newline-delimited or an array, `unstructured` may be used. To demonstrate its use, we use the example file `unstructured.json`:

```
{
    "key1":"value1",
    "key2":"value1"
}
{
    "key1":"value2",
    "key2":"value2"
}
{
    "key1":"value3",
    "key2":"value3"
}
```

```
SELECT *
FROM read_json('unstructured.json', format = 'unstructured');
```

key1	key2
value1	value1
value2	value2
value3	value3

`records` Options

The JSON extension can attempt to determine whether a JSON file contains records when setting `records = auto`. When `records = true`, the JSON extension expects JSON objects, and will unpack the fields of JSON objects into individual columns.

Continuing with the same example file, `records.json`:

```
{"key1":"value1", "key2": "value1"}
{"key1":"value2", "key2": "value2"}
{"key1":"value3", "key2": "value3"}
```

```
SELECT *
FROM read_json('records.json', records = true);
```

key1	key2
value1	value1
value2	value2

key1	key2
value3	value3

When `records = false`, the JSON extension will not unpack the top-level objects, and create STRUCTs instead:

```
SELECT *
FROM read_json('records.json', records = false);
```

json
{'key1': value1, 'key2': value1}
{'key1': value2, 'key2': value2}
{'key1': value3, 'key2': value3}

This is especially useful if we have non-object JSON, for example, `arrays.json`:

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
SELECT *
FROM read_json('arrays.json', records = false);
```

json
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]

Installing and Loading the JSON Extension

The `json` extension is shipped by default in DuckDB builds, otherwise, it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL json;
LOAD json;
```

SQL to/from JSON

DuckDB provides functions to serialize and deserialize `SELECT` statements between SQL and JSON, as well as executing JSON serialized statements.

Function	Type	Description
`json_deserialize_sql(json)`	Scalar	Deserialize one or many `json` serialized statements back to an equivalent SQL string.
`json_execute_serialized_sql(varchar)`	Table	Execute `json` serialized statements and return the resulting rows. Only one statement at a time is supported for now.

Function	Type	Description
`json_serialize_plan(varchar, skip_default := boolean, format := boolean, optimize := boolean)`	Scalar	Serialize the bound logical query plan of a set of semicolon-separated (`;`) statements to JSON. If `optimize` is `true`, the optimized plan is serialized.
`json_serialize_sql(varchar, skip_default := boolean, skip_empty := boolean, skip_null := boolean, format := boolean)`	Scalar	Serialize a set of semicolon-separated (`;`) select statements to an equivalent list of `json` serialized statements.
`PRAGMA json_execute_serialized_sql(varchar)`	Pragma	Pragma version of the `json_execute_serialized_sql` function.

The `json_serialize_sql(varchar)` function takes three optional parameters, `skip_empty`, `skip_null` and `format` that can be used to control the output of the serialized statements.

If you run the `json_execute_serialized_sql(varchar)` table function inside of a transaction the serialized statements will not be able to see any transaction local changes. This is because the statements are executed in a separate query context. You can use the `PRAGMA json_execute_serialized_sql(varchar)` pragma version to execute the statements in the same query context as the pragma, although with the limitation that the serialized JSON must be provided as a constant string, i.e., you cannot do `PRAGMA json_execute_serialized_sql(json_serialize_sql(...))`.

Note that these functions do not preserve syntactic sugar such as `FROM * SELECT ...`, so a statement round-tripped through `json_deserialize_sql(json_serialize_sql(...))` may not be identical to the original statement, but should always be semantically equivalent and produce the same output.

Examples

Simple example:

```
SELECT json_serialize_sql('SELECT 2');
```

```
{"error":false,"statements":[{"node":{"type":"SELECT_NODE","modifiers":[],"cte_map":{"map":[]},"select_
list":[{"class":"CONSTANT","type":"VALUE_CONSTANT","alias":"","query_location":7,"query_location_
length":1,"literal":{"kind":"INTEGER","text":"2"}}],"from_
table":{"type":"EMPTY","alias":"","sample":null,"query_location":18446744073709551615,"query_location_
length":0},"where_clause":null,"group_expressions":[],"group_sets":[],"aggregate_handling":"STANDARD_
HANDLING","having":null,"sample":null,"qualify":null},"named_param_map":[]}]}
```

Example with multiple statements and skip options:

```
SELECT json_serialize_sql('SELECT 1 + 2; SELECT a + b FROM tbl1', skip_empty := true, skip_null := true);
```

```
{"error":false,"statements":[{"node":{"type":"SELECT_NODE","cte_map":{},"select_
list":[{"class":"FUNCTION","type":"FUNCTION","query_location":9,"query_location_length":0,"function_
name":"+","order_bys":{"type":"ORDER_MODIFIER"},"distinct":false,"is_operator":true,"export_
state":false,"arguments":[{"expression":{"class":"CONSTANT","type":"VALUE_CONSTANT","query_
location":7,"query_location_
length":1,"literal":{"kind":"INTEGER","text":"1"}}},{"expression":{"class":"CONSTANT","type":"VALUE_
CONSTANT","query_location":11,"query_location_
length":1,"literal":{"kind":"INTEGER","text":"2"}}}],"qualified_name":{"path":["+"]}}],"from_
table":{"type":"EMPTY","query_location":18446744073709551615,"query_location_length":0},"aggregate_
handling":"STANDARD_HANDLING"}},{"node":{"type":"SELECT_NODE","cte_map":{},"select_
list":[{"class":"FUNCTION","type":"FUNCTION","query_location":23,"query_location_length":0,"function_
name":"+","order_bys":{"type":"ORDER_MODIFIER"},"distinct":false,"is_operator":true,"export_
state":false,"arguments":[{"expression":{"class":"COLUMN_REF","type":"COLUMN_REF","query_
location":21,"query_location_length":1,"column_names":["a"]}},{"expression":{"class":"COLUMN_
REF","type":"COLUMN_REF","query_location":25,"query_location_length":1,"column_names":["b"]}}],"qualified_
name":{"path":["+"]}}],"from_table":{"type":"BASE_TABLE","query_location":32,"query_location_
length":4,"table_name":"tbl1","qualified_name":{"path":["tbl1"]}},"aggregate_handling":"STANDARD_
HANDLING"}}]}
```

Skip the default values in the AST (e.g., `"distinct":false`):

```
SELECT json_serialize_sql('SELECT 1 + 2; SELECT a + b FROM tbl1', skip_default := true, skip_empty := true,
skip_null := true);
```

```
{"error":false,"statements":[{"node":{"type":"SELECT_NODE","cte_map":{},"select_
list":[{"class":"FUNCTION","type":"FUNCTION","query_location":9,"function_name":"+","order_
bys":{"type":"ORDER_MODIFIER"},"is_
operator":true,"arguments":[{"expression":{"class":"CONSTANT","type":"VALUE_CONSTANT","query_
location":7,"query_location_
length":1,"literal":{"kind":"INTEGER","text":"1"}}},{"expression":{"class":"CONSTANT","type":"VALUE_
CONSTANT","query_location":11,"query_location_
length":1,"literal":{"kind":"INTEGER","text":"2"}}}],"qualified_name":{"path":["+"]}}],"from_
table":{"type":"EMPTY"},"aggregate_handling":"STANDARD_HANDLING"}},{"node":{"type":"SELECT_NODE","cte_
map":{},"select_list":[{"class":"FUNCTION","type":"FUNCTION","query_location":23,"function_
name":"+","order_bys":{"type":"ORDER_MODIFIER"},"is_
operator":true,"arguments":[{"expression":{"class":"COLUMN_REF","type":"COLUMN_REF","query_
location":21,"query_location_length":1,"column_names":["a"]}},{"expression":{"class":"COLUMN_
REF","type":"COLUMN_REF","query_location":25,"query_location_length":1,"column_names":["b"]}}],"qualified_
name":{"path":["+"]}}],"from_table":{"type":"BASE_TABLE","query_location":32,"query_location_
length":4,"table_name":"tbl1","qualified_name":{"path":["tbl1"]}},"aggregate_handling":"STANDARD_
HANDLING"}}]}
```

Example with a syntax error:

```
SELECT json_serialize_sql('TOTALLY NOT VALID SQL');
```

```
{"error":true,"error_type":"parser","error_message":"syntax error at or near
\"VALID\"","location":"[12,5]","position":"12","error_subtype":"SYNTAX_ERROR"}
```

Example with deserialize:

```
SELECT json_deserialize_sql(json_serialize_sql('SELECT 1 + 2'));
```

```
SELECT (1 + 2)
```

Example with deserialize and syntax sugar, which is lost during the transformation:

```
SELECT json_deserialize_sql(json_serialize_sql('FROM x SELECT 1 + 2'));
```

```
SELECT (1 + 2) FROM x
```

Example with execute:

```
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('SELECT 1 + 2'));
```

```
3
```

Example with error:

```
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('TOTALLY NOT VALID SQL'));
```

```
Parser Error:
Error parsing json: parser: syntax error at or near "VALID"
```

Serializing the bound logical plan of a statement with `json_serialize_plan`:

```
SELECT json_serialize_plan('SELECT 1 + 2', format := true);
```

```
{
    "error": false,
    "plans": [
        {
            "type": "LOGICAL_PROJECTION",
            ...
        }
    ]
}
```

Caveats

Equality Comparison

> **Warning.** Currently, equality comparison of JSON files can differ based on the context. In some cases, it is based on raw text comparison, while in other cases, it uses logical content comparison.

The following query returns true for all fields:

```
SELECT
    a != b, -- Space is part of physical JSON content. Despite equal logical content, values are treated as not equal.
    c != d, -- Same.
    c[0] = d[0], -- Equality because space was removed from physical content of fields:
    a = c[0], -- Indeed, field is equal to empty list without space...
    b != c[0], -- ... but different from empty list with space.
FROM (
    SELECT
        '[]'::JSON AS a,
        '[ ]'::JSON AS b,
        '[[]]'::JSON AS c,
        '[[ ]]'::JSON AS d
    );
```

(a != b)	(c != d)	(c[0] = d[0])	(a = c[0])	(b != c[0])
true	true	true	true	true

Multiple Files

Reading Multiple Files

DuckDB can read multiple files of different types (CSV, Parquet, JSON files) at the same time using either the glob syntax, or by providing a list of files to read. See the combining schemas page for tips on reading files with different schemas.

CSV

Read all files with a name ending in `.csv` in the folder `dir`:

```
SELECT *
FROM 'dir/*.csv';
```

Read all files with a name ending in `.csv`, two directories deep:

```
SELECT *
FROM '*/*/*.csv';
```

Read all files with a name ending in `.csv`, at any depth in the folder `dir`:

```
SELECT *
FROM 'dir/**/*.csv';
```

Read the CSV files `flights1.csv` and `flights2.csv`:

```
SELECT *
FROM read_csv(['flights1.csv', 'flights2.csv']);
```

Read the CSV files `flights1.csv` and `flights2.csv`, unifying schemas by name and outputting a `filename` column:

```
SELECT *
FROM read_csv(['flights1.csv', 'flights2.csv'], union_by_name = true, filename = true);
```

Parquet

Read all files that match the glob pattern:

```
SELECT *
FROM 'test/*.parquet';
```

Read three Parquet files and treat them as a single table:

```
SELECT *
FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);
```

Read all Parquet files from two specific folders:

```
SELECT *
FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

Read all Parquet files that match the glob pattern at any depth:

```
SELECT *
FROM read_parquet('dir/**/*.parquet');
```

Multi-File Reads and Globs

DuckDB can also read a series of Parquet files and treat them as if they were a single table. Note that this only works if the Parquet files have the same schema. You can specify which Parquet files you want to read using a list parameter, glob pattern matching syntax, or a combination of both.

Directory Paths

The `read_parquet` function accepts a local directory path and recursively reads files with the `.parquet` extension, including files in subdirectories:

```
SELECT *
FROM read_parquet('dir');
```

This reads the same files as `read_parquet('dir/**/*.parquet')`. A trailing slash on the directory path is optional. In contrast, `read_parquet('dir/*.parquet')` reads only files directly inside `dir`.

Files with other extensions are ignored by directory discovery, even if they contain Parquet data. To read those files, supply their filenames explicitly. If the directory contains no matching files, the query raises an error.

Directory paths do not change how schemas are combined. When files have different columns, use `union_by_name = true` as described in combining schemas.

List Parameter

The `read_parquet` function can accept a list of filenames as the input parameter.

Read three Parquet files and treat them as a single table:

```
SELECT *
FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);
```

Glob Syntax

Any file name input to the `read_parquet` function can either be an exact filename, or use a glob syntax to read multiple files that match a pattern.

Wildcard	Description
`*`	Matches any number of any characters (including none)
`**`	Matches any number of subdirectories (including none)
`?`	Matches any single character
`[abc]`	Matches one character given in the bracket
`[a-z]`	Matches one character from the range given in the bracket

Note that the `?` wildcard in globs is not supported for reads over S3 due to HTTP encoding issues.

Here is an example that reads all the files that end with `.parquet` located in the `test` folder:

Read all files that match the glob pattern:

```
SELECT *
FROM read_parquet('test/*.parquet');
```

Handling Patterns That Match No Files If a glob pattern does not match any files, the read functions raise an error:

```
SELECT *
FROM read_parquet('test/*.parquet');
```

```
IO Error:
No files found that match the pattern "test/*.parquet"
```

Starting with DuckDB 2.0, the CSV, JSON and Parquet readers accept the `allow_empty` option, which returns an empty result (with a single `empty` column of type BOOLEAN) instead of raising an error:

```
SELECT *
FROM read_parquet('test/*.parquet', allow_empty = true);
```

Alternatively, use the `glob` function to check whether any files match before reading. Unlike the read functions, `glob` returns an empty result instead of raising an error when nothing matches:

```
SELECT count(*) AS matches
FROM glob('test/*.parquet');
```

matches
0

Your application can then skip the read when `matches` is `0`.

List of Globs

The glob syntax and the list input parameter can be combined to scan files that meet one of multiple patterns.

Read all Parquet files from 2 specific folders.

```
SELECT *
FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

DuckDB can read multiple CSV files at the same time using either the glob syntax, or by providing a list of files to read.

Filename

The `filename` argument can be used to add an extra `filename` column to the result that indicates which row came from which file. For example:

```
SELECT *
FROM read_csv(['flights3.csv', 'flights4.csv'], union_by_name = true, filename = true);
```

FlightDate	OriginCityName	DestCityName	UniqueCarrier	filename
1988-01-01	New York, NY	Los Angeles, CA	NULL	flights3.csv
1988-01-02	New York, NY	Los Angeles, CA	NULL	flights3.csv
1988-01-03	New York, NY	Los Angeles, CA	AA	flights4.csv

> The `filename` argument also accepts a string (e.g., `filename = 'input_file'`). When provided, the string is used as the name of the added column. This is useful when the source data already contains a `filename` column and you want to avoid a name collision.

Glob Function to Find Filenames

The glob pattern matching syntax can also be used to search for filenames using the `glob` table function. It accepts one parameter: the path to search (which may include glob patterns).

Search the current directory for all files.

```
SELECT *
FROM glob('*');
```

file
test.csv
test.json
test.parquet
test2.csv
test2.parquet
todos.json

Combining Schemas

Examples

Read a set of CSV files combining columns by position:

```
SELECT * FROM read_csv('flights*.csv');
```

Read a set of CSV files combining columns by name:

```
SELECT * FROM read_csv('flights*.csv', union_by_name = true);
```

Combining Schemas

When reading from multiple files, we have to **combine schemas** from those files. That is because each file has its own schema that can differ from the other files. DuckDB offers two ways of unifying schemas of multiple files: **by column position** and **by column name**.

By default, DuckDB reads the schema of the first file provided, and then matches the columns of subsequent files to it by column name: columns missing from the first file are dropped, and columns of the first file missing from a later file cause an error. This works correctly as long as all files have the same schema. If the schema of the files differs, you might want to use the `union_by_name` option to allow DuckDB to construct the schema by reading all of the names instead.

Below is an example of how both methods work.

Union by Position

By default, DuckDB uses the schema of the first file and reads the columns of the subsequent files into it. For example, consider the following two files.

`flights1.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-01|AA|New York, NY|Los Angeles, CA
1988-01-02|AA|New York, NY|Los Angeles, CA
```

`flights2.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-03|AA|New York, NY|Los Angeles, CA
```

Reading the two files at the same time will produce the following result set:

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

This is equivalent to the SQL construct `UNION ALL`.

Union by Name

If you are processing multiple files that have different schemas, perhaps because columns have been added or renamed, it might be desirable to unify the columns of different files **by name** instead. This can be done by providing the `union_by_name` option. For example, consider the following two files, where `flights4.csv` has an extra column (`UniqueCarrier`).

`flights3.csv`:

```
FlightDate|OriginCityName|DestCityName
1988-01-01|New York, NY|Los Angeles, CA
1988-01-02|New York, NY|Los Angeles, CA
```

`flights4.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-03|AA|New York, NY|Los Angeles, CA
```

Reading these without `union_by_name` uses the schema of the first file, so the `UniqueCarrier` column of `flights4.csv` is silently dropped (reading them in the opposite order results in a schema mismatch error). When specifying the `union_by_name` option, the columns are correctly unified, and any missing values are set to `NULL`.

```
SELECT * FROM read_csv(['flights3.csv', 'flights4.csv'], union_by_name = true);
```

FlightDate	OriginCityName	DestCityName	UniqueCarrier
1988-01-01	New York, NY	Los Angeles, CA	NULL
1988-01-02	New York, NY	Los Angeles, CA	NULL
1988-01-03	New York, NY	Los Angeles, CA	AA

This is equivalent to the SQL construct `UNION ALL BY NAME`.

> Using the `union_by_name` option increases memory consumption.

Parquet Files

Reading and Writing Parquet Files

Examples

Read a single Parquet file:

```
SELECT * FROM 'test.parquet';
```

Figure out which columns/types are in a Parquet file:

```
DESCRIBE SELECT * FROM 'test.parquet';
```

Create a table from a Parquet file:

```
CREATE TABLE test AS
    SELECT * FROM 'test.parquet';
```

If the file does not end in `.parquet`, use the `read_parquet` function:

```
SELECT *
FROM read_parquet('test.parq');
```

Use list parameter to read three Parquet files and treat them as a single table:

```
SELECT *
FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);
```

Read Parquet files from a local directory, including its subdirectories:

```
SELECT *
FROM read_parquet('test');
```

A directory path recursively reads files with the `.parquet` extension. To read only files directly inside the directory, use `test/*.parquet` instead. See directory paths for details.

Read all files that match the glob pattern:

```
SELECT *
FROM 'test/*.parquet';
```

Read all files that match the glob pattern, and include the `filename` virtual column that specifies which file each row came from (this column is available by default without a configuration options since DuckDB 1.3.0):

```
SELECT *, filename
FROM read_parquet('test/*.parquet');
```

Use a list of globs to read all Parquet files from two specific folders:

```
SELECT *
FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

Read over HTTPS:

```
SELECT *
FROM read_parquet('https://some.url/some_file.parquet');
```

Query the metadata of a Parquet file:

```sql
SELECT *
FROM parquet_metadata('test.parquet');
```

Query the file metadata of a Parquet file:

```sql
SELECT *
FROM parquet_file_metadata('test.parquet');
```

Query the key-value metadata of a Parquet file:

```sql
SELECT *
FROM parquet_kv_metadata('test.parquet');
```

Query the schema of a Parquet file:

```sql
SELECT *
FROM parquet_schema('test.parquet');
```

Write the results of a query to a Parquet file using the default compression (Snappy):

```sql
COPY
    (SELECT * FROM tbl)
    TO 'result-snappy.parquet'
    (FORMAT parquet);
```

Write the results from a query to a Parquet file with specific compression and row group size:

```sql
COPY
    (FROM generate_series(100_000))
    TO 'test.parquet'
    (FORMAT parquet, COMPRESSION zstd, ROW_GROUP_SIZE 100_000);
```

Export the table contents of the entire database as parquet:

```sql
EXPORT DATABASE 'target_directory' (FORMAT parquet);
```

Parquet Files

Parquet files are compressed columnar files that are efficient to load and process. DuckDB provides support for both reading and writing Parquet files in an efficient manner, as well as support for pushing filters and projections into the Parquet file scans.

> Parquet datasets differ based on the number of files, the size of individual files, the compression algorithm used, row group size, etc. These have a significant effect on performance. Please consult the Performance Guide for details.

`read_parquet` Function

Function	Description	Example
`read_parquet(path_or_list_of_paths)`	Read Parquet file(s)	`SELECT * FROM read_parquet('test.parquet');`
`parquet_scan(path_or_list_of_paths)`	Alias for `read_parquet`	`SELECT * FROM parquet_scan('test.parquet');`

If your file ends in `.parquet`, the function syntax is optional. The system will automatically infer that you are reading a Parquet file:

```sql
SELECT * FROM 'test.parquet';
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Parameters

There are a number of options exposed that can be passed to the `read_parquet` function or the COPY statement.

Name	Description	Type	Default
`allow_empty`	If the file pattern or file list matches no files, return an empty result (with a single `empty` column of type `BOOL`) instead of throwing an error. Added in DuckDB 2.0.	`BOOL`	`false`
`binary_as_string`	Parquet files generated by legacy writers do not correctly set the `UTF8` flag for strings, causing string columns to be loaded as `BLOB` instead. Set this to true to load binary columns as strings.	`BOOL`	`false`
`bound_schema`	The complete, ordered output schema of the scan as a struct of column names and types (e.g., `{'a': 'BIGINT', 'b': 'VARCHAR'}`). Columns are matched to the file columns by name and cast to the given types; file columns that are not listed are not returned and every listed column must exist in the files. Cannot be combined with `schema`, `union_by_name` or `allow_empty`, and requires `hive_partitioning` to be set explicitly. Added in DuckDB 2.0.	`STRUCT`	-
`can_have_nan`	Whether `FLOAT` and `DOUBLE` columns may contain `NaN` values. When set to true, the reader accounts for `NaN` when using a column's min/max statistics for filter pushdown, since `NaN` does not compare as ordered.	`BOOL`	`false`
`encryption_config`	Configuration for Parquet encryption.	`STRUCT`	-
`filename`	Whether or not an extra `filename` column should be included in the result. Since DuckDB 1.3.0, the `filename` column is added automatically as a virtual column and this option is only kept for compatibility reasons.	`BOOL`	`false`
`file_row_number`	Whether or not to include the `file_row_number` column.	`BOOL`	`false`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	`BOOL`	(auto-detected)
`int96_as`	How to read `INT96` columns (a deprecated timestamp encoding used by legacy writers such as Impala and older Spark versions). Options are `timestamp` (microsecond precision), `timestamp_ns` (preserves nanoseconds; out-of-range values saturate) and `struct` (a `STRUCT(date DATE, time TIME_NS)` that preserves the full range and precision). Added in DuckDB 2.0.	`VARCHAR`	`timestamp`
`prefetch_strategy`	Strategy for prefetching column data of row groups (applies unless prefetching is disabled with the `disable_parquet_prefetching` setting). With `auto`, DuckDB picks a strategy at runtime based on the projections and filters of the query. With `whole_group`, the entire row group is always prefetched. Added in DuckDB 2.0.	`VARCHAR`	`auto`

Name	Description	Type	Default
`union_by_name`	Whether the columns of multiple schemas should be unified by name, rather than using the schema of the first file.	`BOOL`	`false`
`schema`	Allows you to read a Parquet file as if it has the supplied schema. The `MAP` can be keyed by field-id (`INTEGER`) or by column name (`VARCHAR`).	`MAP`	`NULL`
`utf8_validation`	How to handle invalid UTF-8 in string columns. With `strict`, an error is thrown. With `replace`, invalid bytes are replaced by `?`. With `ignore`, invalid bytes are removed. Added in DuckDB 2.0.	`VARCHAR`	`strict`

Using the `schema` Parameter

The `schema` parameter allows you to read the Parquet file using a specific schema. This is useful for renaming, adding, deleting, reordering, or casting columns when reading Parquet files.

The `MAP` can be keyed by field-id or by column name.

To map by field-id, field IDs are required. To make them available when creating the Parquet using DuckDB, use:

```
COPY (SELECT 42::INTEGER AS i) TO 'integers.parquet' (FIELD_IDS {i: 0});
```

Reading Parquet files:

```
SELECT *
FROM read_parquet('integers.parquet', schema = MAP {
                    0: {name: 'renamed_i', type: 'BIGINT', default_value: NULL},
                    1: {name: 'new_column', type: 'UTINYINT', default_value: 43}
                  });
```

renamed_i int64	new_column uint8
42	43

You can also key the `MAP` by column name. Field IDs are not required:

```
COPY (SELECT 42::INTEGER AS i) TO 'integers.parquet';

SELECT *
FROM read_parquet('integers.parquet', schema = MAP {
                    'i': {name: 'renamed_i', type: 'BIGINT', default_value: NULL},
                    'j': {name: 'new_column', type: 'UTINYINT', default_value: 43}
                  });
```

renamed_i int64	new_column uint8
42	43

> The `schema` parameter cannot be combined with `union_by_name = true`.

The `file_row_group_number` Virtual Column

> New As of DuckDB 2.0, the `file_row_group_number` virtual column exposes the Parquet row group that each row belongs to.

The `file_row_group_number` column returns the file-relative index of the row group that each row was read from, starting at `0` and restarting for every file. It enables row-group-level pruning and inspection:

```
SELECT file_row_group_number, count(*)
FROM read_parquet('test.parquet')
GROUP BY file_row_group_number
ORDER BY file_row_group_number;
```

This complements the `file_row_number` column, which returns the file-relative row index.

Partial Reading

DuckDB supports projection pushdown into the Parquet file itself. That is to say, when querying a Parquet file, only the columns required for the query are read. This allows you to read only the part of the Parquet file that you are interested in. This will be done automatically by DuckDB.

DuckDB also supports filter pushdown into the Parquet reader. When you apply a filter to a column that is scanned from a Parquet file, the filter will be pushed down into the scan, and can even be used to skip parts of the file using the built-in zonemaps. Note that this will depend on whether or not your Parquet file contains zonemaps.

Filter and projection pushdown provide significant performance benefits. See our blog post "Querying Parquet with Precision Using DuckDB" for more information.

Inserts and Views

You can also insert the data into a table or create a table from the Parquet file directly. This will load the data from the Parquet file and insert it into the database:

Insert the data from the Parquet file in the table:

```
INSERT INTO people
    SELECT * FROM read_parquet('test.parquet');
```

Create a table directly from a Parquet file:

```
CREATE TABLE people AS
    SELECT * FROM read_parquet('test.parquet');
```

If you wish to keep the data stored inside the Parquet file, but want to query the Parquet file directly, you can create a view over the `read_parquet` function. You can then query the Parquet file as if it were a built-in table:

Create a view over the Parquet file:

```
CREATE VIEW people AS
    SELECT * FROM read_parquet('test.parquet');
```

Query the Parquet file:

```
SELECT * FROM people;
```

Writing to Parquet Files

DuckDB also has support for writing to Parquet files using the COPY statement syntax. See the COPY Statement page for details, including all possible parameters for the COPY statement.

Write a query to a Snappy-compressed Parquet file:

```
COPY
    (SELECT * FROM tbl)
    TO 'result-snappy.parquet'
    (FORMAT parquet);
```

Write `tbl` to a zstd-compressed Parquet file:

```
COPY tbl
    TO 'result-zstd.parquet'
    (FORMAT parquet, COMPRESSION zstd);
```

Write `tbl` to a zstd-compressed Parquet file with the lowest compression level yielding the fastest compression:

```
COPY tbl
    TO 'result-zstd.parquet'
    (FORMAT parquet, COMPRESSION zstd, COMPRESSION_LEVEL 1);
```

Write to Parquet file with key-value metadata:

```
COPY (
    SELECT
        42 AS number,
        true AS is_even
) TO 'kv_metadata.parquet' (
    FORMAT parquet,
    KV_METADATA {
        number: 'Answer to life, universe, and everything',
        is_even: 'not ''odd''' -- single quotes in values must be escaped
    }
);
```

Write to a Parquet v2 file:

```
COPY tbl
    TO 'result-v2.parquet'
    (FORMAT parquet, PARQUET_VERSION 'V2');
```

Write a CSV file to an uncompressed Parquet file:

```
COPY
    'test.csv'
    TO 'result-uncompressed.parquet'
    (FORMAT parquet, COMPRESSION uncompressed);
```

Write a query to a Parquet file with zstd-compression and row group size:

```
COPY
    (FROM generate_series(100_000))
    TO 'row-groups-zstd.parquet'
    (FORMAT parquet, COMPRESSION zstd, ROW_GROUP_SIZE 100_000);
```

Write data to an LZ4-compressed Parquet file:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-lz4.parquet'
    (FORMAT parquet, COMPRESSION lz4);
```

Or, equivalently:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-lz4.parquet'
    (FORMAT parquet, COMPRESSION lz4_raw);
```

Write data to a Brotli-compressed Parquet file:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-brotli.parquet'
    (FORMAT parquet, COMPRESSION brotli);
```

To configure the page size of Parquet file's dictionary pages, use the `STRING_DICTIONARY_PAGE_SIZE_LIMIT` option (default: 1 GiB):

```
COPY
    lineitem
    TO 'lineitem-with-custom-dictionary-size.parquet'
    (FORMAT parquet, STRING_DICTIONARY_PAGE_SIZE_LIMIT 100_000);
```

> New The following feature was introduced in DuckDB 2.0.

To split data pages below the default 100 MiB cap, use `DATA_PAGE_SIZE_LIMIT` (bytes). The limit is best-effort at vector granularity. E.g., to set the limit to 1 MiB, use:

```
COPY tbl
    TO 'tbl.parquet'
    (FORMAT parquet, DATA_PAGE_SIZE_LIMIT 1_048_576);
```

DuckDB's EXPORT command can be used to export an entire database to a series of Parquet files. See the "EXPORT statement" page for more details:

Export the table contents of the entire database as Parquet:

```
EXPORT DATABASE 'target_directory' (FORMAT parquet);
```

Encryption

DuckDB supports reading and writing encrypted Parquet files.

Supported Features

The list of supported Parquet features is available in the Parquet documentation's "Implementation status" page.

Installing and Loading the Parquet Extension

The support for Parquet files is enabled via extension. The `parquet` extension is bundled with almost all clients. However, if your client does not bundle the `parquet` extension, the extension must be installed separately:

```
INSTALL parquet;
```

Querying Parquet Metadata

Parquet Metadata

The `parquet_metadata` function can be used to query the metadata contained within a Parquet file, which reveals various internal details of the Parquet file such as the statistics of the different columns. This can be useful for figuring out what kind of skipping is possible in Parquet files, or even to obtain a quick overview of what the different columns contain. The function supports glob patterns to query metadata across multiple files in parallel:

```
SELECT *
FROM parquet_metadata('test.parquet');
```

```sql
SELECT *
FROM parquet_metadata('data/*.parquet');
```

Below is a table of the columns returned by `parquet_metadata`.

Field	Type
file_name	VARCHAR
row_group_id	BIGINT
row_group_num_rows	BIGINT
row_group_num_columns	BIGINT
row_group_bytes	BIGINT
column_id	BIGINT
file_offset	BIGINT
num_values	BIGINT
path_in_schema	VARCHAR
type	VARCHAR
stats_min	VARCHAR
stats_max	VARCHAR
stats_null_count	BIGINT
stats_distinct_count	BIGINT
stats_min_value	VARCHAR
stats_max_value	VARCHAR
compression	VARCHAR
encodings	VARCHAR
index_page_offset	BIGINT
dictionary_page_offset	BIGINT
data_page_offset	BIGINT
total_compressed_size	BIGINT
total_uncompressed_size	BIGINT
key_value_metadata	MAP(BLOB, BLOB)
bloom_filter_offset	BIGINT
bloom_filter_length	BIGINT
min_is_exact	BOOLEAN
max_is_exact	BOOLEAN
row_group_compressed_bytes	BIGINT

Parquet Schema

The `parquet_schema` function can be used to query the internal schema contained within a Parquet file. Note that this is the schema as it is contained within the metadata of the Parquet file. If you want to figure out the column names and types contained within a Parquet file it is easier to use `DESCRIBE`.

Fetch the column names and column types:

```
DESCRIBE SELECT * FROM 'test.parquet';
```

Fetch the internal schema of a Parquet file:

```
SELECT *
FROM parquet_schema('test.parquet');
```

Below is a table of the columns returned by `parquet_schema`.

Field	Type
file_name	VARCHAR
name	VARCHAR
type	VARCHAR
type_length	VARCHAR
repetition_type	VARCHAR
num_children	BIGINT
converted_type	VARCHAR
scale	BIGINT
precision	BIGINT
field_id	BIGINT
logical_type	VARCHAR

Parquet File Metadata

The `parquet_file_metadata` function can be used to query file-level metadata such as the format version and the encryption algorithm used:

```
SELECT *
FROM parquet_file_metadata('test.parquet');
```

Below is a table of the columns returned by `parquet_file_metadata`.

Field	Type
file_name	VARCHAR
created_by	VARCHAR
num_rows	BIGINT
num_row_groups	BIGINT
format_version	BIGINT
encryption_algorithm	VARCHAR
footer_signing_key_metadata	VARCHAR
file_size_bytes	UBIGINT
footer_size	UBIGINT
column_orders	VARCHAR[]

Parquet Key-Value Metadata

The `parquet_kv_metadata` function can be used to query custom metadata defined as key-value pairs:

```
SELECT *
FROM parquet_kv_metadata('test.parquet');
```

Below is a table of the columns returned by `parquet_kv_metadata`.

Field	Type
file_name	VARCHAR
key	BLOB
value	BLOB

GeoParquet Metadata

GeoParquet files store their geospatial metadata (the primary geometry column, its encoding, the coordinate reference system, and more) under the `geo` key of the Parquet key-value metadata. Because `parquet_kv_metadata` returns the key and value as BLOBs that contain JSON, use the `decode` function to turn them into strings and the JSON extraction operators to read individual fields:

```
SELECT
    decode(value)->>'$.version' AS version,
    decode(value)->>'$.primary_column' AS primary_column,
    decode(value)->>'$.columns.geometry.encoding' AS encoding
FROM parquet_kv_metadata('file.parquet')
WHERE decode(key) = 'geo';
```

version	primary_column	encoding
1.0.0	geometry	WKB

When the file embeds a coordinate reference system, it is stored as PROJJSON under the geometry column's `crs` key, for example `decode(value)->>'$.columns.geometry.crs.id.code'`.

Full Metadata

The `parquet_full_metadata` function returns all metadata for a Parquet file in a single row, combining the results of `parquet_file_metadata`, `parquet_metadata`, `parquet_schema`, and `parquet_kv_metadata` as nested struct arrays:

```
SELECT *
FROM parquet_full_metadata('test.parquet');
```

Field	Type
parquet_file_metadata	STRUCT(...)[]
parquet_metadata	STRUCT(...)[]
parquet_schema	STRUCT(...)[]
parquet_kv_metadata	STRUCT(...)[]

Each struct array contains the same columns as the corresponding standalone function.

Bloom Filters

DuckDB supports Bloom filters for pruning the row groups that need to be read to answer highly selective queries. Currently, Bloom filters are supported for the following types:

- Integer types: `TINYINT`, `UTINYINT`, `SMALLINT`, `USMALLINT`, `INTEGER`, `UINTEGER`, `BIGINT`, `UBIGINT`
- Floating point types: `FLOAT`, `DOUBLE`
- `VARCHAR`
- `BLOB`

The `parquet_bloom_probe(filename, column_name, value)` function shows which row groups can be excluded when filtering for a given value of a given column using the Bloom filter. For example:

```
FROM parquet_bloom_probe('my_file.parquet', 'my_col', 500);
```

file_name	row_group_id	bloom_filter_excludes
my_file.parquet	0	true
…	…	…
my_file.parquet	9	false

Parquet Encryption

Starting with version 0.10.0, DuckDB supports reading and writing encrypted Parquet files. DuckDB broadly follows the Parquet Modular Encryption specification with some limitations.

Reading and Writing Encrypted Files

Using the PRAGMA `add_parquet_key` function, named encryption keys of 128, 192, or 256 bits can be added to a session. These keys are stored in-memory:

```
PRAGMA add_parquet_key('key128', '0123456789112345');
PRAGMA add_parquet_key('key192', '012345678911234501234567');
PRAGMA add_parquet_key('key256', '0123456789112345012345678911234 5');
PRAGMA add_parquet_key('key256base64', 'MDEyMzQ1Njc4OTExMjM0NTAxMjM0NTY3ODkxMTIzNDU=');
```

Writing Encrypted Parquet Files

After specifying the key (e.g., `key256`), files can be encrypted as follows:

```
COPY tbl TO 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});
```

Reading Encrypted Parquet Files

An encrypted Parquet file using a specific key (e.g., `key256`), can then be read as follows:

```
COPY tbl FROM 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});
```

Or:

```
SELECT *
FROM read_parquet('tbl.parquet', encryption_config = {footer_key: 'key256'});
```

Interoperability

DuckDB can read uniformly encrypted Parquet files written by the Arrow C++ API (e.g., via PyArrow), as long as the same encryption key is used for both the footer and all columns.

Limitations

DuckDB's Parquet encryption currently has the following limitations.

DuckDB encrypts the footer and all columns using the `footer_key`. The Parquet specification allows encryption of individual columns with different keys, e.g.:

```
COPY tbl TO 'tbl.parquet'
    (ENCRYPTION_CONFIG {
        footer_key: 'key256',
        column_keys: {key256: ['col0', 'col1']}
    });
```

However, this is unsupported at the moment and will cause an error to be thrown (for now):

```
Not implemented Error: Parquet encryption_config column_keys not yet implemented
```

Performance Implications

Note that encryption has some performance implications. Without encryption, reading/writing the `lineitem` table from TPC-H at SF1, which is 6M rows and 15 columns, from/to a Parquet file takes 0.26 and 0.99 seconds, respectively. With encryption, this takes 0.64 and 2.21 seconds, both approximately 2.5× slower than the unencrypted version.

Parquet Tips

Below is a collection of tips to help when dealing with Parquet files.

Tips for Reading Parquet Files

Use `union_by_name` When Loading Files with Different Schemas

The `union_by_name` option can be used to unify the schema of files that have different or missing columns. For files that do not have certain columns, `NULL` values are filled in:

```
SELECT *
FROM read_parquet('flights*.parquet', union_by_name = true);
```

Tips for Writing Parquet Files

Using a glob pattern upon read or a Hive partitioning structure are good ways to transparently handle multiple files.

Enabling PER_THREAD_OUTPUT

If the final number of Parquet files is not important, writing one file per thread can significantly improve performance:

```
COPY
    (FROM generate_series(10_000_000))
    TO 'test.parquet'
    (FORMAT parquet, PER_THREAD_OUTPUT);
```

Selecting a ROW_GROUP_SIZE

The ROW_GROUP_SIZE parameter specifies the minimum number of rows in a Parquet row group, with a minimum value equal to DuckDB's vector size, 2,048, and a default of 122,880. A Parquet row group is a partition of rows, consisting of a column chunk for each column in the dataset.

Compression algorithms are only applied per row group, so the larger the row group size, the more opportunities to compress the data. On the other hand, larger row group sizes mean that each thread keeps more data in memory before flushing when streaming results. Another argument for smaller row group sizes is that DuckDB can read Parquet row groups in parallel even within the same file and uses predicate pushdown to only scan the row groups whose metadata ranges match the WHERE clause of the query. However, there is some overhead associated with reading the metadata in each group.

A good rule of thumb is to ensure that the number of row groups per file is at least as large as the number of CPU threads used to query that file. More row groups beyond the thread count would improve the speed of highly selective queries, but slow down queries that must scan the whole file like aggregations.

To write a query to a Parquet file with a different row group size, run:

```
COPY
    (FROM generate_series(100_000))
    TO 'row-groups.parquet'
    (FORMAT parquet, ROW_GROUP_SIZE 100_000);
```

The ROW_GROUPS_PER_FILE Option

The ROW_GROUPS_PER_FILE parameter creates a new Parquet file if the current one has a specified number of row groups.

```
COPY
    (FROM generate_series(100_000))
    TO 'output-directory'
    (FORMAT parquet, ROW_GROUP_SIZE 20_000, ROW_GROUPS_PER_FILE 2);
```

> If multiple threads are active, the number of row groups in a file may slightly exceed the specified number of row groups to limit the amount of locking – similarly to the behavior of FILE_SIZE_BYTES. However, if PER_THREAD_OUTPUT is set, only one thread writes to each file, and it becomes accurate again.

Sorting Rows to Improve Row Group Pruning

DuckDB writes per-column statistics, including min/max values, for every row group, and as described above the reader uses these to skip row groups that cannot match a query's WHERE clause. Sorting the data on the columns that you filter on most often before writing makes these min/max ranges tight and non-overlapping between row groups, so that highly selective queries on those columns scan far fewer row groups:

```
COPY
    (FROM 'events.parquet' ORDER BY event_time)
    TO 'events-sorted.parquet'
    (FORMAT parquet);
```

This pairs well with a ROW_GROUP_SIZE that keeps each row group focused on a narrow range of the sort key.

See the Performance Guide on "File Formats" for more tips.

Partitioning

Hive Partitioning

Examples

Read data from a Hive partitioned dataset:

```
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = true);
```

Write a table to a Hive partitioned dataset:

```
COPY orders
TO 'orders' (FORMAT parquet, PARTITION_BY (year, month));
```

Note that the PARTITION_BY options cannot use expressions. You can produce columns on the fly using the following syntax:

```
COPY (SELECT *, year(timestamp) AS year, month(timestamp) AS month FROM services)
TO 'test' (PARTITION_BY (year, month));
```

When reading, the partition columns are read from the directory structure and can be included or excluded depending on the `hive_partitioning` parameter.

```
FROM read_parquet('test/*/*/*.parquet', hive_partitioning = false); -- will not include year, month columns
FROM read_parquet('test/*/*/*.parquet', hive_partitioning = true);  -- will include year, month partition
columns
```

Hive Partitioning

Hive partitioning is a partitioning strategy that is used to split a table into multiple files based on **partition keys**. The files are organized into folders. Within each folder, the **partition key** has a value that is determined by the name of the folder.

Below is an example of a Hive partitioned file hierarchy. The files are partitioned on two keys (`year` and `month`).

```
orders
├── year=2021
│   ├── month=1
│   │   ├── file1.parquet
│   │   └── file2.parquet
│   └── month=2
│       └── file3.parquet
└── year=2022
    ├── month=11
    │   ├── file4.parquet
    │   └── file5.parquet
    └── month=12
        └── file6.parquet
```

Files stored in this hierarchy can be read using the `hive_partitioning` flag.

```
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = true);
```

When we specify the `hive_partitioning` flag, the values of the columns will be read from the directories.

Filter Pushdown

Filters on the partition keys are automatically pushed down into the files. This way the system skips reading files that are not necessary to answer a query. For example, consider the following query on the above dataset:

```
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = true)
WHERE year = 2022
  AND month = 11;
```

When executing this query, only the following files will be read:

```
orders
└── year=2022
    └── month=11
        ├── file4.parquet
        └── file5.parquet
```

Auto-detection

By default the system tries to infer if the provided files are in a hive partitioned hierarchy. And if so, the `hive_partitioning` flag is enabled automatically. The auto-detection will look at the names of the folders and search for a `'key' = 'value'` pattern. This behavior can be overridden by using the `hive_partitioning` option of the reader function:

```
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = false);
```

Hive Types

`hive_types` is a way to specify the logical types of the hive partitions in a struct:

```
SELECT *
FROM read_parquet(
    'dir/**/*.parquet',
    hive_partitioning = true,
    hive_types = {'release': 'DATE', 'orders': 'BIGINT'}
);
```

`hive_types` will be auto-detected for the following types: `DATE`, `TIMESTAMP` and `BIGINT`. To switch off the auto-detection, the flag `hive_types_autocast = 0` can be set.

Writing Partitioned Files

See the Partitioned Writes section.

Partitioned Writes

Examples

Write a table to a Hive partitioned dataset of Parquet files:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month));
```

Write a table to a Hive partitioned dataset of CSV files, allowing overwrites:

```
COPY orders TO 'orders'
(FORMAT csv, PARTITION_BY (year, month), OVERWRITE_OR_IGNORE);
```

Write a table to a Hive partitioned dataset of GZIP-compressed CSV files, setting explicit data files' extension:

```
COPY orders TO 'orders'
(FORMAT csv, PARTITION_BY (year, month), COMPRESSION gzip, FILE_EXTENSION 'csv.gz');
```

Partitioned Writes

When the `PARTITION_BY` clause is specified for the COPY statement, the files are written in a Hive partitioned folder hierarchy. The target is the name of the root directory (in the example above: `orders`). The files are written in-order in the file hierarchy. Currently, one file is written per thread to each directory.

```
orders
├── year=2021
│   ├── month=1
│   │   ├── data_1.parquet
│   │   └── data_2.parquet
│   └── month=2
│       └── data_1.parquet
└── year=2022
    ├── month=11
    │   ├── data_1.parquet
    │   └── data_2.parquet
    └── month=12
        └── data_1.parquet
```

The values of the partitions are automatically extracted from the data. Note that it can be very expensive to write a larger number of partitions as many files will be created. The ideal partition count depends on how large your dataset is.

To limit the maximum number of files the system can keep open before flushing to disk when writing using `PARTITION_BY`, use the `partitioned_write_max_open_files` configuration option (default: 100):

```
SET partitioned_write_max_open_files = 10;
```

> **Best practice.** Writing data into many small partitions is expensive. It is generally recommended to have at least `100 MB` of data per partition.

Filename Pattern

By default, files will be named `data_0.parquet` or `data_0.csv`. With the flag `FILENAME_PATTERN` a pattern with `{i}` or `{uuid}` can be defined to create specific filenames:

- `{i}` will be replaced by an index.
- `{uuid}` will be replaced by a 128 bits long UUID.

Write a table to a Hive partitioned dataset of .parquet files, with an index in the filename:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), OVERWRITE_OR_IGNORE, FILENAME_PATTERN 'orders_{i}');
```

Write a table to a Hive partitioned dataset of .parquet files, with unique filenames:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), OVERWRITE_OR_IGNORE, FILENAME_PATTERN 'file_{uuid}');
```

Custom Partition Paths

> New As of DuckDB 2.0, the PARTITION_PATH option gives full control over the subdirectory that each partition is written to, replacing the default col=value Hive layout.

By default, the PARTITION_BY clause writes each partition to a col=value subdirectory following the Hive partitioning convention. The PARTITION_PATH option replaces this with an expression over the PARTITION_BY columns that computes the output subdirectory for each partition:

```
COPY sales TO 'sales'
(FORMAT parquet, PARTITION_BY (region),
    PARTITION_PATH CASE
        WHEN region = 'eu' THEN 'europe'
        ELSE 'region=' || coalesce(region, 'unknown')
    END);
```

The expression is evaluated once per partition and its result is cast to VARCHAR. The resulting path must be relative (no leading /), non-empty, and must not contain .. segments. PARTITION_PATH requires PARTITION_BY and cannot be combined with HIVE_FILE_PATTERN.

Two helper functions are available to build Hive-style paths inside a PARTITION_PATH expression:

- hive_partition_component(name, value) returns a URL-encoded name=value component. A NULL value produces __HIVE_DEFAULT_PARTITION__:

  ```
  SELECT hive_partition_component('year', 2024);
  ```

  ```
  year=2024
  ```

  ```
  SELECT hive_partition_component('a/b', 'c d');
  ```

  ```
  a%2Fb=c%20d
  ```

- path_join(path, ...) joins its arguments with the platform path separator:

  ```
  SELECT path_join('dir', 'subdir', 'file.txt');
  ```

  ```
  dir/subdir/file.txt
  ```

Sorting Rows Within Partitions

> New As of DuckDB 2.0, the ORDER_BY copy option sorts the rows before they are written. When combined with PARTITION_BY, the rows are sorted within each partition:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), ORDER_BY (amount DESC NULLS LAST));
```

Each expression in the list can optionally be followed by ASC or DESC and NULLS FIRST or NULLS LAST.

Overwriting

By default the partitioned write will not allow overwriting existing directories. On a local file system, the OVERWRITE and OVERWRITE_OR_IGNORE options remove the existing directories. On remote file systems, overwriting is not supported.

Appending

To append to an existing Hive partitioned directory structure, use the APPEND option:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), APPEND);
```

Using the APPEND option results in a behavior similar to the OVERWRITE_OR_IGNORE, FILENAME_PATTERN '{uuid}' options, but DuckDB performs an extra check for whether the file already exists and then regenerates the UUID in the rare event that it does (to avoid clashes).

Handling Slashes in Columns

To handle slashes in column names, use Percent-Encoding implemented by the `url_encode` function.

Appender

The Appender can be used to load bulk data into a DuckDB database. It is available in the C, C++, Go, Java, Julia, Node.js, Python, and Rust clients. This page describes how the Appender behaves across clients and shows a minimal example in each of them. For the full API of a given client, follow the links under Client APIs.

How It Works

An Appender is tied to a single connection and always appends to a single table in the database file, using that connection's transaction context.

Values added to the Appender are cached before being inserted into the database, for performance reasons. That means that, while appending, the rows might not be immediately visible in the system. The cache is flushed automatically when the Appender is closed or goes out of scope, and it can also be flushed manually. Once the Appender has been flushed or closed, all of its data has been written to the database.

> **Note.** In Python, call `flush()` or use a `with` block, which flushes on successful exit. Garbage collection discards any remaining buffered rows. See the Python Appender.

Constraints such as `NOT NULL`, `PRIMARY KEY`, and `UNIQUE` are checked when the cache is flushed rather than as each row is added, so a row can be accepted but later fail at flush time. See Handling Constraint Violations.

Within a row, a value must be appended for every column of the table, in column order, before the row is ended. Ending a row before all columns have been appended, or appending more values than the table has columns, results in an error.

Creating an Appender

Each client creates an Appender from a connection and a target table, adds rows, and then flushes. The examples below insert into a table with an integer and a string column.

C++

The AppendRow function is the easiest way of appending data. It uses recursive templates to allow you to put all the values of a single row within one function call:

```
DuckDB db;
Connection con(db);
con.Query("CREATE TABLE people (id INTEGER, name VARCHAR)");
Appender appender(con, "people");
appender.AppendRow(1, "Mark");
```

Rows can also be individually constructed using the BeginRow, EndRow, and Append methods. This is done internally by AppendRow, and hence has the same performance characteristics:

```
appender.BeginRow();
appender.Append<int32_t>(2);
appender.Append<string>("Hannes");
appender.EndRow();
```

For more information, see the C++ client.

Java (JDBC)

An appender is created from a `DuckDBConnection` with `createAppender()`, and rows are built with `beginRow()`, `append()`, and `endRow()`. Using try-with-resources flushes and closes it at the end of the scope:

```java
try (var appender = conn.createAppender(DuckDBConnection.DEFAULT_SCHEMA, "tbl")) {
    appender.beginRow();
    appender.append(10);
    appender.append("hello");
    appender.endRow();
}
```

For more information, see the Java Appender API.

Go

Obtain an appender by supplying a connection to `NewAppenderFromConn()`, then add rows with `AppendRow()`:

```go
appender, err := NewAppenderFromConn(conn, "", "test")
defer appender.Close()

err = appender.AppendRow(1, "hello")

// Optional, if you want to access the appended rows immediately.
err = appender.Flush()
```

For more information, see the Go Appender API.

Rust

Create an appender from a `Connection` with `appender()`, then push rows built with the `params!` macro:

```rust
let mut app = conn.appender("foo")?;
app.append_row(params![1, "hello"])?;
app.flush()?;
```

For more information, see the Rust Appender API.

Python

Create an appender from a connection with `appender()`, then add rows with `append_row()`. A `with` block flushes on successful exit; if an exception leaves the block, the buffered rows are discarded:

```python
con = duckdb.connect()
con.execute("CREATE TABLE people (id INTEGER, name VARCHAR)")
with con.appender("people") as appender:
    appender.append_row(1, "Mark")
```

For more information, see the Python Appender.

C

For more information, see the C Appender API.

Date, Time and Timestamps

While numbers and strings are rather self-explanatory, dates, times, and timestamps require some explanation. In the C++ API, they can be directly appended using the methods provided by `duckdb::Date`, `duckdb::Time` or `duckdb::Timestamp`. They can also be appended using the internal `duckdb::Value` type, however, this adds some additional overheads and should be avoided if possible.

Below is a short example:

```
con.Query("CREATE TABLE dates (d DATE, t TIME, ts TIMESTAMP)");
Appender appender(con, "dates");

// construct the values using the Date/Time/Timestamp types
// (this is the most efficient approach)
appender.AppendRow(
    Date::FromDate(1992, 1, 1),
    Time::FromTime(1, 1, 1, 0),
    Timestamp::FromDatetime(Date::FromDate(1992, 1, 1), Time::FromTime(1, 1, 1, 0))
);
// construct duckdb::Value objects
appender.AppendRow(
    Value::DATE(1992, 1, 1),
    Value::TIME(1, 1, 1, 0),
    Value::TIMESTAMP(1992, 1, 1, 1, 1, 1, 0)
);
```

Commit Frequency

For clients other than Python, the appender performs commits every 204,800 rows by default. You can change this by explicitly using transactions and surrounding your batches of appended rows by `BEGIN TRANSACTION` and `COMMIT` statements.

Handling Constraint Violations

If the Appender encounters a `PRIMARY KEY` conflict or a `UNIQUE` constraint violation, it fails and returns the following error:

```
Constraint Error:
PRIMARY KEY or UNIQUE constraint violated: duplicate key "..."
```

In this case, the entire append operation fails and no rows are inserted.

Client APIs

Each client's documentation covers its full Appender API, including constructors for non-default schemas and catalogs and any client-specific features:

- C
- Go
- Java (JDBC)
- Julia
- Python
- Rust
- Node.js

INSERT Statements

`INSERT` statements are the standard way of loading data into a relational database. When using `INSERT` statements, the values are supplied row-by-row. While simple, there is significant overhead involved in parsing and processing individual `INSERT` statements. This makes lots of individual row-by-row insertions very inefficient for bulk insertion.

> **Best practice.** As a rule-of-thumb, avoid using lots of individual row-by-row `INSERT` statements when inserting more than a few rows (i.e., avoid using `INSERT` statements as part of a loop). When bulk inserting data, try to maximize the amount of data that is inserted per statement.

If you must use `INSERT` statements to load data in a loop, avoid executing the statements in auto-commit mode. After every commit, the database is required to sync the changes made to disk to ensure no data is lost. In auto-commit mode every single statement will be wrapped in a separate transaction, meaning `fsync` will be called for every statement. This is typically unnecessary when bulk loading and will significantly slow down your program.

> **Tip.** If you absolutely must use `INSERT` statements in a loop to load data, wrap them in calls to `BEGIN TRANSACTION` and `COMMIT`.

Syntax

An example of using `INSERT INTO` to load data in a table is as follows:

```
CREATE TABLE people (id INTEGER, name VARCHAR);
INSERT INTO people VALUES (1, 'Mark'), (2, 'Hannes');
```

For a more detailed description together with a syntax diagram, see the page on the `INSERT` statement.

Lakehouse Formats

Lakehouse formats, often referred to as open table formats, are specifications for storing data in object storage while maintaining some guarantees such as ACID transactions or keeping snapshot history. Over time, multiple lakehouse formats have emerged, each one with its own unique approach to managing its metadata (a.k.a. catalog). In this page, we will go over the support that DuckDB offers for some of these formats as well as some workarounds that you can use to still use DuckDB and get close to full interoperability with these formats.

DuckDB Lakehouse Support Matrix

DuckDB supports Iceberg, Delta, Lance and DuckLake as first-class citizens. The following matrix represents what DuckDB natively supports out of the box through core extensions.

	DuckLake	Iceberg	Delta	Lance
Extension	`ducklake`	`iceberg`	`delta`	`lance`
Read	✅	✅	✅	✅
Write	✅	✅	✅	✅
Deletes	✅	✅	❌	✅
Updates	✅	✅	❌	✅
Upserting	✅	✅	❌	✅
Create table	✅	✅	✅	✅
Create table with partitions	✅	✅	✅	❌
Attaching to a catalog	✅	✅	✅*	✅
Rename table	✅	✅	❌	❌
Rename columns	✅	✅	❌	✅
Add/drop columns	✅	✅	❌	✅
Alter column type	✅	✅	❌	✅
Compaction and maintenance	✅	✅	❌	✅
Encryption	✅	❌	❌	❌
Manage table properties	✅	✅	✅**	❌
Time travel	✅	✅	✅	❌
Query table changes	✅	❌	❌	❌

* Through the `unity_catalog` extension.

** Table properties can be set when creating the table but cannot be altered afterwards.

DuckDB aims to build native extensions with minimal dependencies. The `iceberg` extension for example, has no dependencies on third-party Iceberg libraries, which means all data and metadata operations are implemented natively in the DuckDB extension. For the `delta` extension, we use the `delta-kernel-rs` project, which is meant to be a lightweight platform for engines to build delta integrations that are as close to native as possible.

> **Tip. Why do native implementations matter?** Native implementations allow DuckDB to do more performance optimizations such as complex filter pushdowns (with file-level and row-group level pruning) and improve memory management.

Client APIs

Client Overview

DuckDB is primarily an in-process database system, and as such, it offers client APIs ("drivers") for several languages.

Client API	Maintainer	Support tier	Latest version
C	Core team	Primary	{{ latest_version }}
CLI (command line interface)	Core team	Primary	{{ latest_version }}
Java (JDBC)	Core team	Primary	{{ latest_version }}
Go	Core team	Primary	{{ latest_version }}
Node.js (node-neo)	Jeff Ray-makers	Primary	{{ latest_version }}
ODBC	Core team	Primary	{{ latest_version }}
Python	Core team	Primary	{{ latest_version }}
R	Kirill Müller	Primary	{{ latest_version }}
Rust	Core team	Primary	{{ latest_version }}
WebAssembly (Wasm)	Core team	Primary	{{ latest_version }}
ADBC (Arrow)	Core team	Secondary	{{ latest_version }}
C# (.NET)	Giorgi	Secondary	{{ latest_version }}
C++	Core team	Secondary	{{ latest_version }}

The table above lists the DuckDB clients with the primary and secondary support tiers. For a list of tertiary clients, see the "Tertiary Clients" page.

Support Tiers

There are three tiers of support for clients. Primary clients are the first to receive new features and are covered by community support. Secondary clients receive new features but are not covered by community support. Finally, there are no feature or support guarantees for tertiary clients.

> The DuckDB clients listed above are open-source and we welcome community contributions to these libraries. All primary and secondary clients are available under the MIT license. For tertiary clients, please consult the repository for the license.

Compatibility

All DuckDB clients support the same DuckDB SQL syntax and use the same on-disk database format. DuckDB extensions are also portable between clients with some exceptions (see Wasm extensions).

ADBC Client

> Installation To use the DuckDB ADBC client, download the `libduckdb` archive for your platform and follow the instructions below. For Python, install the `duckdb` package instead, as described in the Python section.
>
> The latest stable version of the DuckDB ADBC client is 2.0.0.

Arrow Database Connectivity (ADBC), similarly to ODBC and JDBC, is a C-style API that enables code portability between different database systems. This allows developers to effortlessly build applications that communicate with database systems without using code specific to that system. The main difference between ADBC and ODBC/JDBC is that ADBC uses Arrow to transfer data between the database system and the application. DuckDB has an ADBC driver, which takes advantage of the zero-copy integration between DuckDB and Arrow to efficiently transfer data.

Please refer to the ADBC documentation page for a more extensive discussion on ADBC and a detailed API explanation.

Implemented Functionality

The DuckDB-ADBC driver implements the full ADBC specification, with the exception of the `ConnectionReadPartition`, `StatementExecutePartitions` and `StatementSetSubstraitPlan` functions. The first two exist to support systems that internally partition the query results, which does not apply to DuckDB. Substrait plans are not supported. In this section, we will describe the main functions that exist in ADBC, along with the arguments they take and provide examples for each function.

Database

Set of functions that operate on a database.

Function name	Description	Arguments	Example
`DatabaseNew`	Allocate a new (but uninitialized) database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseNew(&adbc_database, &adbc_error)`
`DatabaseSetOption`	Set a char* option.	`(AdbcDatabase *database, const char *key, const char *value, AdbcError *error)`	`AdbcDatabaseSetOption(&adbc_database, "path", "test.db", &adbc_error)`
`DatabaseInit`	Finish setting options and initialize the database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseInit(&adbc_database, &adbc_error)`
`DatabaseRelease`	Destroy the database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseRelease(&adbc_database, &adbc_error)`

Database Options

Option	Description
`driver`	Path to the DuckDB shared library (`libduckdb.so`, `libduckdb.dylib`, or `duckdb.dll`).
`entrypoint`	Entry point function name. Must be `duckdb_adbc_init`.
`path`	Path to a DuckDB database file. If not set, an in-memory database is created.
`uri`	Alternative to `path`. Accepts plain paths or `file:` URIs (e.g., `file:test.db`, `file:///absolute/path.db`). Takes precedence over `path` if both are set.

Connection

A set of functions that create and destroy a connection to interact with a database.

Function name	Description	Arguments	Example
`ConnectionNew`	Allocate a new (but uninitialized) connection.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionNew(&adbc_connection, &adbc_error)`
`ConnectionSetOption`	Options may be set before ConnectionInit.	`(AdbcConnection*, const char*, const char*, AdbcError*)`	`AdbcConnectionSetOption(&adbc_connection, ADBC_CONNECTION_OPTION_AUTOCOMMIT, ADBC_OPTION_VALUE_DISABLED, &adbc_error)`
`ConnectionInit`	Finish setting options and initialize the connection.	`(AdbcConnection*, AdbcDatabase*, AdbcError*)`	`AdbcConnectionInit(&adbc_connection, &adbc_database, &adbc_error)`
`ConnectionRelease`	Destroy this connection.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionRelease(&adbc_connection, &adbc_error)`

A set of functions that retrieve metadata about the database. In general, these functions will return Arrow objects, specifically an ArrowArrayStream.

Function name	Description	Arguments	Example
`ConnectionGetInfo`	Get metadata about the driver and database.	`(AdbcConnection*, const uint32_t*, size_t, ArrowArrayStream*, AdbcError*)`	`AdbcConnectionGetInfo(&adbc_connection, NULL, 0, &arrow_stream, &adbc_error)`
`ConnectionGetObjects`	Get a hierarchical view of all catalogs, database schemas, tables and columns.	`(AdbcConnection*, int, const char*, const char*, const char*, const char**, const char*, ArrowArrayStream*, AdbcError*)`	`AdbcConnectionGetObjects(&adbc_connection, ADBC_OBJECT_DEPTH_ALL, NULL, NULL, NULL, NULL, NULL, &arrow_stream, &adbc_error)`

Function name	Description	Arguments	Example
`ConnectionGetTableSchema`	Get the Arrow schema of a table.	`(AdbcConnection*, const char*, const char*, const char*, ArrowSchema*, AdbcError*)`	`AdbcConnectionGetTableSchema(&adbc_connection, NULL, NULL, "my_table", &arrow_schema, &adbc_error)`
`ConnectionGetTableTypes`	Get a list of table types in the database.	`(AdbcConnection*, ArrowArrayStream*, AdbcError*)`	`AdbcConnectionGetTableTypes(&adbc_connection, &arrow_stream, &adbc_error)`

The `ConnectionGetInfo` function supports the following info codes:

Info code	Constant	Description
0	`ADBC_INFO_VENDOR_NAME`	Database vendor name (duckdb).
1	`ADBC_INFO_VENDOR_VERSION`	Database version.
100	`ADBC_INFO_DRIVER_NAME`	Driver name.
101	`ADBC_INFO_DRIVER_VERSION`	Driver version.
102	`ADBC_INFO_DRIVER_ARROW_VERSION`	Arrow library version.
103	`ADBC_INFO_DRIVER_ADBC_VERSION`	ADBC specification version supported by the driver (as an integer, e.g., `1001000` for 1.1.0).

A set of functions with transaction semantics for the connection. By default, all connections start with auto-commit mode on, but this can be turned off via the ConnectionSetOption function.

Function name	Description	Arguments	Example
`ConnectionCommit`	Commit any pending transactions.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionCommit(&adbc_connection, &adbc_error)`
`ConnectionRollback`	Rollback any pending transactions.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionRollback(&adbc_connection, &adbc_error)`

Statement

Statements hold state related to query execution. They represent both one-off queries and prepared statements. They can be reused; however, doing so will invalidate prior result sets from that statement.

The functions used to create, destroy and set options for a statement:

Function name	Description	Arguments	Example
`StatementNew`	Create a new statement for a given connection.	`(AdbcConnection*, AdbcStatement*, AdbcError*)`	`AdbcStatementNew(&adbc_connection, &adbc_statement, &adbc_error)`
`StatementRelease`	Destroy a statement.	`(AdbcStatement*, AdbcError*)`	`AdbcStatementRelease(&adbc_statement, &adbc_error)`
`StatementSetOption`	Set a string option on a statement.	`(AdbcStatement*, const char*, const char*, AdbcError*)`	`StatementSetOption(&adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE, "TABLE_NAME", &adbc_error)`

Functions related to query execution:

Function name	Description	Arguments	Example
`StatementSetSqlQuery`	Set the SQL query to execute. The query can then be executed with StatementExecuteQuery.	`(AdbcStatement*, const char*, AdbcError*)`	`AdbcStatementSetSqlQuery(&adbc_statement, "SELECT * FROM TABLE", &adbc_error)`
`StatementExecuteQuery`	Execute a statement and get the results.	`(AdbcStatement*, ArrowArrayStream*, int64_t*, AdbcError*)`	`AdbcStatementExecuteQuery(&adbc_statement, &arrow_stream, &rows_affected, &adbc_error)`
`StatementPrepare`	Turn this statement into a prepared statement to be executed multiple times.	`(AdbcStatement*, AdbcError*)`	`AdbcStatementPrepare(&adbc_statement, &adbc_error)`

Functions related to binding, used for bulk insertion or in prepared statements.

Function name	Description	Arguments	Example
`StatementBindStream`	Bind Arrow Stream. This can be used for bulk inserts or prepared statements.	`(AdbcStatement*, ArrowArrayStream*, AdbcError*)`	`StatementBindStream(&adbc_statement, &input_data, &adbc_error)`

Ingestion Modes When ingesting data via `StatementBindStream`, the ingestion mode can be set using `ADBC_INGEST_OPTION_MODE`:

Mode	Constant	Description
Create	`ADBC_INGEST_OPTION_MODE_CREATE`	Create a new table (error if the table already exists). This is the default.
Append	`ADBC_INGEST_OPTION_MODE_APPEND`	Append to an existing table (error if the table does not exist).
Replace	`ADBC_INGEST_OPTION_MODE_REPLACE`	Drop the existing table and create a new one.
Create or Append	`ADBC_INGEST_OPTION_MODE_CREATE_APPEND`	Create a new table if it does not exist, otherwise append to it.

`StatementExecuteQuery` returns the number of rows affected through its `rows_affected` output parameter.

Ingestion Options

Option	Description
`adbc.ingest.target_table`	The target table name for ingestion.
`adbc.ingest.target_catalog`	The target catalog (attached database) for ingestion. When set without `adbc.ingest.target_db_schema`, the schema defaults to `main`.
`adbc.ingest.target_db_schema`	The target schema for ingestion.
`adbc.ingest.temporary`	Set to `enabled` to create a temporary table. Incompatible with `target_catalog` and `target_db_schema`.

Setting Up the DuckDB ADBC Driver

Before using DuckDB as an ADBC driver, you must install the `libduckdb` shared library on your system and make it available to your application. This library contains the core DuckDB engine that the ADBC driver interfaces with.

Downloading libduckdb

Download the appropriate `libduckdb` library for your platform from the DuckDB releases page:

- **Linux**: `duckdb-shared-libs-linux-amd64.tar.gz` or `duckdb-shared-libs-linux-arm64.tar.gz` (contains `libduckdb.so`)
- **macOS**: `duckdb-shared-libs-osx-universal.tar.gz` (contains `libduckdb.dylib`)
- **Windows**: `duckdb-shared-libs-windows-amd64.tar.gz` or `duckdb-shared-libs-windows-arm64.tar.gz` (contains `duckdb.dll`)

Extract the archive to obtain the shared library file.

Installing the Library

Linux

1. Extract the `libduckdb.so` file from the downloaded archive
2. Make sure your code can use the library. You can:
    - Either copy it to a system library directory (requires root access):

```
sudo cp libduckdb.so /usr/local/lib/
sudo ldconfig
```

- Or place it in a custom directory and add that directory to your `LD_LIBRARY_PATH`:

```
mkdir -p ~/lib
cp libduckdb.so ~/lib/
export LD_LIBRARY_PATH=~/lib:$LD_LIBRARY_PATH
```

macOS

1. Extract the `libduckdb.dylib` file from the downloaded archive
2. Make sure your code can use the library. You can:

   - Either copy it to a system library directory:

   ```
   sudo cp libduckdb.dylib /usr/local/lib/
   ```

   - Or place it in a custom directory and add that directory to your `DYLD_LIBRARY_PATH`:

   ```
   mkdir -p ~/lib
   cp libduckdb.dylib ~/lib/
   export DYLD_LIBRARY_PATH=~/lib:$DYLD_LIBRARY_PATH
   ```

Windows

1. Extract the `duckdb.dll` file from the downloaded archive
2. Place it in one of the following locations:

   - The same directory as your application executable
   - A directory listed in your `PATH` environment variable
   - The Windows system directory (e.g., `C:\Windows\System32`)

Understanding Library Paths

The `LD_LIBRARY_PATH` (Linux) and `DYLD_LIBRARY_PATH` (macOS) are environment variables that tell the system where to look for shared libraries at runtime. When your application tries to load `libduckdb`, the system searches these paths to locate the library file.

Verifying Installation

You can verify that the library is properly installed and accessible:

Linux/macOS:

```
ldd path/to/your/application  # Linux
otool -L path/to/your/application  # macOS
```

Examples

Regardless of the programming language being used, there are two database options which will be required to utilize ADBC with DuckDB. The first one is the `driver`, which takes a path to the DuckDB library (see Setting Up the DuckDB ADBC Driver above for installation instructions). The second option is the `entrypoint`, which is an exported function from the DuckDB-ADBC driver that initializes all the ADBC functions. Once we have configured these two options, we can optionally set the `path` option, providing a path on disk to store our DuckDB database. If not set, an in-memory database is created. After configuring all the necessary options, we can proceed to initialize our database. Below is how you can do so with various different language environments.

C++

We begin our C++ example by declaring the essential variables for querying data through ADBC. These variables include Error, Database, Connection, Statement handling and an Arrow Stream to transfer data between DuckDB and the application.

```
AdbcError adbc_error;
AdbcDatabase adbc_database;
AdbcConnection adbc_connection;
AdbcStatement adbc_statement;
ArrowArrayStream arrow_stream;
```

We can then initialize our database variable. Before initializing the database, we need to set the `driver` and `entrypoint` options as mentioned above. Then we set the `path` option and initialize the database. The `driver` option should point to your installed `libduckdb` library – see Setting Up the DuckDB ADBC Driver for installation instructions.

```
AdbcDatabaseNew(&adbc_database, &adbc_error);
AdbcDatabaseSetOption(&adbc_database, "driver", "path/to/libduckdb.dylib", &adbc_error);
AdbcDatabaseSetOption(&adbc_database, "entrypoint", "duckdb_adbc_init", &adbc_error);
// By default, we start an in-memory database, but you can optionally define a path to store it on disk.
AdbcDatabaseSetOption(&adbc_database, "path", "test.db", &adbc_error);
AdbcDatabaseInit(&adbc_database, &adbc_error);
```

After initializing the database, we must create and initialize a connection to it.

```
AdbcConnectionNew(&adbc_connection, &adbc_error);
AdbcConnectionInit(&adbc_connection, &adbc_database, &adbc_error);
```

We can now initialize our statement and run queries through our connection. After the `AdbcStatementExecuteQuery` the `arrow_stream` is populated with the result.

```
AdbcStatementNew(&adbc_connection, &adbc_statement, &adbc_error);
AdbcStatementSetSqlQuery(&adbc_statement, "SELECT 42", &adbc_error);
int64_t rows_affected;
AdbcStatementExecuteQuery(&adbc_statement, &arrow_stream, &rows_affected, &adbc_error);
arrow_stream.release(arrow_stream)
```

Besides running queries, we can also ingest data via `arrow_streams`. For this we need to set an option with the table name we want to insert to, bind the stream and then execute the query.

```
StatementSetOption(&adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE, "AnswerToEverything", &adbc_error);
StatementBindStream(&adbc_statement, &arrow_stream, &adbc_error);
StatementExecuteQuery(&adbc_statement, nullptr, nullptr, &adbc_error);
```

Python

The `duckdb` package contains the DuckDB ADBC driver as its `adbc_driver_duckdb` module, so Python needs no `libduckdb` archive. Install it together with the ADBC driver manager and `pyarrow`:

```
pip install duckdb adbc_driver_manager pyarrow
```

`pyarrow` to fetch rows (`fetchone` and `fetchall`). It is not needed for `fetch_arrow()`, which returns an object of the Arrow PyCapsule interface.

> For details on the `adbc_driver_manager` package, see the `adbc_driver_manager` package documentation.

`adbc_driver_duckdb.dbapi.connect` sets the `driver` and `entrypoint` options itself, using the driver in the installed `duckdb` package. It takes the path of the database file as its first argument, and opens an in-memory database without one. There is no need to import `duckdb` first.

```
import adbc_driver_duckdb.dbapi

with adbc_driver_duckdb.dbapi.connect("test.db") as conn, conn.cursor() as cur:
    cur.execute("SELECT 42")
```

```
    # fetch a pyarrow table
    tbl = cur.fetch_arrow_table()
    print(tbl)
```

Alongside `fetch_arrow_table`, the cursor implements the standard DB-API methods, such as `fetchone` and `fetchall`. To load Arrow data into a table, pass the table name and the data, such as a `pyarrow` record batch or table, to `cur.adbc_ingest`. By default it creates the table and fails if the table exists; its `mode` argument also takes `"append"`, `"create_append"` and `"replace"`, as in the ingestion modes above.

```
import adbc_driver_duckdb.dbapi
import pyarrow

data = pyarrow.record_batch(
    [[1, 2, 3, 4], ["a", "b", "c", "d"]],
    names = ["ints", "strs"],
)

with adbc_driver_duckdb.dbapi.connect("test.db") as conn, conn.cursor() as cur:
    cur.adbc_ingest("AnswerToEverything", data)
    conn.commit()
```

The connection does not commit automatically, so without `conn.commit()` the ingested data is lost when the connection closes.

Go

Make sure to install the `libduckdb` library first – see Setting Up the DuckDB ADBC Driver for detailed installation instructions.

The following example uses an in-memory DuckDB database to modify in-memory Arrow RecordBatches via SQL queries:

```
package main

import (
    "bytes"
    "context"
    "fmt"
    "io"

    "github.com/apache/arrow-adbc/go/adbc"
    "github.com/apache/arrow-adbc/go/adbc/drivermgr"
    "github.com/apache/arrow-go/v18/arrow"
    "github.com/apache/arrow-go/v18/arrow/array"
    "github.com/apache/arrow-go/v18/arrow/ipc"
    "github.com/apache/arrow-go/v18/arrow/memory"
)

func _makeSampleArrowRecord() arrow.Record {
    b := array.NewFloat64Builder(memory.DefaultAllocator)
    b.AppendValues([]float64{1, 2, 3}, nil)
    col := b.NewArray()

    defer col.Release()
    defer b.Release()

    schema := arrow.NewSchema([]arrow.Field{{Name: "column1", Type: arrow.PrimitiveTypes.Float64}}, nil)
    return array.NewRecord(schema, []arrow.Array{col}, int64(col.Len()))
}

type DuckDBSQLRunner struct {
    ctx  context.Context
    conn adbc.Connection
    db   adbc.Database
}

func NewDuckDBSQLRunner(ctx context.Context) (*DuckDBSQLRunner, error) {
```

```
    var drv drivermgr.Driver
    db, err := drv.NewDatabase(map[string]string{
        "driver":     "duckdb",
        "entrypoint": "duckdb_adbc_init",
        "path":       ":memory:",
    })
    if err != nil {
        return nil, fmt.Errorf("failed to create new in-memory DuckDB database: %w", err)
    }
    conn, err := db.Open(ctx)
    if err != nil {
        return nil, fmt.Errorf("failed to open connection to new in-memory DuckDB database: %w", err)
    }
    return &DuckDBSQLRunner{ctx: ctx, conn: conn, db: db}, nil
}

func serializeRecord(record arrow.Record) (io.Reader, error) {
    buf := new(bytes.Buffer)
    wr := ipc.NewWriter(buf, ipc.WithSchema(record.Schema()))
    if err := wr.Write(record); err != nil {
        return nil, fmt.Errorf("failed to write record: %w", err)
    }
    if err := wr.Close(); err != nil {
        return nil, fmt.Errorf("failed to close writer: %w", err)
    }
    return buf, nil
}

func (r *DuckDBSQLRunner) importRecord(sr io.Reader) error {
    rdr, err := ipc.NewReader(sr)
    if err != nil {
        return fmt.Errorf("failed to create IPC reader: %w", err)
    }
    defer rdr.Release()

    _, err = adbc.IngestStream(r.ctx, r.conn, rdr, "temp_table", adbc.OptionValueIngestModeCreate,
adbc.IngestStreamOptions{})

    return err
}

func (r *DuckDBSQLRunner) runSQL(sql string) ([]arrow.Record, error) {
    stmt, err := r.conn.NewStatement()
    if err != nil {
        return nil, fmt.Errorf("failed to create new statement: %w", err)
    }
    defer stmt.Close()

    if err := stmt.SetSqlQuery(sql); err != nil {
        return nil, fmt.Errorf("failed to set SQL query: %w", err)
    }
    out, n, err := stmt.ExecuteQuery(r.ctx)
    if err != nil {
        return nil, fmt.Errorf("failed to execute query: %w", err)
    }
    defer out.Release()

    result := make([]arrow.Record, 0, n)
    for out.Next() {
        rec := out.Record()
        rec.Retain() // .Next() will release the record, so we need to retain it
        result = append(result, rec)
    }
    if out.Err() != nil {
        return nil, out.Err()
```

```
    }
    return result, nil
}

func (r *DuckDBSQLRunner) RunSQLOnRecord(record arrow.Record, sql string) ([]arrow.Record, error) {
    serializedRecord, err := serializeRecord(record)
    if err != nil {
        return nil, fmt.Errorf("failed to serialize record: %w", err)
    }
    if err := r.importRecord(serializedRecord); err != nil {
        return nil, fmt.Errorf("failed to import record: %w", err)
    }
    result, err := r.runSQL(sql)
    if err != nil {
        return nil, fmt.Errorf("failed to run SQL: %w", err)
    }

    if _, err := r.runSQL("DROP TABLE temp_table"); err != nil {
        return nil, fmt.Errorf("failed to drop temp table after running query: %w", err)
    }
    return result, nil
}

func (r *DuckDBSQLRunner) Close() {
    r.conn.Close()
    r.db.Close()
}

func main() {
    rec := _makeSampleArrowRecord()
    fmt.Println(rec)

    runner, err := NewDuckDBSQLRunner(context.Background())
    if err != nil {
        panic(err)
    }
    defer runner.Close()

    resultRecords, err := runner.RunSQLOnRecord(rec, "SELECT column1+1 FROM temp_table")
    if err != nil {
        panic(err)
    }

    for _, resultRecord := range resultRecords {
        fmt.Println(resultRecord)
        resultRecord.Release()
    }
}
```

Running it produces the following output:

```
record:
  schema:
  fields: 1
    - column1: type=float64
  rows: 3
  col[0][column1]: [1 2 3]

record:
  schema:
  fields: 1
    - (column1 + 1): type=float64, nullable
  rows: 3
  col[0][(column1 + 1)]: [2 3 4]
```

C

Overview

> Installation To use the DuckDB C API, download the `libduckdb` archive for your platform.
>
> The latest stable version of the DuckDB C API is 2.0.0.

DuckDB implements a custom C API modeled somewhat following the SQLite C API. The API is contained in the `duckdb.h` header. Continue to Startup & Shutdown to get started, or check out the Full API overview.

C API v2

> **Warning.** The C API v2 is not yet stable. Its interface may change in upcoming releases.

DuckDB 2.0 ships a revamped C API, which is considerably more powerful than its predecessor. Among others, it exposes components such as file systems and table functions to extensions.

The C API v2 is declared in the `duckdb_v2.h` header. Extensions that target it include `duckdb_extension_v2.h` instead of `duckdb_extension.h`; an extension targets either the v1 or the v2 C API, never both.

Before marking the C API v2 as stable, we would like to test it in the field. Please give it a try and send us your feedback!

Installation

The DuckDB C API can be installed as part of the `libduckdb` packages. Please see the installation page for details.

Startup & Shutdown

To use DuckDB, you must first initialize a `duckdb_database` handle using `duckdb_open()`. `duckdb_open()` takes as parameter the database file to read and write from. The special value `NULL` (`nullptr`) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process).

With the `duckdb_database` handle, you can create one or many `duckdb_connection` using `duckdb_connect()`. While individual connections are thread-safe, they will be locked during querying. It is therefore recommended that each thread uses its own connection to allow for the best parallel performance.

All `duckdb_connections` have to explicitly be disconnected with `duckdb_disconnect()` and the `duckdb_database` has to be explicitly closed with `duckdb_close()` to avoid memory and file handle leaking.

Example

```
duckdb_database db;
duckdb_connection con;

if (duckdb_open(NULL, &db) == DuckDBError) {
    // handle error
}
if (duckdb_connect(db, &con) == DuckDBError) {
```

```
    // handle error
}

// run queries...

// cleanup
duckdb_disconnect(&con);
duckdb_close(&db);
```

API Reference Overview

```
            duckdb_instance_cache duckdb_create_instance_cache();
duckdb_state duckdb_get_or_create_from_cache(duckdb_instance_cache instance_cache, const char *path,
duckdb_database *out_database, duckdb_config config, char **out_error);
void duckdb_destroy_instance_cache(duckdb_instance_cache *instance_cache);
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
duckdb_state duckdb_open_ext(const char *path, duckdb_database *out_database, duckdb_config config, char
**out_error);
void duckdb_close(duckdb_database *database);
duckdb_state duckdb_connect(duckdb_database database, duckdb_connection *out_connection);
void duckdb_interrupt(duckdb_connection connection);
duckdb_query_progress_type duckdb_query_progress(duckdb_connection connection);
void duckdb_disconnect(duckdb_connection *connection);
void duckdb_connection_get_client_context(duckdb_connection connection, duckdb_client_context *out_
context);
void duckdb_connection_get_arrow_options(duckdb_connection connection, duckdb_arrow_options *out_arrow_
options);
idx_t duckdb_client_context_get_connection_id(duckdb_client_context context);
void duckdb_destroy_client_context(duckdb_client_context *context);
void duckdb_destroy_arrow_options(duckdb_arrow_options *arrow_options);
const char *duckdb_library_version();
duckdb_value duckdb_get_table_names(duckdb_connection connection, const char *query, bool qualified);
```

`duckdb_create_instance_cache` Creates a new database instance cache. The instance cache is necessary if a client/program (re)opens multiple databases to the same file within the same process. Must be destroyed with 'duckdb_destroy_instance_cache'.

Return Value The database instance cache.

Syntax

```
            duckdb_instance_cache duckdb_create_instance_cache(

);
```

`duckdb_get_or_create_from_cache` Creates a new database instance in the instance cache, or retrieves an existing database instance. Must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_get_or_create_from_cache(
  duckdb_instance_cache instance_cache,
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `instance_cache`: The instance cache in which to create the database, or from which to take the database.
- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open or retrieve an in-memory database.
- `out_database`: The resulting cached database.
- `config`: (Optional) configuration used to create the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_instance_cache` Destroys an existing database instance cache and de-allocates its memory.

Syntax

```
           void duckdb_destroy_instance_cache(
  duckdb_instance_cache *instance_cache
);
```

Parameters

- `instance_cache`: The instance cache to destroy.

`duckdb_open` Creates a new database or opens an existing database file stored at the given path. If no path is given a new in-memory database is created instead. The database must be closed with 'duckdb_close'.

Syntax

```
           duckdb_state duckdb_open(
  const char *path,
  duckdb_database *out_database
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_open_ext` Extended version of duckdb_open. Creates a new database or opens an existing database file stored at the given path. The database must be closed with 'duckdb_close'.

Syntax

```
           duckdb_state duckdb_open_ext(
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.
- `config`: (Optional) configuration used to start up the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_close` Closes the specified database and de-allocates all memory allocated for that database. This should be called after you are done with any database allocated through `duckdb_open` or `duckdb_open_ext`. Note that failing to call `duckdb_close` (in case of e.g., a program crash) will not cause data corruption. Still, it is recommended to always correctly close a database object after you are done with it.

Syntax

```
            void duckdb_close(
  duckdb_database *database
);
```

Parameters

- `database`: The database object to shut down.

`duckdb_connect` Opens a connection to a database. Connections are required to query the database and store transactional state associated with the connection. The instantiated connection should be closed using 'duckdb_disconnect'.

Syntax

```
            duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```

Parameters

- `database`: The database file to connect to.
- `out_connection`: The result connection object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_interrupt` Interrupt running query

Syntax

```
            void duckdb_interrupt(
  duckdb_connection connection
);
```

Parameters

- `connection`: The connection to interrupt

`duckdb_query_progress` Get progress of the running query

Syntax

```
            duckdb_query_progress_type duckdb_query_progress(
  duckdb_connection connection
);
```

Parameters

- `connection`: The working connection

Return Value -1 if no progress or a percentage of the progress

`duckdb_disconnect` Closes the specified connection and de-allocates all memory allocated for that connection.

Syntax

```
            void duckdb_disconnect(
  duckdb_connection *connection
);
```

Parameters

- `connection`: The connection to close.

`duckdb_connection_get_client_context` Retrieves the client context of the connection.

Syntax

```
            void duckdb_connection_get_client_context(
  duckdb_connection connection,
  duckdb_client_context *out_context
);
```

Parameters

- `connection`: The connection.
- `out_context`: The client context of the connection. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_connection_get_arrow_options` Retrieves the arrow options of the connection.

Syntax

```
            void duckdb_connection_get_arrow_options(
  duckdb_connection connection,
  duckdb_arrow_options *out_arrow_options
);
```

Parameters

- `connection`: The connection.

`duckdb_client_context_get_connection_id` Returns the connection id of the client context.

Syntax

```
            idx_t duckdb_client_context_get_connection_id(
  duckdb_client_context context
);
```

Parameters

- `context`: The client context.

Return Value The connection id of the client context.

`duckdb_destroy_client_context` Destroys the client context and deallocates its memory.

Syntax

```
            void duckdb_destroy_client_context(
  duckdb_client_context *context
);
```

Parameters

- `context`: The client context to destroy.

`duckdb_destroy_arrow_options` Destroys the arrow options and deallocates its memory.

Syntax

```
            void duckdb_destroy_arrow_options(
  duckdb_arrow_options *arrow_options
);
```

Parameters

- `arrow_options`: The arrow options to destroy.

`duckdb_library_version` Returns the version of the linked DuckDB, with a version postfix for dev versions

Usually used for developing C extensions that must return this for a compatibility check.

Syntax

```
            const char *duckdb_library_version(

);
```

`duckdb_get_table_names` Get the list of (fully qualified) table names of the query.

Syntax

```
            duckdb_value duckdb_get_table_names(
  duckdb_connection connection,
  const char *query,
  bool qualified
);
```

Parameters

- `connection`: The connection for which to get the table names.
- `query`: The query for which to get the table names.
- `qualified`: Returns fully qualified table names (catalog.schema.table), if set to true, else only the (not escaped) table names.

Return Value A duckdb_value of type VARCHAR[] containing the (fully qualified) table names of the query. Must be destroyed with duckdb_destroy_value.

Configuration

Configuration options can be provided to change different settings of the database system. Note that many of these settings can be changed later on using PRAGMA statements as well. The configuration object should be created, filled with values and passed to `duckdb_open_ext`.

Example

```
duckdb_database db;
duckdb_config config;

// create the configuration object
if (duckdb_create_config(&config) == DuckDBError) {
    // handle error
}
// set some configuration options
duckdb_set_config(config, "access_mode", "READ_WRITE"); // or READ_ONLY
duckdb_set_config(config, "threads", "8");
duckdb_set_config(config, "max_memory", "8GB");
duckdb_set_config(config, "default_order", "DESC");

// open the database using the configuration
if (duckdb_open_ext(NULL, &db, config, NULL) == DuckDBError) {
    // handle error
}
// cleanup the configuration object
duckdb_destroy_config(&config);

// run queries...

// cleanup
duckdb_close(&db);
```

API Reference Overview

```
            duckdb_state duckdb_create_config(duckdb_config *out_config);
size_t duckdb_config_count();
duckdb_state duckdb_get_config_flag(size_t index, const char **out_name, const char **out_description);
duckdb_state duckdb_set_config(duckdb_config config, const char *name, const char *option);
void duckdb_destroy_config(duckdb_config *config);
```

`duckdb_create_config` Initializes an empty configuration object that can be used to provide start-up options for the DuckDB instance through `duckdb_open_ext`. The duckdb_config must be destroyed using 'duckdb_destroy_config'

This will always succeed unless there is a malloc failure.

Note that `duckdb_destroy_config` should always be called on the resulting config, even if the function returns `DuckDBError`.

Syntax

```
            duckdb_state duckdb_create_config(
  duckdb_config *out_config
);
```

Parameters

- `out_config`: The result configuration object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_config_count` This returns the total amount of configuration options available for usage with `duckdb_get_config_flag`.

This should not be called in a loop as it internally loops over all the options.

Return Value The amount of config options available.

Syntax

```
            size_t duckdb_config_count(

);
```

`duckdb_get_config_flag` Obtains a human-readable name and description of a specific configuration option. This can be used to e.g. display configuration options. This will succeed unless `index` is out of range (i.e., `>= duckdb_config_count`).

The result name or description MUST NOT be freed.

Syntax

```
            duckdb_state duckdb_get_config_flag(
  size_t index,
  const char **out_name,
  const char **out_description
);
```

Parameters

- `index`: The index of the configuration option (between 0 and `duckdb_config_count`)
- `out_name`: A name of the configuration flag.
- `out_description`: A description of the configuration flag.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_set_config` Sets the specified option for the specified configuration. The configuration option is indicated by name. To obtain a list of config options, see `duckdb_get_config_flag`.

In the source code, configuration options are defined in `config.cpp`.

This can fail if either the name is invalid, or if the value provided for the option is invalid.

Syntax

```
            duckdb_state duckdb_set_config(
  duckdb_config config,
  const char *name,
  const char *option
);
```

Parameters

- `config`: The configuration object to set the option on.
- `name`: The name of the configuration flag to set.
- `option`: The value to set the configuration flag to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_config` Destroys the specified configuration object and de-allocates all memory allocated for the object.

Syntax

```
            void duckdb_destroy_config(
  duckdb_config *config
);
```

Parameters

- `config`: The configuration object to destroy.

Query

The `duckdb_query` method allows SQL queries to be run in DuckDB from C. This method takes two parameters, a (null-terminated) SQL query string and a `duckdb_result` result pointer. The result pointer may be `NULL` if the application is not interested in the result set or if the query produces no result. After the result is consumed, the `duckdb_destroy_result` method should be used to clean up the result.

Elements can be extracted from the `duckdb_result` object using a variety of methods. The `duckdb_column_count` can be used to extract the number of columns. `duckdb_column_name` and `duckdb_column_type` can be used to extract the names and types of individual columns.

Example

```
duckdb_state state;
duckdb_result result;

// create a table
state = duckdb_query(con, "CREATE TABLE integers (i INTEGER, j INTEGER);", NULL);
if (state == DuckDBError) {
    // handle error
```

```
}
// insert three rows into the table
state = duckdb_query(con, "INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL);", NULL);
if (state == DuckDBError) {
    // handle error
}
// query rows again
state = duckdb_query(con, "SELECT * FROM integers", &result);
if (state == DuckDBError) {
    // handle error
}
// handle the result
// ...

// destroy the result after we are done with it
duckdb_destroy_result(&result);
```

Value Extraction

Values can be extracted using either the `duckdb_fetch_chunk` function, or using the `duckdb_value` convenience functions. The `duckdb_fetch_chunk` function directly hands you data chunks in DuckDB's native array format and can therefore be very fast. The `duckdb_value` functions perform bounds- and type-checking, and will automatically cast values to the desired type. This makes them more convenient and easier to use, at the expense of being slower.

See the Types page for more information.

> For optimal performance, use `duckdb_fetch_chunk` to extract data from the query result. The `duckdb_value` functions perform internal type-checking, bounds-checking and casting which makes them slower.

`duckdb_fetch_chunk`

Below is an end-to-end example that prints the above result to CSV format using the `duckdb_fetch_chunk` function. Note that the function is NOT generic: we do need to know exactly what the types of the result columns are.

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "CREATE TABLE integers (i INTEGER, j INTEGER);", NULL);
duckdb_query(con, "INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL);", NULL);
duckdb_query(con, "SELECT * FROM integers;", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the first column
    duckdb_vector col1 = duckdb_data_chunk_get_vector(result, 0);
    int32_t *col1_data = (int32_t *) duckdb_vector_get_data(col1);
    uint64_t *col1_validity = duckdb_vector_get_validity(col1);

    // get the second column
    duckdb_vector col2 = duckdb_data_chunk_get_vector(result, 1);
    int32_t *col2_data = (int32_t *) duckdb_vector_get_data(col2);
```

```
    uint64_t *col2_validity = duckdb_vector_get_validity(col2);

    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (duckdb_validity_row_is_valid(col1_validity, row)) {
            printf("%d", col1_data[row]);
        } else {
            printf("NULL");
        }
        printf(",");
        if (duckdb_validity_row_is_valid(col2_validity, row)) {
            printf("%d", col2_data[row]);
        } else {
            printf("NULL");
        }
        printf("\n");
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

This prints the following result:

```
3,4
5,6
7,NULL
```

duckdb_value

> **Deprecated.** The `duckdb_value` functions are deprecated and are scheduled for removal in a future release.

Below is an example that prints the above result to CSV format using the `duckdb_value_varchar` function. Note that the function is generic: we do not need to know about the types of the individual result columns.

```
// print the above result to CSV format using `duckdb_value_varchar`
idx_t row_count = duckdb_row_count(&result);
idx_t column_count = duckdb_column_count(&result);
for (idx_t row = 0; row < row_count; row++) {
    for (idx_t col = 0; col < column_count; col++) {
        if (col > 0) printf(",");
        auto str_val = duckdb_value_varchar(&result, col, row);
        printf("%s", str_val);
        duckdb_free(str_val);
   }
   printf("\n");
}
```

API Reference Overview

```
            duckdb_state duckdb_query(duckdb_connection connection, const char *query, duckdb_result *out_
result);
void duckdb_destroy_result(duckdb_result *result);
const char *duckdb_column_name(duckdb_result *result, idx_t col);
duckdb_type duckdb_column_type(duckdb_result *result, idx_t col);
duckdb_statement_type duckdb_result_statement_type(duckdb_result result);
duckdb_logical_type duckdb_column_logical_type(duckdb_result *result, idx_t col);
duckdb_arrow_options duckdb_result_get_arrow_options(duckdb_result *result);
idx_t duckdb_column_count(duckdb_result *result);
idx_t duckdb_row_count(duckdb_result *result);
idx_t duckdb_rows_changed(duckdb_result *result);
```

```
void *duckdb_column_data(duckdb_result *result, idx_t col);
bool *duckdb_nullmask_data(duckdb_result *result, idx_t col);
const char *duckdb_result_error(duckdb_result *result);
duckdb_error_type duckdb_result_error_type(duckdb_result *result);
```

`duckdb_query` Executes a SQL query within a connection and stores the full (materialized) result in the out_result pointer. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_result_error`.

Note that after running `duckdb_query`, `duckdb_destroy_result` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
            duckdb_state duckdb_query(
  duckdb_connection connection,
  const char *query,
  duckdb_result *out_result
);
```

Parameters

- `connection`: The connection to perform the query in.
- `query`: The SQL query to run.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_result` Closes the result and de-allocates all memory allocated for that result.

Syntax

```
            void duckdb_destroy_result(
  duckdb_result *result
);
```

Parameters

- `result`: The result to destroy.

`duckdb_column_name` Returns the column name of the specified column. The result should not need to be freed; the column names will automatically be destroyed when the result is destroyed.

Returns `NULL` if the column is out of range.

Syntax

```
            const char *duckdb_column_name(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column name from.
- `col`: The column index.

Return Value The column name of the specified column.

`duckdb_column_type` Returns the column type of the specified column.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
            duckdb_type duckdb_column_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The column type of the specified column.

`duckdb_result_statement_type` Returns the statement type of the statement that was executed

Syntax

```
            duckdb_statement_type duckdb_result_statement_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the statement type from.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_column_logical_type` Returns the logical column type of the specified column.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Returns `NULL` if the column is out of range.

Syntax

```
            duckdb_logical_type duckdb_column_logical_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The logical column type of the specified column.

`duckdb_result_get_arrow_options` Returns the arrow options associated with the given result. These options are definitions of how the arrow arrays/schema should be produced.

Syntax

```
            duckdb_arrow_options duckdb_result_get_arrow_options(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch arrow options from.

Return Value The arrow options associated with the given result. This must be destroyed with `duckdb_destroy_arrow_options`.

`duckdb_column_count` Returns the number of columns present in the result object.

Syntax

```
            idx_t duckdb_column_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of columns present in the result object.

`duckdb_row_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows present in the result object.

Syntax

```
            idx_t duckdb_row_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows present in the result object.

`duckdb_rows_changed` Returns the number of rows changed by the query stored in the result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
            idx_t duckdb_rows_changed(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows changed.

`duckdb_column_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the data of a specific column of a result in columnar format.

The function returns a dense array which contains the result data. The exact type stored in the array depends on the corresponding duckdb_type (as provided by `duckdb_column_type`). For the exact type by which the data should be accessed, see the comments in the types section or the DUCKDB_TYPE enum.

For example, for a column of type DUCKDB_TYPE_INTEGER, rows can be accessed in the following manner:

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

Syntax

```
            void *duckdb_column_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column data from.
- `col`: The column index.

Return Value The column data of the specified column.

`duckdb_nullmask_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the nullmask of a specific column of a result in columnar format. The nullmask indicates for every row whether or not the corresponding row is NULL. If a row is NULL, the values present in the array provided by `duckdb_column_data` are undefined.

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
bool *nullmask = duckdb_nullmask_data(&result, 0);
if (nullmask[row]) {
    printf("Data for row %d: NULL\n", row);
} else {
    printf("Data for row %d: %d\n", row, data[row]);
}
```

Syntax

```
            bool *duckdb_nullmask_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the nullmask from.
- `col`: The column index.

Return Value The nullmask of the specified column.

`duckdb_result_error` Returns the error message contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_result` is called.

Syntax

```
            const char *duckdb_result_error(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error of the result.

`duckdb_result_error_type` Returns the result error type contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

Syntax

```
            duckdb_error_type duckdb_result_error_type(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error type of the result.

Data Chunks

Data chunks represent a horizontal slice of a table. They hold a number of vectors, that can each hold up to the `VECTOR_SIZE` rows. The vector size can be obtained through the `duckdb_vector_size` function and is configurable, but is usually set to `2048`.

Data chunks and vectors are what DuckDB uses natively to store and represent data. For this reason, the data chunk interface is the most efficient way of interfacing with DuckDB. Be aware, however, that correctly interfacing with DuckDB using the data chunk API does require knowledge of DuckDB's internal vector format.

Data chunks can be used in two manners:

- **Reading Data**: Data chunks can be obtained from query results using the `duckdb_fetch_chunk` method, or as input to a user-defined function. In this case, the vector methods can be used to read individual values.
- **Writing Data**: Data chunks can be created using `duckdb_create_data_chunk`. The data chunk can then be filled with values and used in `duckdb_append_data_chunk` to write data to the database.

The primary manner of interfacing with data chunks is by obtaining the internal vectors of the data chunk using the `duckdb_data_chunk_get_vector` method. Afterwards, the vector methods can be used to read from or write to the individual vectors.

API Reference Overview

```
duckdb_data_chunk duckdb_create_data_chunk(duckdb_logical_type *types, idx_t column_count);
void duckdb_destroy_data_chunk(duckdb_data_chunk *chunk);
void duckdb_data_chunk_reset(duckdb_data_chunk chunk);
idx_t duckdb_data_chunk_get_column_count(duckdb_data_chunk chunk);
duckdb_vector duckdb_data_chunk_get_vector(duckdb_data_chunk chunk, idx_t col_idx);
idx_t duckdb_data_chunk_get_size(duckdb_data_chunk chunk);
void duckdb_data_chunk_set_size(duckdb_data_chunk chunk, idx_t size);
```

`duckdb_create_data_chunk` Creates an empty data chunk with the specified column types. The result must be destroyed with `duckdb_destroy_data_chunk`.

Syntax

```
duckdb_data_chunk duckdb_create_data_chunk(
  duckdb_logical_type *types,
  idx_t column_count
);
```

Parameters

- `types`: An array of column types. Column types cannot contain ANY and INVALID types.
- `column_count`: The number of columns.

Return Value The data chunk.

`duckdb_destroy_data_chunk` Destroys the data chunk and de-allocates all memory allocated for that chunk.

Syntax

```
void duckdb_destroy_data_chunk(
  duckdb_data_chunk *chunk
);
```

Parameters

- chunk: The data chunk to destroy.

`duckdb_data_chunk_reset` Resets a data chunk, clearing the validity masks and setting the cardinality of the data chunk to 0. After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            void duckdb_data_chunk_reset(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to reset.

`duckdb_data_chunk_get_column_count` Retrieves the number of columns in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_column_count(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of columns in the data chunk

`duckdb_data_chunk_get_vector` Retrieves the vector at the specified column index in the data chunk.

The pointer to the vector is valid for as long as the chunk is alive. It does NOT need to be destroyed.

Syntax

```
            duckdb_vector duckdb_data_chunk_get_vector(
  duckdb_data_chunk chunk,
  idx_t col_idx
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The vector

`duckdb_data_chunk_get_size` Retrieves the current number of tuples in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_size(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of tuples in the data chunk

`duckdb_data_chunk_set_size` Sets the current number of tuples in a data chunk.

Syntax

```
            void duckdb_data_chunk_set_size(
  duckdb_data_chunk chunk,
  idx_t size
);
```

Parameters

- chunk: The data chunk to set the size in
- size: The number of tuples in the data chunk

Vectors

Vectors represent a horizontal slice of a column. They hold a number of values of a specific type, similar to an array. Vectors are the core data representation used in DuckDB. Vectors are typically stored within data chunks.

The vector and data chunk interfaces are the most efficient way of interacting with DuckDB, allowing for the highest performance. However, the interfaces are also difficult to use and care must be taken when using them.

Vector Format

Vectors are arrays of a specific data type. The logical type of a vector can be obtained using `duckdb_vector_get_column_type`. The type id of the logical type can then be obtained using `duckdb_get_type_id`.

Vectors themselves do not have sizes. Instead, the parent data chunk has a size (that can be obtained through `duckdb_data_chunk_get_size`). All vectors that belong to a data chunk have the same size.

Primitive Types

For primitive types, the underlying array can be obtained using the `duckdb_vector_get_data` method. The array can then be accessed using the correct native type. Below is a table that contains a mapping of the `duckdb_type` to the native type of the array.

duckdb_type	NativeType
DUCKDB_TYPE_BOOLEAN	bool
DUCKDB_TYPE_TINYINT	int8_t
DUCKDB_TYPE_SMALLINT	int16_t

duckdb_type	NativeType
DUCKDB_TYPE_INTEGER	int32_t
DUCKDB_TYPE_BIGINT	int64_t
DUCKDB_TYPE_UTINYINT	uint8_t
DUCKDB_TYPE_USMALLINT	uint16_t
DUCKDB_TYPE_UINTEGER	uint32_t
DUCKDB_TYPE_UBIGINT	uint64_t
DUCKDB_TYPE_FLOAT	float
DUCKDB_TYPE_DOUBLE	double
DUCKDB_TYPE_TIMESTAMP	duckdb_timestamp
DUCKDB_TYPE_DATE	duckdb_date
DUCKDB_TYPE_TIME	duckdb_time
DUCKDB_TYPE_INTERVAL	duckdb_interval
DUCKDB_TYPE_HUGEINT	duckdb_hugeint
DUCKDB_TYPE_UHUGEINT	duckdb_uhugeint
DUCKDB_TYPE_VARCHAR	duckdb_string_t
DUCKDB_TYPE_BLOB	duckdb_string_t
DUCKDB_TYPE_TIMESTAMP_S	duckdb_timestamp
DUCKDB_TYPE_TIMESTAMP_MS	duckdb_timestamp
DUCKDB_TYPE_TIMESTAMP_NS	duckdb_timestamp
DUCKDB_TYPE_UUID	duckdb_hugeint
DUCKDB_TYPE_TIME_TZ	duckdb_time_tz
DUCKDB_TYPE_TIMESTAMP_TZ	duckdb_timestamp

NULL Values

Any value in a vector can be `NULL`. When a value is `NULL`, the values contained within the primary array at that index is undefined (and can be uninitialized). The validity mask is a bitmask consisting of `uint64_t` elements. For every 64 values in the vector, one `uint64_t` element exists (rounded up). The validity mask has its bit set to 1 if the value is valid, or set to 0 if the value is invalid (i.e., `NULL`).

The bits of the bitmask can be read directly, or the slower helper method `duckdb_validity_row_is_valid` can be used to check whether or not a value is `NULL`.

The `duckdb_vector_get_validity` returns a pointer to the validity mask. Note that if all values in a vector are valid, this function **might** return `nullptr` in which case the validity mask does not need to be checked.

Strings

String values are stored as a `duckdb_string_t`. This is a special struct that stores the string inline (if it is short, i.e., `<= 12 bytes`) or a pointer to the string data if it is longer than `12` bytes.

```
typedef struct {
    union {
        struct {
            uint32_t length;
            char prefix[4];
```

```
            char *ptr;
        } pointer;
        struct {
            uint32_t length;
            char inlined[12];
        } inlined;
    } value;
} duckdb_string_t;
```

The length can either be accessed directly, or the `duckdb_string_is_inlined` can be used to check if a string is inlined.

Decimals

Decimals are stored as integer values internally. The exact native type depends on the `width` of the decimal type, as shown in the following table:

Width	NativeType
<= 4	int16_t
<= 9	int32_t
<= 18	int64_t
<= 38	duckdb_hugeint

The `duckdb_decimal_internal_type` can be used to obtain the internal type of the decimal.

Decimals are stored as integer values multiplied by `10^scale`. The scale of a decimal can be obtained using `duckdb_decimal_scale`. For example, a decimal value of `10.5` with type `DECIMAL(8, 3)` is stored internally as an `int32_t` value of `10500`. In order to obtain the correct decimal value, the value should be divided by the appropriate power-of-ten.

Enums

Enums are stored as unsigned integer values internally. The exact native type depends on the size of the enum dictionary, as shown in the following table:

Dictionary size	NativeType
<= 255	uint8_t
<= 65535	uint16_t
<= 4294967295	uint32_t

The `duckdb_enum_internal_type` can be used to obtain the internal type of the enum.

In order to obtain the actual string value of the enum, the `duckdb_enum_dictionary_value` function must be used to obtain the enum value that corresponds to the given dictionary entry. Note that the enum dictionary is the same for the entire column – and so only needs to be constructed once.

Structs

Structs are nested types that contain any number of child types. Think of them like a `struct` in C. The way to access struct data using vectors is to access the child vectors recursively using the `duckdb_struct_vector_get_child` method.

The struct vector itself does not have any data (i.e., you should not use `duckdb_vector_get_data` method on the struct). **However**, the struct vector itself **does** have a validity mask. The reason for this is that the child elements of a struct can be `NULL`, but the struct **itself** can also be `NULL`.

Lists

Lists are nested types that contain a single child type, repeated x times per row. Think of them like a variable-length array in C. The way to access list data using vectors is to access the child vector using the `duckdb_list_vector_get_child` method.

The `duckdb_vector_get_data` must be used to get the offsets and lengths of the lists stored as `duckdb_list_entry`, that can then be applied to the child vector.

```
typedef struct {
    uint64_t offset;
    uint64_t length;
} duckdb_list_entry;
```

Note that both list entries itself **and** any children stored in the lists can also be `NULL`. This must be checked using the validity mask again.

Arrays

Arrays are nested types that contain a single child type, repeated exactly `array_size` times per row. Think of them like a fixed-size array in C. Arrays work exactly the same as lists, **except** the length and offset of each entry is fixed. The fixed array size can be obtained by using `duckdb_array_type_array_size`. The data for entry n then resides at `offset = n * array_size` and always has `length = array_size`.

Note that much like lists, arrays can still be `NULL`, which must be checked using the validity mask.

Examples

Below are several full end-to-end examples of how to interact with vectors.

Example: Reading an int64 Vector with NULL Values

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "SELECT CASE WHEN i%2=0 THEN NULL ELSE i END res_col FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the first column
    duckdb_vector res_col = duckdb_data_chunk_get_vector(result, 0);
    // get the native array and the validity mask of the vector
    int64_t *vector_data = (int64_t *) duckdb_vector_get_data(res_col);
    uint64_t *vector_validity = duckdb_vector_get_validity(res_col);
    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (duckdb_validity_row_is_valid(vector_validity, row)) {
            printf("%lld\n", vector_data[row]);
        } else {
            printf("NULL\n");
        }
    }
```

```
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

Example: Reading a String Vector

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "SELECT CASE WHEN i%2=0 THEN CONCAT('short_', i) ELSE CONCAT('longstringprefix', i) END
FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the first column
    duckdb_vector res_col = duckdb_data_chunk_get_vector(result, 0);
    // get the native array and the validity mask of the vector
    duckdb_string_t *vector_data = (duckdb_string_t *) duckdb_vector_get_data(res_col);
    uint64_t *vector_validity = duckdb_vector_get_validity(res_col);
    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (duckdb_validity_row_is_valid(vector_validity, row)) {
            duckdb_string_t str = vector_data[row];
            if (duckdb_string_is_inlined(str)) {
                // use inlined string
                printf("%.*s\n", str.value.inlined.length, str.value.inlined.inlined);
            } else {
                // follow string pointer
                printf("%.*s\n", str.value.pointer.length, str.value.pointer.ptr);
            }
        } else {
            printf("NULL\n");
        }
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

Example: Reading a Struct Vector

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
```

```
duckdb_query(con, "SELECT CASE WHEN i%5=0 THEN NULL ELSE {'col1': i, 'col2': CASE WHEN i%2=0 THEN NULL ELSE
100 + i * 42 END} END FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the struct column
    duckdb_vector struct_col = duckdb_data_chunk_get_vector(result, 0);
    uint64_t *struct_validity = duckdb_vector_get_validity(struct_col);
    // get the child columns of the struct
    duckdb_vector col1_vector = duckdb_struct_vector_get_child(struct_col, 0);
    int64_t *col1_data = (int64_t *) duckdb_vector_get_data(col1_vector);
    uint64_t *col1_validity = duckdb_vector_get_validity(col1_vector);

    duckdb_vector col2_vector = duckdb_struct_vector_get_child(struct_col, 1);
    int64_t *col2_data = (int64_t *) duckdb_vector_get_data(col2_vector);
    uint64_t *col2_validity = duckdb_vector_get_validity(col2_vector);

    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (!duckdb_validity_row_is_valid(struct_validity, row)) {
            // entire struct is NULL
            printf("NULL\n");
            continue;
        }
        // read col1
        printf("{'col1': ");
        if (!duckdb_validity_row_is_valid(col1_validity, row)) {
            // col1 is NULL
            printf("NULL");
        } else {
            printf("%lld", col1_data[row]);
        }
        printf(", 'col2': ");
        if (!duckdb_validity_row_is_valid(col2_validity, row)) {
            // col2 is NULL
            printf("NULL");
        } else {
            printf("%lld", col2_data[row]);
        }
        printf("}\n");
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

Example: Reading a List Vector

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
```

```
duckdb_query(con, "SELECT CASE WHEN i % 5 = 0 THEN NULL WHEN i % 2 = 0 THEN [i, i + 1] ELSE [i * 42, NULL, i
* 84] END FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the list column
    duckdb_vector list_col = duckdb_data_chunk_get_vector(result, 0);
    duckdb_list_entry *list_data = (duckdb_list_entry *) duckdb_vector_get_data(list_col);
    uint64_t *list_validity = duckdb_vector_get_validity(list_col);
    // get the child column of the list
    duckdb_vector list_child = duckdb_list_vector_get_child(list_col);
    int64_t *child_data = (int64_t *) duckdb_vector_get_data(list_child);
    uint64_t *child_validity = duckdb_vector_get_validity(list_child);

    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (!duckdb_validity_row_is_valid(list_validity, row)) {
            // entire list is NULL
            printf("NULL\n");
            continue;
        }
        // read the list offsets for this row
        duckdb_list_entry list = list_data[row];
        printf("[");
        for (idx_t child_idx = list.offset; child_idx < list.offset + list.length; child_idx++) {
            if (child_idx > list.offset) {
                printf(", ");
            }
            if (!duckdb_validity_row_is_valid(child_validity, child_idx)) {
                // col1 is NULL
                printf("NULL");
            } else {
                printf("%lld", child_data[child_idx]);
            }
        }
        printf("]\n");
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

API Reference Overview

```
            duckdb_vector duckdb_create_vector(duckdb_logical_type type, idx_t capacity);
void duckdb_destroy_vector(duckdb_vector *vector);
duckdb_logical_type duckdb_vector_get_column_type(duckdb_vector vector);
void *duckdb_vector_get_data(duckdb_vector vector);
uint64_t *duckdb_vector_get_validity(duckdb_vector vector);
void duckdb_vector_ensure_validity_writable(duckdb_vector vector);
void duckdb_vector_assign_string_element(duckdb_vector vector, idx_t index, const char *str);
void duckdb_vector_assign_string_element_len(duckdb_vector vector, idx_t index, const char *str, idx_t str_
len);
duckdb_vector duckdb_list_vector_get_child(duckdb_vector vector);
idx_t duckdb_list_vector_get_size(duckdb_vector vector);
```

```
duckdb_state duckdb_list_vector_set_size(duckdb_vector vector, idx_t size);
duckdb_state duckdb_list_vector_reserve(duckdb_vector vector, idx_t required_capacity);
duckdb_vector duckdb_struct_vector_get_child(duckdb_vector vector, idx_t index);
duckdb_vector duckdb_array_vector_get_child(duckdb_vector vector);
void duckdb_slice_vector(duckdb_vector vector, duckdb_selection_vector sel, idx_t len);
void duckdb_vector_copy_sel(duckdb_vector src, duckdb_vector dst, duckdb_selection_vector sel, idx_t src_
count, idx_t src_offset, idx_t dst_offset);
void duckdb_vector_reference_value(duckdb_vector vector, duckdb_value value);
void duckdb_vector_reference_vector(duckdb_vector to_vector, duckdb_vector from_vector);
```

Validity Mask Functions

```
            bool duckdb_validity_row_is_valid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_validity(uint64_t *validity, idx_t row, bool valid);
void duckdb_validity_set_row_invalid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_valid(uint64_t *validity, idx_t row);
```

`duckdb_create_vector` Creates a flat vector. Must be destroyed with `duckdb_destroy_vector`.

Syntax

```
            duckdb_vector duckdb_create_vector(
  duckdb_logical_type type,
  idx_t capacity
);
```

Parameters

- `type`: The logical type of the vector.
- `capacity`: The capacity of the vector.

Return Value The vector.

`duckdb_destroy_vector` Destroys the vector and de-allocates its memory.

Syntax

```
            void duckdb_destroy_vector(
  duckdb_vector *vector
);
```

Parameters

- `vector`: A pointer to the vector.

`duckdb_vector_get_column_type` Retrieves the column type of the specified vector.

The result must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_vector_get_column_type(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector get the data from

Return Value The type of the vector

`duckdb_vector_get_data` Retrieves the data pointer of the vector.

The data pointer can be used to read or write values from the vector. How to read or write values depends on the type of the vector.

Syntax

```
            void *duckdb_vector_get_data(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The data pointer

`duckdb_vector_get_validity` Retrieves the validity mask pointer of the specified vector.

If all values are valid, this function MIGHT return NULL!

The validity mask is a bitset that signifies null-ness within the data chunk. It is a series of uint64_t values, where each uint64_t value contains validity for 64 tuples. The bit is set to 1 if the value is valid (i.e., not NULL) or 0 if the value is invalid (i.e., NULL).

Validity of a specific value can be obtained like this:

idx_t entry_idx = row_idx / 64; idx_t idx_in_entry = row_idx % 64; bool is_valid = validity_mask[entry_idx] & (1 « idx_in_entry);

Alternatively, the (slower) duckdb_validity_row_is_valid function can be used.

Syntax

```
            uint64_t *duckdb_vector_get_validity(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The pointer to the validity mask, or NULL if no validity mask is present

`duckdb_vector_ensure_validity_writable` Ensures the validity mask is writable by allocating it.

After this function is called, `duckdb_vector_get_validity` will ALWAYS return non-NULL. This allows NULL values to be written to the vector, regardless of whether a validity mask was present before.

Syntax

```
            void duckdb_vector_ensure_validity_writable(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to alter

`duckdb_vector_assign_string_element` Assigns a string element in the vector at the specified location.

Syntax

```
            void duckdb_vector_assign_string_element(
  duckdb_vector vector,
  idx_t index,
  const char *str
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The null-terminated string

`duckdb_vector_assign_string_element_len` Assigns a string element in the vector at the specified location. You may also use this function to assign BLOBs.

Syntax

```
            void duckdb_vector_assign_string_element_len(
  duckdb_vector vector,
  idx_t index,
  const char *str,
  idx_t str_len
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The string
- `str_len`: The length of the string (in bytes)

`duckdb_list_vector_get_child` Retrieves the child vector of a list vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_list_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_list_vector_get_size` Returns the size of the child vector of the list.

Syntax

```
            idx_t duckdb_list_vector_get_size(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The size of the child list

`duckdb_list_vector_set_size` Sets the total size of the underlying child-vector of a list vector.

Syntax

```
            duckdb_state duckdb_list_vector_set_size(
  duckdb_vector vector,
  idx_t size
);
```

Parameters

- `vector`: The list vector.
- `size`: The size of the child list.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_list_vector_reserve` Sets the total capacity of the underlying child-vector of a list.

After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            duckdb_state duckdb_list_vector_reserve(
  duckdb_vector vector,
  idx_t required_capacity
);
```

Parameters

- `vector`: The list vector.
- `required_capacity`: the total capacity to reserve.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_struct_vector_get_child` Retrieves the child vector of a struct vector. The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_struct_vector_get_child(
  duckdb_vector vector,
  idx_t index
);
```

Parameters

- `vector`: The vector
- `index`: The child index

Return Value The child vector

`duckdb_array_vector_get_child` Retrieves the child vector of an array vector. The resulting vector is valid as long as the parent vector is valid. The resulting vector has the size of the parent vector multiplied by the array size.

Syntax

```
            duckdb_vector duckdb_array_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_slice_vector` Slice a vector with a selection vector. The length of the selection vector must be less than or equal to the length of the vector. Turns the vector into a dictionary vector.

Syntax

```
            void duckdb_slice_vector(
  duckdb_vector vector,
  duckdb_selection_vector sel,
  idx_t len
);
```

Parameters

- `vector`: The vector to slice.
- `sel`: The selection vector.
- `len`: The length of the selection vector.

`duckdb_vector_copy_sel` Copy the src vector to the dst with a selection vector that identifies which indices to copy.

Syntax

```
            void duckdb_vector_copy_sel(
  duckdb_vector src,
  duckdb_vector dst,
  duckdb_selection_vector sel,
  idx_t src_count,
  idx_t src_offset,
  idx_t dst_offset
);
```

Parameters

- `src`: The vector to copy from.
- `dst`: The vector to copy to.
- `sel`: The selection vector. The length of the selection vector should not be more than the length of the src vector
- `src_count`: The number of entries from selection vector to copy. Think of this as the effective length of the selection vector starting from index 0
- `src_offset`: The offset in the selection vector to copy from (important: actual number of items copied = src_count - src_offset).
- `dst_offset`: The offset in the dst vector to start copying to.

`duckdb_vector_reference_value` Copies the value from `value` to `vector`.

Syntax

```
            void duckdb_vector_reference_value(
  duckdb_vector vector,
  duckdb_value value
);
```

Parameters

- `vector`: The receiving vector.
- `value`: The value to copy into the vector.

`duckdb_vector_reference_vector` Changes `to_vector` to reference ‘from_vector. After, the vectors share ownership of the data.

Syntax

```
            void duckdb_vector_reference_vector(
  duckdb_vector to_vector,
  duckdb_vector from_vector
);
```

Parameters

- `to_vector`: The receiving vector.
- `from_vector`: The vector to reference.

`duckdb_validity_row_is_valid` Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

Syntax

```
           bool duckdb_validity_row_is_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`
- `row`: The row index

Return Value true if the row is valid, false otherwise

`duckdb_validity_set_row_validity` In a validity mask, sets a specific row to either valid or invalid.

Note that `duckdb_vector_ensure_validity_writable` should be called before calling `duckdb_vector_get_validity`, to ensure that there is a validity mask to write to.

Syntax

```
           void duckdb_validity_set_row_validity(
  uint64_t *validity,
  idx_t row,
  bool valid
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`.
- `row`: The row index
- `valid`: Whether or not to set the row to valid, or invalid

`duckdb_validity_set_row_invalid` In a validity mask, sets a specific row to invalid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to false.

Syntax

```
           void duckdb_validity_set_row_invalid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

`duckdb_validity_set_row_valid` In a validity mask, sets a specific row to valid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```
            void duckdb_validity_set_row_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

Values

The value class represents a single value of any type.

API Reference Overview

```
            void duckdb_destroy_value(duckdb_value *value);
duckdb_value duckdb_create_varchar(const char *text);
duckdb_value duckdb_create_varchar_length(const char *text, idx_t length);
duckdb_value duckdb_create_bool(bool input);
duckdb_value duckdb_create_int8(int8_t input);
duckdb_value duckdb_create_uint8(uint8_t input);
duckdb_value duckdb_create_int16(int16_t input);
duckdb_value duckdb_create_uint16(uint16_t input);
duckdb_value duckdb_create_int32(int32_t input);
duckdb_value duckdb_create_uint32(uint32_t input);
duckdb_value duckdb_create_uint64(uint64_t input);
duckdb_value duckdb_create_int64(int64_t val);
duckdb_value duckdb_create_hugeint(duckdb_hugeint input);
duckdb_value duckdb_create_uhugeint(duckdb_uhugeint input);
duckdb_value duckdb_create_bignum(duckdb_bignum input);
duckdb_value duckdb_create_decimal(duckdb_decimal input);
duckdb_value duckdb_create_float(float input);
duckdb_value duckdb_create_double(double input);
duckdb_value duckdb_create_date(duckdb_date input);
duckdb_value duckdb_create_time(duckdb_time input);
duckdb_value duckdb_create_time_ns(duckdb_time_ns input);
duckdb_value duckdb_create_time_tz_value(duckdb_time_tz value);
duckdb_value duckdb_create_timestamp(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_tz(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_s(duckdb_timestamp_s input);
duckdb_value duckdb_create_timestamp_ms(duckdb_timestamp_ms input);
duckdb_value duckdb_create_timestamp_ns(duckdb_timestamp_ns input);
duckdb_value duckdb_create_interval(duckdb_interval input);
duckdb_value duckdb_create_blob(const uint8_t *data, idx_t length);
duckdb_value duckdb_create_bit(duckdb_bit input);
duckdb_value duckdb_create_uuid(duckdb_uhugeint input);
bool duckdb_get_bool(duckdb_value val);
int8_t duckdb_get_int8(duckdb_value val);
uint8_t duckdb_get_uint8(duckdb_value val);
int16_t duckdb_get_int16(duckdb_value val);
uint16_t duckdb_get_uint16(duckdb_value val);
int32_t duckdb_get_int32(duckdb_value val);
uint32_t duckdb_get_uint32(duckdb_value val);
int64_t duckdb_get_int64(duckdb_value val);
uint64_t duckdb_get_uint64(duckdb_value val);
duckdb_hugeint duckdb_get_hugeint(duckdb_value val);
duckdb_uhugeint duckdb_get_uhugeint(duckdb_value val);
duckdb_bignum duckdb_get_bignum(duckdb_value val);
```

```
duckdb_decimal duckdb_get_decimal(duckdb_value val);
float duckdb_get_float(duckdb_value val);
double duckdb_get_double(duckdb_value val);
duckdb_date duckdb_get_date(duckdb_value val);
duckdb_time duckdb_get_time(duckdb_value val);
duckdb_time_ns duckdb_get_time_ns(duckdb_value val);
duckdb_time_tz duckdb_get_time_tz(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp_tz(duckdb_value val);
duckdb_timestamp_s duckdb_get_timestamp_s(duckdb_value val);
duckdb_timestamp_ms duckdb_get_timestamp_ms(duckdb_value val);
duckdb_timestamp_ns duckdb_get_timestamp_ns(duckdb_value val);
duckdb_interval duckdb_get_interval(duckdb_value val);
duckdb_logical_type duckdb_get_value_type(duckdb_value val);
duckdb_blob duckdb_get_blob(duckdb_value val);
duckdb_bit duckdb_get_bit(duckdb_value val);
duckdb_uhugeint duckdb_get_uuid(duckdb_value val);
char *duckdb_get_varchar(duckdb_value value);
duckdb_value duckdb_create_struct_value(duckdb_logical_type type, duckdb_value *values);
duckdb_value duckdb_create_list_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
duckdb_value duckdb_create_array_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
duckdb_value duckdb_create_map_value(duckdb_logical_type map_type, duckdb_value *keys, duckdb_value
*values, idx_t entry_count);
duckdb_value duckdb_create_union_value(duckdb_logical_type union_type, idx_t tag_index, duckdb_value
value);
idx_t duckdb_get_map_size(duckdb_value value);
duckdb_value duckdb_get_map_key(duckdb_value value, idx_t index);
duckdb_value duckdb_get_map_value(duckdb_value value, idx_t index);
bool duckdb_is_null_value(duckdb_value value);
duckdb_value duckdb_create_null_value();
idx_t duckdb_get_list_size(duckdb_value value);
duckdb_value duckdb_get_list_child(duckdb_value value, idx_t index);
duckdb_value duckdb_create_enum_value(duckdb_logical_type type, uint64_t value);
uint64_t duckdb_get_enum_value(duckdb_value value);
duckdb_value duckdb_get_struct_child(duckdb_value value, idx_t index);
char *duckdb_value_to_string(duckdb_value value);
```

`duckdb_destroy_value` Destroys the value and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_value(
  duckdb_value *value
);
```

Parameters

- `value`: The value to destroy.

`duckdb_create_varchar` Creates a value from a null-terminated string

Syntax

```
            duckdb_value duckdb_create_varchar(
  const char *text
);
```

Parameters

- `text`: The null-terminated string

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varchar_length` Creates a value from a string

Syntax

```
            duckdb_value duckdb_create_varchar_length(
  const char *text,
  idx_t length
);
```

Parameters

- `text`: The text
- `length`: The length of the text

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bool` Creates a value from a boolean

Syntax

```
            duckdb_value duckdb_create_bool(
  bool input
);
```

Parameters

- `input`: The boolean value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int8` Creates a value from an int8_t (a tinyint)

Syntax

```
            duckdb_value duckdb_create_int8(
  int8_t input
);
```

Parameters

- `input`: The tinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint8` Creates a value from a uint8_t (a utinyint)

Syntax

```
            duckdb_value duckdb_create_uint8(
  uint8_t input
);
```

Parameters

- `input`: The utinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int16` Creates a value from an int16_t (a smallint)

Syntax

```
            duckdb_value duckdb_create_int16(
  int16_t input
);
```

Parameters

- `input`: The smallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint16` Creates a value from a uint16_t (a usmallint)

Syntax

```
            duckdb_value duckdb_create_uint16(
  uint16_t input
);
```

Parameters

- `input`: The usmallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int32` Creates a value from an int32_t (an integer)

Syntax

```
            duckdb_value duckdb_create_int32(
  int32_t input
);
```

Parameters

- `input`: The integer value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint32` Creates a value from a uint32_t (a uinteger)

Syntax

```
            duckdb_value duckdb_create_uint32(
  uint32_t input
);
```

Parameters

- `input`: The uinteger value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint64` Creates a value from a uint64_t (a ubigint)

Syntax

```
            duckdb_value duckdb_create_uint64(
  uint64_t input
);
```

Parameters

- `input`: The ubigint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int64` Creates a value from an int64

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_int64(
  int64_t val
);
```

`duckdb_create_hugeint` Creates a value from a hugeint

Syntax

```
            duckdb_value duckdb_create_hugeint(
  duckdb_hugeint input
);
```

Parameters

- `input`: The hugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uhugeint` Creates a value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uhugeint(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The uhugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bignum` Creates a BIGNUM value from a duckdb_bignum

Syntax

```
            duckdb_value duckdb_create_bignum(
  duckdb_bignum input
);
```

Parameters

- `input`: The duckdb_bignum value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_decimal` Creates a DECIMAL value from a duckdb_decimal

Syntax

```
            duckdb_value duckdb_create_decimal(
  duckdb_decimal input
);
```

Parameters

- `input`: The duckdb_decimal value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_float` Creates a value from a float

Syntax

```
            duckdb_value duckdb_create_float(
  float input
);
```

Parameters

- `input`: The float value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_double` Creates a value from a double

Syntax

```
            duckdb_value duckdb_create_double(
  double input
);
```

Parameters

- `input`: The double value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_date` Creates a value from a date

Syntax

```
            duckdb_value duckdb_create_date(
  duckdb_date input
);
```

Parameters

- `input`: The date value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time` Creates a value from a time

Syntax

```
            duckdb_value duckdb_create_time(
  duckdb_time input
);
```

Parameters

- `input`: The time value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time_ns` Creates a value from a time_ns

Syntax

```
            duckdb_value duckdb_create_time_ns(
  duckdb_time_ns input
);
```

Parameters

- `input`: The time value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time_tz_value` Creates a value from a time_tz. Not to be confused with `duckdb_create_time_tz`, which creates a duckdb_time_tz_t.

Syntax

```
            duckdb_value duckdb_create_time_tz_value(
  duckdb_time_tz value
);
```

Parameters

- `value`: The time_tz value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp` Creates a TIMESTAMP value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_tz` Creates a TIMESTAMP_TZ value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp_tz(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_s` Creates a TIMESTAMP_S value from a duckdb_timestamp_s

Syntax

```
            duckdb_value duckdb_create_timestamp_s(
  duckdb_timestamp_s input
);
```

Parameters

- `input`: The duckdb_timestamp_s value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ms` Creates a TIMESTAMP_MS value from a duckdb_timestamp_ms

Syntax

```
            duckdb_value duckdb_create_timestamp_ms(
  duckdb_timestamp_ms input
);
```

Parameters

- `input`: The duckdb_timestamp_ms value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ns` Creates a TIMESTAMP_NS value from a duckdb_timestamp_ns

Syntax

```
            duckdb_value duckdb_create_timestamp_ns(
  duckdb_timestamp_ns input
);
```

Parameters

- `input`: The duckdb_timestamp_ns value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_interval` Creates a value from an interval

Syntax

```
            duckdb_value duckdb_create_interval(
  duckdb_interval input
);
```

Parameters

- `input`: The interval value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_blob` Creates a value from a blob

Syntax

```
            duckdb_value duckdb_create_blob(
  const uint8_t *data,
  idx_t length
);
```

Parameters

- `data`: The blob data
- `length`: The length of the blob data

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bit` Creates a BIT value from a duckdb_bit

Syntax

```
            duckdb_value duckdb_create_bit(
  duckdb_bit input
);
```

Parameters

- `input`: The duckdb_bit value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uuid` Creates a UUID value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uuid(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The duckdb_uhugeint containing the UUID

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_get_bool` Returns the boolean value of the given value.

Syntax

```
            bool duckdb_get_bool(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a boolean

Return Value A boolean, or false if the value cannot be converted

`duckdb_get_int8` Returns the int8_t value of the given value.

Syntax

```
            int8_t duckdb_get_int8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a tinyint

Return Value A int8_t, or MinValue if the value cannot be converted

`duckdb_get_uint8` Returns the uint8_t value of the given value.

Syntax

```
            uint8_t duckdb_get_uint8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a utinyint

Return Value A uint8_t, or MinValue if the value cannot be converted

`duckdb_get_int16` Returns the int16_t value of the given value.

Syntax

```
            int16_t duckdb_get_int16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a smallint

Return Value A int16_t, or MinValue if the value cannot be converted

`duckdb_get_uint16` Returns the uint16_t value of the given value.

Syntax

```
            uint16_t duckdb_get_uint16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a usmallint

Return Value A uint16_t, or MinValue if the value cannot be converted

`duckdb_get_int32` Returns the int32_t value of the given value.

Syntax

```
            int32_t duckdb_get_int32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing an integer

Return Value A int32_t, or MinValue if the value cannot be converted

`duckdb_get_uint32` Returns the uint32_t value of the given value.

Syntax

```
            uint32_t duckdb_get_uint32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uinteger

Return Value A uint32_t, or MinValue if the value cannot be converted

`duckdb_get_int64` Returns the int64_t value of the given value.

Syntax

```
            int64_t duckdb_get_int64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a bigint

Return Value A int64_t, or MinValue if the value cannot be converted

`duckdb_get_uint64` Returns the uint64_t value of the given value.

Syntax

```
            uint64_t duckdb_get_uint64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a ubigint

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_hugeint` Returns the hugeint value of the given value.

Syntax

```
            duckdb_hugeint duckdb_get_hugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a hugeint

Return Value A duckdb_hugeint, or MinValue if the value cannot be converted

`duckdb_get_uhugeint` Returns the uhugeint value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uhugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uhugeint

Return Value A duckdb_uhugeint, or MinValue if the value cannot be converted

`duckdb_get_bignum` Returns the duckdb_bignum value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_bignum duckdb_get_bignum(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a BIGNUM

Return Value A duckdb_bignum. The `data` field must be destroyed with `duckdb_free`.

`duckdb_get_decimal` Returns the duckdb_decimal value of the given value.

Syntax

```
            duckdb_decimal duckdb_get_decimal(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a DECIMAL

Return Value A duckdb_decimal, or MinValue if the value cannot be converted

`duckdb_get_float` Returns the float value of the given value.

Syntax

```
            float duckdb_get_float(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a float

Return Value A float, or NAN if the value cannot be converted

`duckdb_get_double` Returns the double value of the given value.

Syntax

```
            double duckdb_get_double(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a double

Return Value A double, or NAN if the value cannot be converted

`duckdb_get_date` Returns the date value of the given value.

Syntax

```
            duckdb_date duckdb_get_date(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a date

Return Value A duckdb_date, or MinValue if the value cannot be converted

`duckdb_get_time` Returns the time value of the given value.

Syntax

```
            duckdb_time duckdb_get_time(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time

Return Value A duckdb_time, or MinValue if the value cannot be converted

`duckdb_get_time_ns` Returns the time_ns value of the given value.

Syntax

```
            duckdb_time_ns duckdb_get_time_ns(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time_ns

Return Value A duckdb_time_ns, or MinValue if the value cannot be converted

`duckdb_get_time_tz` Returns the time_tz value of the given value.

Syntax

```
            duckdb_time_tz duckdb_get_time_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time_tz

Return Value A duckdb_time_tz, or MinValue if the value cannot be converted

`duckdb_get_timestamp` Returns the TIMESTAMP value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_tz` Returns the TIMESTAMP_TZ value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_TZ

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_s` Returns the duckdb_timestamp_s value of the given value.

Syntax

```
            duckdb_timestamp_s duckdb_get_timestamp_s(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_S

Return Value A duckdb_timestamp_s, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ms` Returns the duckdb_timestamp_ms value of the given value.

Syntax

```
            duckdb_timestamp_ms duckdb_get_timestamp_ms(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_MS

Return Value A duckdb_timestamp_ms, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ns` Returns the duckdb_timestamp_ns value of the given value.

Syntax

```
            duckdb_timestamp_ns duckdb_get_timestamp_ns(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_NS

Return Value A duckdb_timestamp_ns, or MinValue if the value cannot be converted

`duckdb_get_interval` Returns the interval value of the given value.

Syntax

```
            duckdb_interval duckdb_get_interval(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a interval

Return Value A duckdb_interval, or MinValue if the value cannot be converted

`duckdb_get_value_type` Returns the type of the given value. The type is valid as long as the value is not destroyed. The type itself must not be destroyed.

Syntax

```
            duckdb_logical_type duckdb_get_value_type(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value

Return Value A duckdb_logical_type.

`duckdb_get_blob` Returns the blob value of the given value.

Syntax

```
            duckdb_blob duckdb_get_blob(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a blob

Return Value A duckdb_blob

`duckdb_get_bit` Returns the duckdb_bit value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_bit duckdb_get_bit(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a BIT

Return Value A duckdb_bit

`duckdb_get_uuid` Returns a duckdb_uhugeint representing the UUID value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uuid(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a UUID

Return Value A duckdb_uhugeint representing the UUID value

`duckdb_get_varchar` Obtains a string representation of the given value. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_get_varchar(
  duckdb_value value
);
```

Parameters

- `value`: The value

Return Value The string value. This must be destroyed with `duckdb_free`.

`duckdb_create_struct_value` Creates a struct value from a type and an array of values. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_struct_value(
  duckdb_logical_type type,
  duckdb_value *values
);
```

Parameters

- `type`: The type of the struct
- `values`: The values for the struct fields

Return Value The struct value, or nullptr, if any child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_list_value` Creates a list value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_list_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the list
- `values`: The values for the list
- `value_count`: The number of values in the list

Return Value The list value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_array_value` Creates an array value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_array_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the array
- `values`: The values for the array
- `value_count`: The number of values in the array

Return Value The array value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_map_value` Creates a map value from a map type and two arrays, one for the keys and one for the values, each of length `entry_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_map_value(
  duckdb_logical_type map_type,
  duckdb_value *keys,
  duckdb_value *values,
  idx_t entry_count
);
```

Parameters

- `map_type`: The map type
- `keys`: The keys of the map
- `values`: The values of the map
- `entry_count`: The number of entries (key-value pairs) in the map

Return Value The map value, or nullptr, if the parameters are invalid.

`duckdb_create_union_value` Creates a union value from a union type, a tag index and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_union_value(
  duckdb_logical_type union_type,
  idx_t tag_index,
  duckdb_value value
);
```

Parameters

- `union_type`: The union type
- `tag_index`: The index of the tag of the union
- `value`: The value of the union for that tag

Return Value The union value, or nullptr, if the parameters are invalid.

`duckdb_get_map_size` Returns the number of elements in a MAP value.

Syntax

```
            idx_t duckdb_get_map_size(
  duckdb_value value
);
```

Parameters

- `value`: The MAP value.

Return Value The number of elements in the map.

`duckdb_get_map_key` Returns the MAP key at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_key(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the key.

Return Value The key as a duckdb_value.

`duckdb_get_map_value` Returns the MAP value at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_value(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the value.

Return Value The value as a duckdb_value.

`duckdb_is_null_value` Returns whether the value's type is SQLNULL or not.

Syntax

```
            bool duckdb_is_null_value(
  duckdb_value value
);
```

Parameters

- `value`: The value to check.

Return Value True, if the value's type is SQLNULL, otherwise false.

`duckdb_create_null_value` Creates a value of type SQLNULL.

Return Value The duckdb_value representing SQLNULL. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_null_value(

);
```

`duckdb_get_list_size` Returns the number of elements in a LIST value.

Syntax

```
idx_t duckdb_get_list_size(
  duckdb_value value
);
```

Parameters

- `value`: The LIST value.

Return Value The number of elements in the list.

`duckdb_get_list_child` Returns the LIST child at index as a duckdb_value.

Syntax

```
duckdb_value duckdb_get_list_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The LIST value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_create_enum_value` Creates an enum value from a type and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_enum_value(
  duckdb_logical_type type,
  uint64_t value
);
```

Parameters

- `type`: The type of the enum
- `value`: The value for the enum

Return Value The enum value, or nullptr.

`duckdb_get_enum_value`

Returns the enum value of the given value.

Syntax

```
            uint64_t duckdb_get_enum_value(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value containing an enum

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_struct_child`

Returns the STRUCT child at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_struct_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The STRUCT value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_value_to_string`

Returns the SQL string representation of the given value.

Syntax

```
            char *duckdb_value_to_string(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value.

Return Value The SQL string representation as a null-terminated string. The result must be freed with `duckdb_free`.

Types

DuckDB is a strongly typed database system. As such, every column has a single type specified. This type is constant over the entire column. That is to say, a column that is labeled as an `INTEGER` column will only contain `INTEGER` values.

DuckDB also supports columns of composite types. For example, it is possible to define an array of integers (`INTEGER[]`). It is also possible to define types as arbitrary structs (`ROW(i INTEGER, j VARCHAR)`). For that reason, native DuckDB type objects are not mere enums, but a class that can potentially be nested.

Types in the C API are modeled using an enum (`duckdb_type`) and a complex class (`duckdb_logical_type`). For most primitive types, e.g., integers or varchars, the enum is sufficient. For more complex types, such as lists, structs or decimals, the logical type must be used.

```
typedef enum DUCKDB_TYPE {
  DUCKDB_TYPE_INVALID = 0,
  DUCKDB_TYPE_BOOLEAN = 1,
  DUCKDB_TYPE_TINYINT = 2,
  DUCKDB_TYPE_SMALLINT = 3,
  DUCKDB_TYPE_INTEGER = 4,
  DUCKDB_TYPE_BIGINT = 5,
  DUCKDB_TYPE_UTINYINT = 6,
  DUCKDB_TYPE_USMALLINT = 7,
  DUCKDB_TYPE_UINTEGER = 8,
  DUCKDB_TYPE_UBIGINT = 9,
  DUCKDB_TYPE_FLOAT = 10,
  DUCKDB_TYPE_DOUBLE = 11,
  DUCKDB_TYPE_TIMESTAMP = 12,
  DUCKDB_TYPE_DATE = 13,
  DUCKDB_TYPE_TIME = 14,
  DUCKDB_TYPE_INTERVAL = 15,
  DUCKDB_TYPE_HUGEINT = 16,
  DUCKDB_TYPE_UHUGEINT = 32,
  DUCKDB_TYPE_VARCHAR = 17,
  DUCKDB_TYPE_BLOB = 18,
  DUCKDB_TYPE_DECIMAL = 19,
  DUCKDB_TYPE_TIMESTAMP_S = 20,
  DUCKDB_TYPE_TIMESTAMP_MS = 21,
  DUCKDB_TYPE_TIMESTAMP_NS = 22,
  DUCKDB_TYPE_ENUM = 23,
  DUCKDB_TYPE_LIST = 24,
  DUCKDB_TYPE_STRUCT = 25,
  DUCKDB_TYPE_MAP = 26,
  DUCKDB_TYPE_ARRAY = 33,
  DUCKDB_TYPE_UUID = 27,
  DUCKDB_TYPE_UNION = 28,
  DUCKDB_TYPE_BIT = 29,
  DUCKDB_TYPE_TIME_TZ = 30,
  DUCKDB_TYPE_TIMESTAMP_TZ = 31,
  DUCKDB_TYPE_ANY = 34,
  DUCKDB_TYPE_BIGNUM = 35,
  DUCKDB_TYPE_SQLNULL = 36,
  DUCKDB_TYPE_STRING_LITERAL = 37,
  DUCKDB_TYPE_INTEGER_LITERAL = 38,
  DUCKDB_TYPE_TIME_NS = 39,
  DUCKDB_TYPE_GEOMETRY = 40,
  DUCKDB_TYPE_VARIANT = 41,
```

```
  DUCKDB_TYPE_TIMESTAMP_TZ_NS = 42,
} duckdb_type;
```

Functions

The enum type of a column in the result can be obtained using the `duckdb_column_type` function. The logical type of a column can be obtained using the `duckdb_column_logical_type` function.

`duckdb_value`

The `duckdb_value` functions will auto-cast values as required. For example, it is no problem to use `duckdb_value_double` on a column of type `duckdb_value_int32`. The value will be auto-cast and returned as a double. Note that in certain cases the cast may fail. For example, this can happen if we request a `duckdb_value_int8` and the value does not fit within an `int8` value. In this case, a default value will be returned (usually `0` or `nullptr`). The same default value will also be returned if the corresponding value is `NULL`.

The `duckdb_value_is_null` function can be used to check if a specific value is `NULL` or not.

The exception to the auto-cast rule is the `duckdb_value_varchar_internal` function. This function does not auto-cast and only works for VARCHAR columns. The reason this function exists is that the result does not need to be freed.

> `duckdb_value_varchar` and `duckdb_value_blob` require the result to be de-allocated using `duckdb_free`.

`duckdb_fetch_chunk`

The `duckdb_fetch_chunk` function can be used to read data chunks from a DuckDB result set, and is the most efficient way of reading data from a DuckDB result using the C API. It is also the only way of reading data of certain types from a DuckDB result. For example, the `duckdb_value` functions do not support structural reading of composite types (lists or structs) or more complex types like enums or decimals.

For more information about data chunks, see the documentation on data chunks.

API Reference Overview

```
            duckdb_data_chunk duckdb_result_get_chunk(duckdb_result result, idx_t chunk_index);
bool duckdb_result_is_streaming(duckdb_result result);
idx_t duckdb_result_chunk_count(duckdb_result result);
duckdb_result_type duckdb_result_return_type(duckdb_result result);
```

Date Time Timestamp Helpers

```
            duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
bool duckdb_is_finite_date(duckdb_date date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time_tz duckdb_create_time_tz(int64_t micros, int32_t offset);
duckdb_time_tz_struct duckdb_from_time_tz(duckdb_time_tz micros);
duckdb_time duckdb_to_time(duckdb_time_struct time);
duckdb_timestamp_struct duckdb_from_timestamp(duckdb_timestamp ts);
duckdb_timestamp duckdb_to_timestamp(duckdb_timestamp_struct ts);
bool duckdb_is_finite_timestamp(duckdb_timestamp ts);
bool duckdb_is_finite_timestamp_s(duckdb_timestamp_s ts);
bool duckdb_is_finite_timestamp_ms(duckdb_timestamp_ms ts);
bool duckdb_is_finite_timestamp_ns(duckdb_timestamp_ns ts);
```

Hugeint Helpers

```
            double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
```

Decimal Helpers

```
            duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);
double duckdb_decimal_to_double(duckdb_decimal val);
```

Logical Type Interface

```
            duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
char *duckdb_logical_type_get_alias(duckdb_logical_type type);
void duckdb_logical_type_set_alias(duckdb_logical_type type, const char *alias);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_array_type(duckdb_logical_type type, idx_t array_size);
duckdb_logical_type duckdb_create_map_type(duckdb_logical_type key_type, duckdb_logical_type value_type);
duckdb_logical_type duckdb_create_union_type(duckdb_logical_type *member_types, const char **member_names,
idx_t member_count);
duckdb_logical_type duckdb_create_struct_type(duckdb_logical_type *member_types, const char **member_
names, idx_t member_count);
duckdb_logical_type duckdb_create_enum_type(const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_decimal_type(uint8_t width, uint8_t scale);
duckdb_type duckdb_get_type_id(duckdb_logical_type type);
uint8_t duckdb_decimal_width(duckdb_logical_type type);
uint8_t duckdb_decimal_scale(duckdb_logical_type type);
duckdb_type duckdb_decimal_internal_type(duckdb_logical_type type);
duckdb_type duckdb_enum_internal_type(duckdb_logical_type type);
uint32_t duckdb_enum_dictionary_size(duckdb_logical_type type);
char *duckdb_enum_dictionary_value(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_array_type_child_type(duckdb_logical_type type);
idx_t duckdb_array_type_array_size(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_value_type(duckdb_logical_type type);
idx_t duckdb_struct_type_child_count(duckdb_logical_type type);
char *duckdb_struct_type_child_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_struct_type_child_type(duckdb_logical_type type, idx_t index);
idx_t duckdb_union_type_member_count(duckdb_logical_type type);
char *duckdb_union_type_member_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_union_type_member_type(duckdb_logical_type type, idx_t index);
void duckdb_destroy_logical_type(duckdb_logical_type *type);
duckdb_state duckdb_register_logical_type(duckdb_connection con, duckdb_logical_type type, duckdb_create_
type_info info);
```

`duckdb_result_get_chunk`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetches a data chunk from the duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function supersedes all `duckdb_value` functions, as well as the `duckdb_column_data` and `duckdb_nullmask_data` functions. It results in significantly better performance, and should be preferred in newer code-bases.

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions).

Use `duckdb_result_chunk_count` to figure out how many chunks there are in the result.

Syntax

```
            duckdb_data_chunk duckdb_result_get_chunk(
  duckdb_result result,
  idx_t chunk_index
);
```

Parameters

- `result`: The result object to fetch the data chunk from.
- `chunk_index`: The chunk index to fetch from.

Return Value The resulting data chunk. Returns `NULL` if the chunk index is out of bounds.

`duckdb_result_is_streaming`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Checks if the type of the internal result is StreamQueryResult.

Syntax

```
            bool duckdb_result_is_streaming(
  duckdb_result result
);
```

Parameters

- `result`: The result object to check.

Return Value Whether or not the result object is of the type StreamQueryResult

`duckdb_result_chunk_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of data chunks present in the result.

Syntax

```
            idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value Number of data chunks present in the result.

`duckdb_result_return_type` Returns the return_type of the given result, or DUCKDB_RETURN_TYPE_INVALID on error

Syntax

```
            duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value The return_type

`duckdb_from_date`

Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

Syntax

```
            duckdb_date_struct duckdb_from_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value The `duckdb_date_struct` with the decomposed elements.

`duckdb_to_date`

Re-compose a `duckdb_date` from year, month and date (`duckdb_date_struct`).

Syntax

```
            duckdb_date duckdb_to_date(
  duckdb_date_struct date
);
```

Parameters

- `date`: The year, month and date stored in a `duckdb_date_struct`.

Return Value The `duckdb_date` element.

`duckdb_is_finite_date`

Test a `duckdb_date` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value True if the date is finite, false if it is ±infinity.

`duckdb_from_time` Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

Syntax

```
            duckdb_time_struct duckdb_from_time(
  duckdb_time time
);
```

Parameters

- `time`: The time object, as obtained from a `DUCKDB_TYPE_TIME` column.

Return Value The `duckdb_time_struct` with the decomposed elements.

`duckdb_create_time_tz` Create a `duckdb_time_tz` object from micros and a timezone offset.

Syntax

```
            duckdb_time_tz duckdb_create_time_tz(
  int64_t micros,
  int32_t offset
);
```

Parameters

- `micros`: The microsecond component of the time.
- `offset`: The timezone offset component of the time.

Return Value The `duckdb_time_tz` element.

`duckdb_from_time_tz` Decompose a TIME_TZ objects into micros and a timezone offset.

Use `duckdb_from_time` to further decompose the micros into hour, minute, second and microsecond.

Syntax

```
            duckdb_time_tz_struct duckdb_from_time_tz(
  duckdb_time_tz micros
);
```

Parameters

- `micros`: The time object, as obtained from a `DUCKDB_TYPE_TIME_TZ` column.

`duckdb_to_time` Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

Syntax

```
            duckdb_time duckdb_to_time(
  duckdb_time_struct time
);
```

Parameters

- `time`: The hour, minute, second and microsecond in a `duckdb_time_struct`.

Return Value The `duckdb_time` element.

`duckdb_from_timestamp` Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

Syntax

```
            duckdb_timestamp_struct duckdb_from_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The ts object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value The `duckdb_timestamp_struct` with the decomposed elements.

`duckdb_to_timestamp` Re-compose a `duckdb_timestamp` from a duckdb_timestamp_struct.

Syntax

```
            duckdb_timestamp duckdb_to_timestamp(
  duckdb_timestamp_struct ts
);
```

Parameters

- `ts`: The de-composed elements in a `duckdb_timestamp_struct`.

Return Value The `duckdb_timestamp` element.

`duckdb_is_finite_timestamp` Test a `duckdb_timestamp` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The duckdb_timestamp object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_s` Test a `duckdb_timestamp_s` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_s(
  duckdb_timestamp_s ts
);
```

Parameters

- `ts`: The duckdb_timestamp_s object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_S` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ms` Test a `duckdb_timestamp_ms` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ms(
  duckdb_timestamp_ms ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ms object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_MS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ns` Test a `duckdb_timestamp_ns` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ns(
  duckdb_timestamp_ns ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ns object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_NS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_hugeint_to_double` Converts a duckdb_hugeint object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

Syntax

```
            double duckdb_hugeint_to_double(
  duckdb_hugeint val
);
```

Parameters

- `val`: The hugeint value.

Return Value The converted `double` element.

`duckdb_double_to_hugeint` Converts a double value to a duckdb_hugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
            duckdb_hugeint duckdb_double_to_hugeint(
  double val
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_hugeint` element.

`duckdb_double_to_decimal` Converts a double value to a duckdb_decimal object.

If the conversion fails because the double value is too big, or the width/scale are invalid the result will be 0.

Syntax

```
            duckdb_decimal duckdb_double_to_decimal(
  double val,
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_decimal` element.

`duckdb_decimal_to_double` Converts a duckdb_decimal object (as obtained from a `DUCKDB_TYPE_DECIMAL` column) into a double.

Syntax

```
            double duckdb_decimal_to_double(
  duckdb_decimal val
);
```

Parameters

- `val`: The decimal value.

Return Value The converted `double` element.

`duckdb_create_logical_type` Creates a `duckdb_logical_type` from a primitive type. The resulting logical type must be destroyed with `duckdb_destroy_logical_type`.

Returns an invalid logical type, if type is: `DUCKDB_TYPE_INVALID`, `DUCKDB_TYPE_DECIMAL`, `DUCKDB_TYPE_ENUM`, `DUCKDB_TYPE_LIST`, `DUCKDB_TYPE_STRUCT`, `DUCKDB_TYPE_MAP`, `DUCKDB_TYPE_ARRAY`, or `DUCKDB_TYPE_UNION`.

Syntax

```
            duckdb_logical_type duckdb_create_logical_type(
  duckdb_type type
);
```

Parameters

- `type`: The primitive type to create.

Return Value The logical type.

`duckdb_logical_type_get_alias` Returns the alias of a duckdb_logical_type, if set, else `nullptr`. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_logical_type_get_alias(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type

Return Value The alias or `nullptr`

`duckdb_logical_type_set_alias` Sets the alias of a duckdb_logical_type.

Syntax

```
            void duckdb_logical_type_set_alias(
  duckdb_logical_type type,
  const char *alias
);
```

Parameters

- `type`: The logical type
- `alias`: The alias to set

`duckdb_create_list_type` Creates a LIST type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_list_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The child type of the list

Return Value The logical type.

`duckdb_create_array_type` Creates an ARRAY type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_array_type(
  duckdb_logical_type type,
  idx_t array_size
);
```

Parameters

- `type`: The child type of the array.
- `array_size`: The number of elements in the array.

Return Value The logical type.

`duckdb_create_map_type` Creates a MAP type from its key type and value type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_map_type(
  duckdb_logical_type key_type,
  duckdb_logical_type value_type
);
```

Parameters

- `key_type`: The map's key type.
- `value_type`: The map's value type.

Return Value The logical type.

`duckdb_create_union_type` Creates a UNION type from the passed arrays. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_union_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of union member types.
- `member_names`: The union member names.
- `member_count`: The number of union members.

Return Value The logical type.

`duckdb_create_struct_type` Creates a STRUCT type based on the member types and names. The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_struct_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of types of the struct members.
- `member_names`: The array of names of the struct members.
- `member_count`: The number of members of the struct.

Return Value The logical type.

`duckdb_create_enum_type` Creates an ENUM type from the passed member name array. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_enum_type(
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_names`: The array of names that the enum should consist of.
- `member_count`: The number of elements that were specified in the array.

Return Value The logical type.

`duckdb_create_decimal_type` Creates a DECIMAL type with the specified width and scale. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_decimal_type(
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `width`: The width of the decimal type
- `scale`: The scale of the decimal type

Return Value The logical type.

`duckdb_get_type_id` Retrieves the enum `duckdb_type` of a `duckdb_logical_type`.

Syntax

```
            duckdb_type duckdb_get_type_id(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type.

Return Value The `duckdb_type` id.

`duckdb_decimal_width` Retrieves the width of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_width(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The width of the decimal type

`duckdb_decimal_scale` Retrieves the scale of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_scale(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The scale of the decimal type

`duckdb_decimal_internal_type` Retrieves the internal storage type of a decimal type.

Syntax

```
            duckdb_type duckdb_decimal_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The internal type of the decimal type

`duckdb_enum_internal_type` Retrieves the internal storage type of an enum type.

Syntax

```
            duckdb_type duckdb_enum_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The internal type of the enum type

`duckdb_enum_dictionary_size` Retrieves the dictionary size of the enum type.

Syntax

```
            uint32_t duckdb_enum_dictionary_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The dictionary size of the enum type

`duckdb_enum_dictionary_value` Retrieves the dictionary value at the specified position from the enum.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_enum_dictionary_value(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The index in the dictionary

Return Value The string value of the enum type. Must be freed with `duckdb_free`.

`duckdb_list_type_child_type` Retrieves the child type of the given LIST type. Also accepts MAP types. The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_list_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type, either LIST or MAP.

Return Value The child type of the LIST or MAP type.

`duckdb_array_type_child_type` Retrieves the child type of the given ARRAY type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_array_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type. Must be ARRAY.

Return Value The child type of the ARRAY type.

`duckdb_array_type_array_size` Retrieves the array size of the given array type.

Syntax

```
            idx_t duckdb_array_type_array_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The fixed number of elements the values of this array type can store.

`duckdb_map_type_key_type`

Retrieves the key type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_key_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The key type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_value_type`

Retrieves the value type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_value_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The value type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_struct_type_child_count`

Returns the number of children of a struct type.

Syntax

```
            idx_t duckdb_struct_type_child_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The number of children of a struct type.

`duckdb_struct_type_child_name` Retrieves the name of the struct child.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_struct_type_child_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the struct type. Must be freed with `duckdb_free`.

`duckdb_struct_type_child_type` Retrieves the child type of the given struct type at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_struct_type_child_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the struct type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_union_type_member_count` Returns the number of members that the union type has.

Syntax

```
            idx_t duckdb_union_type_member_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type (union) object

Return Value The number of members of a union type.

`duckdb_union_type_member_name` Retrieves the name of the union member.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_union_type_member_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the union member. Must be freed with `duckdb_free`.

`duckdb_union_type_member_type`

Retrieves the child type of the given union member at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_union_type_member_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the union member. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_destroy_logical_type`

Destroys the logical type and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_logical_type(
  duckdb_logical_type *type
);
```

Parameters

- `type`: The logical type to destroy.

`duckdb_register_logical_type`

Registers a custom type within the given connection. The type must have an alias

Syntax

```
            duckdb_state duckdb_register_logical_type(
  duckdb_connection con,
  duckdb_logical_type type,
  duckdb_create_type_info info
);
```

Parameters

- `con`: The connection to use
- `type`: The custom type to register

Return Value Whether or not the registration was successful.

Prepared Statements

A prepared statement is a parameterized query. The query is prepared with question marks (`?`) or dollar symbols (`$1`) indicating the parameters of the query. Values can then be bound to these parameters, after which the prepared statement can be executed using those parameters. A single query can be prepared once and executed many times.

Prepared statements are useful to:

- Easily supply parameters to functions while avoiding string concatenation/SQL injection attacks.
- Speeding up queries that will be executed many times with different parameters.

DuckDB supports prepared statements in the C API with the `duckdb_prepare` method. The `duckdb_bind` family of functions is used to supply values for subsequent execution of the prepared statement using `duckdb_execute_prepared`. After we are done with the prepared statement it can be cleaned up using the `duckdb_destroy_prepare` method.

Example

```
duckdb_prepared_statement stmt;
duckdb_result result;
if (duckdb_prepare(con, "INSERT INTO integers VALUES ($1, $2)", &stmt) == DuckDBError) {
    // handle error
}

duckdb_bind_int32(stmt, 1, 42); // the parameter index starts counting at 1!
duckdb_bind_int32(stmt, 2, 43);
// NULL as second parameter means no result set is requested
duckdb_execute_prepared(stmt, NULL);
duckdb_destroy_prepare(&stmt);

// we can also query result sets using prepared statements
if (duckdb_prepare(con, "SELECT * FROM integers WHERE i = ?", &stmt) == DuckDBError) {
    // handle error
}
duckdb_bind_int32(stmt, 1, 42);
duckdb_execute_prepared(stmt, &result);

// do something with result

// clean up
duckdb_destroy_result(&result);
duckdb_destroy_prepare(&stmt);
```

After calling `duckdb_prepare`, the prepared statement parameters can be inspected using `duckdb_nparams` and `duckdb_param_type`. In case the prepare fails, the error can be obtained through `duckdb_prepare_error`.

It is not required that the `duckdb_bind` family of functions matches the prepared statement parameter type exactly. The values will be auto-cast to the required value as required. For example, calling `duckdb_bind_int8` on a parameter type of `DUCKDB_TYPE_INTEGER` will work as expected.

> **Warning.** Do **not** use prepared statements to insert large amounts of data into DuckDB. Instead it is recommended to use the Appender.

API Reference Overview

```
            duckdb_state duckdb_prepare(duckdb_connection connection, const char *query, duckdb_prepared_
statement *out_prepared_statement);
void duckdb_destroy_prepare(duckdb_prepared_statement *prepared_statement);
const char *duckdb_prepare_error(duckdb_prepared_statement prepared_statement);
idx_t duckdb_nparams(duckdb_prepared_statement prepared_statement);
const char *duckdb_parameter_name(duckdb_prepared_statement prepared_statement, idx_t index);
duckdb_type duckdb_param_type(duckdb_prepared_statement prepared_statement, idx_t param_idx);
duckdb_logical_type duckdb_param_logical_type(duckdb_prepared_statement prepared_statement, idx_t param_
idx);
duckdb_state duckdb_clear_bindings(duckdb_prepared_statement prepared_statement);
duckdb_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement statement);
idx_t duckdb_prepared_statement_column_count(duckdb_prepared_statement prepared_statement);
const char *duckdb_prepared_statement_column_name(duckdb_prepared_statement prepared_statement, idx_t col_
idx);
duckdb_logical_type duckdb_prepared_statement_column_logical_type(duckdb_prepared_statement prepared_
statement, idx_t col_idx);
duckdb_type duckdb_prepared_statement_column_type(duckdb_prepared_statement prepared_statement, idx_t col_
idx);
```

`duckdb_prepare` Create a prepared statement object from a query.

Note that after calling `duckdb_prepare`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
            duckdb_state duckdb_prepare(
  duckdb_connection connection,
  const char *query,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`: The connection object
- `query`: The SQL query to prepare
- `out_prepared_statement`: The resulting prepared statement object

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_prepare` Closes the prepared statement and de-allocates all memory allocated for the statement.

Syntax

```
            void duckdb_destroy_prepare(
  duckdb_prepared_statement *prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to destroy.

`duckdb_prepare_error` Returns the error message associated with the given prepared statement. If the prepared statement has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_prepare` is called.

Syntax

```
            const char *duckdb_prepare_error(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_nparams` Returns the number of parameters that can be provided to the given prepared statement.

Returns 0 if the query was not successfully prepared.

Syntax

```
            idx_t duckdb_nparams(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the number of parameters for.

`duckdb_parameter_name` Returns the name used to identify the parameter The returned string should be freed using `duckdb_free`.

Returns NULL if the index is out of range for the provided prepared statement.

Syntax

```
            const char *duckdb_parameter_name(
  duckdb_prepared_statement prepared_statement,
  idx_t index
);
```

Parameters

- `prepared_statement`: The prepared statement for which to get the parameter name from.

`duckdb_param_type` Returns the parameter type for the parameter at the given index.

Returns `DUCKDB_TYPE_INVALID` if the parameter index is out of range or the statement was not successfully prepared.

Syntax

```
            duckdb_type duckdb_param_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The parameter type

`duckdb_param_logical_type` Returns the logical type for the parameter at the given index.

Returns `nullptr` if the parameter index is out of range or the statement was not successfully prepared.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
           duckdb_logical_type duckdb_param_logical_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The logical type of the parameter

`duckdb_clear_bindings` Clear the params bind to the prepared statement.

Syntax

```
           duckdb_state duckdb_clear_bindings(
  duckdb_prepared_statement prepared_statement
);
```

`duckdb_prepared_statement_type` Returns the statement type of the statement to be executed

Syntax

```
           duckdb_statement_type duckdb_prepared_statement_type(
  duckdb_prepared_statement statement
);
```

Parameters

- `statement`: The prepared statement.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_prepared_statement_column_count` Returns the number of columns present in the result of the prepared statement. If any of the column types are invalid, the result will be 1.

Syntax

```
            idx_t duckdb_prepared_statement_column_count(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement.

Return Value The number of columns present in the result of the prepared statement.

`duckdb_prepared_statement_column_name` Returns the name of the specified column of the result of the prepared_statement. The returned string should be freed using `duckdb_free`.

Returns `nullptr` if the column is out of range.

Syntax

```
            const char *duckdb_prepared_statement_column_name(
  duckdb_prepared_statement prepared_statement,
  idx_t col_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `col_idx`: The column index.

Return Value The column name of the specified column.

`duckdb_prepared_statement_column_logical_type` Returns the column type of the specified column of the result of the prepared_statement.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range. The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_prepared_statement_column_logical_type(
  duckdb_prepared_statement prepared_statement,
  idx_t col_idx
);
```

Parameters

- `prepared_statement`: The prepared statement to fetch the column type from.
- `col_idx`: The column index.

Return Value The logical type of the specified column.

`duckdb_prepared_statement_column_type` Returns the column type of the specified column of the result of the prepared_statement.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
           duckdb_type duckdb_prepared_statement_column_type(
  duckdb_prepared_statement prepared_statement,
  idx_t col_idx
);
```

Parameters

- `prepared_statement`: The prepared statement to fetch the column type from.
- `col_idx`: The column index.

Return Value The type of the specified column.

Appender

Appenders are the most efficient way of loading data into DuckDB from within the C interface, and are recommended for fast data loading. The appender is much faster than using prepared statements or individual `INSERT INTO` statements.

Appends are made in row-wise format. For every column, a `duckdb_append_[type]` call should be made, after which the row should be finished by calling `duckdb_appender_end_row`. After all rows have been appended, `duckdb_appender_destroy` should be used to finalize the appender and clean up the resulting memory.

Note that `duckdb_appender_destroy` should always be called on the resulting appender, even if the function returns `DuckDBError`.

Example

```
duckdb_query(con, "CREATE TABLE people (id INTEGER, name VARCHAR)", NULL);

duckdb_appender appender;
if (duckdb_appender_create(con, NULL, "people", &appender) == DuckDBError) {
  // handle error
}
// append the first row (1, Mark)
duckdb_append_int32(appender, 1);
duckdb_append_varchar(appender, "Mark");
duckdb_appender_end_row(appender);

// append the second row (2, Hannes)
duckdb_append_int32(appender, 2);
duckdb_append_varchar(appender, "Hannes");
duckdb_appender_end_row(appender);

// finish appending and flush all the rows to the table
duckdb_appender_destroy(&appender);
```

API Reference Overview

```
           duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const
char *table, duckdb_appender *out_appender);
duckdb_state duckdb_appender_create_ext(duckdb_connection connection, const char *catalog, const char
*schema, const char *table, duckdb_appender *out_appender);
duckdb_state duckdb_appender_create_query(duckdb_connection connection, const char *query, idx_t column_
count, duckdb_logical_type *types, const char *table_name, const char **column_names, duckdb_appender *out_
appender);
idx_t duckdb_appender_column_count(duckdb_appender appender);
duckdb_logical_type duckdb_appender_column_type(duckdb_appender appender, idx_t col_idx);
```

```
const char *duckdb_appender_error(duckdb_appender appender);
duckdb_error_data duckdb_appender_error_data(duckdb_appender appender);
duckdb_state duckdb_appender_flush(duckdb_appender appender);
duckdb_state duckdb_appender_close(duckdb_appender appender);
duckdb_state duckdb_appender_destroy(duckdb_appender *appender);
duckdb_state duckdb_appender_add_column(duckdb_appender appender, const char *name);
duckdb_state duckdb_appender_clear_columns(duckdb_appender appender);
duckdb_state duckdb_appender_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_append_default(duckdb_appender appender);
duckdb_state duckdb_append_default_to_chunk(duckdb_appender appender, duckdb_data_chunk chunk, idx_t col,
idx_t row);
duckdb_state duckdb_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_append_int32(duckdb_appender appender, int32_t value);
duckdb_state duckdb_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_append_hugeint(duckdb_appender appender, duckdb_hugeint value);
duckdb_state duckdb_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_append_uhugeint(duckdb_appender appender, duckdb_uhugeint value);
duckdb_state duckdb_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_append_date(duckdb_appender appender, duckdb_date value);
duckdb_state duckdb_append_time(duckdb_appender appender, duckdb_time value);
duckdb_state duckdb_append_timestamp(duckdb_appender appender, duckdb_timestamp value);
duckdb_state duckdb_append_interval(duckdb_appender appender, duckdb_interval value);
duckdb_state duckdb_append_varchar(duckdb_appender appender, const char *val);
duckdb_state duckdb_append_varchar_length(duckdb_appender appender, const char *val, idx_t length);
duckdb_state duckdb_append_blob(duckdb_appender appender, const void *data, idx_t length);
duckdb_state duckdb_append_null(duckdb_appender appender);
duckdb_state duckdb_append_value(duckdb_appender appender, duckdb_value value);
duckdb_state duckdb_append_data_chunk(duckdb_appender appender, duckdb_data_chunk chunk);
```

`duckdb_appender_create` Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_create_ext` Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create_ext(
  duckdb_connection connection,
  const char *catalog,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `catalog`: The catalog of the table to append to, or `nullptr` for the default catalog.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_create_query` Creates an appender object that executes the given query with any data appended to it.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create_query(
  duckdb_connection connection,
  const char *query,
  idx_t column_count,
  duckdb_logical_type *types,
  const char *table_name,
  const char **column_names,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `query`: The query to execute, can be an INSERT, DELETE, UPDATE or MERGE INTO statement.
- `column_count`: The number of columns to append.
- `types`: The types of the columns to append.
- `table_name`: (optionally) the table name used to refer to the appended data, defaults to "appended_data".
- `column_names`: (optionally) the list of column names, defaults to "col1", "col2", ...
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_column_count` Returns the number of columns that belong to the appender. If there is no active column list, then this equals the table's physical columns.

Syntax

```
            idx_t duckdb_appender_column_count(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the column count from.

Return Value The number of columns in the data chunks.

`duckdb_appender_column_type` Returns the type of the column at the specified index. This is either a type in the active column list, or the same type as a column in the receiving table.

Note: The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_appender_column_type(
  duckdb_appender appender,
  idx_t col_idx
);
```

Parameters

- `appender`: The appender to get the column type from.
- `col_idx`: The index of the column to get the type of.

Return Value The `duckdb_logical_type` of the column.

`duckdb_appender_error`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release. Use duckdb_appender_error_data instead.

Returns the error message associated with the appender. If the appender has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_appender_destroy` is called.

Syntax

```
            const char *duckdb_appender_error(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_appender_error_data` Returns the error data associated with the appender. Must be destroyed with duckdb_destroy_error_data.

Syntax

```
            duckdb_error_data duckdb_appender_error_data(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the error data from.

Return Value The error data.

`duckdb_appender_flush` Flush the appender to the table, forcing the cache of the appender to be cleared. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. It is not possible to append more values. Call duckdb_appender_error_data to obtain the error data followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_flush(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_close` Closes the appender by flushing all intermediate states and closing it for further appends. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. Call duckdb_appender_error_data to obtain the error data followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_close(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush and close.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_destroy` Closes the appender by flushing all intermediate states to the table and destroying it. By destroying it, this function de-allocates all memory associated with the appender. If flushing the data triggers a constraint violation, then all data is invalidated, and this function returns DuckDBError. Due to the destruction of the appender, it is no longer possible to obtain the specific error message with duckdb_appender_error. Therefore, call duckdb_appender_close before destroying the appender, if you need insights into the specific error.

Syntax

```
            duckdb_state duckdb_appender_destroy(
  duckdb_appender *appender
);
```

Parameters

- `appender`: The appender to flush, close and destroy.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_add_column` Appends a column to the active column list of the appender. Immediately flushes all previous data.

The active column list specifies all columns that are expected when flushing the data. Any non-active columns are filled with their default values, or NULL.

Syntax

```
            duckdb_state duckdb_appender_add_column(
  duckdb_appender appender,
  const char *name
);
```

Parameters

- `appender`: The appender to add the column to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_clear_columns` Removes all columns from the active column list of the appender, resetting the appender to treat all columns as active. Immediately flushes all previous data.

Syntax

```
            duckdb_state duckdb_appender_clear_columns(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to clear the columns from.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_begin_row` A nop function, provided for backwards compatibility reasons. Does nothing. Only `duckdb_appender_end_row` is required.

Syntax

```
            duckdb_state duckdb_appender_begin_row(
  duckdb_appender appender
);
```

`duckdb_appender_end_row` Finish the current row of appends. After end_row is called, the next row can be appended.

Syntax

```
            duckdb_state duckdb_appender_end_row(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_default` Append a DEFAULT value (NULL if DEFAULT not available for column) to the appender.

Syntax

```
            duckdb_state duckdb_append_default(
  duckdb_appender appender
);
```

`duckdb_append_default_to_chunk` Append a DEFAULT value, at the specified row and column, (NULL if DEFAULT not available for column) to the chunk created from the specified appender. The default value of the column must be a constant value. Non-deterministic expressions like nextval('seq') or random() are not supported.

Syntax

```
            duckdb_state duckdb_append_default_to_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk,
  idx_t col,
  idx_t row
);
```

Parameters

- `appender`: The appender to get the default value from.
- `chunk`: The data chunk to append the default value to.
- `col`: The chunk column index to append the default value to.
- `row`: The chunk row index to append the default value to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_bool` Append a bool value to the appender.

Syntax

```
            duckdb_state duckdb_append_bool(
  duckdb_appender appender,
  bool value
);
```

`duckdb_append_int8` Append an int8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int8(
  duckdb_appender appender,
  int8_t value
);
```

`duckdb_append_int16` Append an int16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int16(
  duckdb_appender appender,
  int16_t value
);
```

`duckdb_append_int32` Append an int32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int32(
  duckdb_appender appender,
  int32_t value
);
```

`duckdb_append_int64` Append an int64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int64(
  duckdb_appender appender,
  int64_t value
);
```

`duckdb_append_hugeint` Append a duckdb_hugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_hugeint(
  duckdb_appender appender,
  duckdb_hugeint value
);
```

`duckdb_append_uint8` Append a uint8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint8(
  duckdb_appender appender,
  uint8_t value
);
```

`duckdb_append_uint16` Append a uint16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint16(
  duckdb_appender appender,
  uint16_t value
);
```

`duckdb_append_uint32` Append a uint32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint32(
  duckdb_appender appender,
  uint32_t value
);
```

`duckdb_append_uint64` Append a uint64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint64(
  duckdb_appender appender,
  uint64_t value
);
```

`duckdb_append_uhugeint` Append a duckdb_uhugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_uhugeint(
  duckdb_appender appender,
  duckdb_uhugeint value
);
```

`duckdb_append_float` Append a float value to the appender.

Syntax

```
            duckdb_state duckdb_append_float(
  duckdb_appender appender,
  float value
);
```

`duckdb_append_double` Append a double value to the appender.

Syntax

```
            duckdb_state duckdb_append_double(
  duckdb_appender appender,
  double value
);
```

`duckdb_append_date` Append a duckdb_date value to the appender.

Syntax

```
            duckdb_state duckdb_append_date(
  duckdb_appender appender,
  duckdb_date value
);
```

`duckdb_append_time` Append a duckdb_time value to the appender.

Syntax

```
            duckdb_state duckdb_append_time(
  duckdb_appender appender,
  duckdb_time value
);
```

`duckdb_append_timestamp` Append a duckdb_timestamp value to the appender.

Syntax

```
            duckdb_state duckdb_append_timestamp(
  duckdb_appender appender,
  duckdb_timestamp value
);
```

`duckdb_append_interval` Append a duckdb_interval value to the appender.

Syntax

```
            duckdb_state duckdb_append_interval(
  duckdb_appender appender,
  duckdb_interval value
);
```

`duckdb_append_varchar` Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar(
  duckdb_appender appender,
  const char *val
);
```

`duckdb_append_varchar_length` Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar_length(
  duckdb_appender appender,
  const char *val,
  idx_t length
);
```

`duckdb_append_blob` Append a blob value to the appender.

Syntax

```
            duckdb_state duckdb_append_blob(
  duckdb_appender appender,
  const void *data,
  idx_t length
);
```

`duckdb_append_null` Append a NULL value to the appender (of any type).

Syntax

```
            duckdb_state duckdb_append_null(
  duckdb_appender appender
);
```

`duckdb_append_value` Append a duckdb_value to the appender.

Syntax

```
            duckdb_state duckdb_append_value(
  duckdb_appender appender,
  duckdb_value value
);
```

`duckdb_append_data_chunk` Appends a pre-filled data chunk to the specified appender. Attempts casting, if the data chunk types do not match the active appender types.

Syntax

```
            duckdb_state duckdb_append_data_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk
);
```

Parameters

- `appender`: The appender to append to.
- `chunk`: The data chunk to append.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

Table Functions

The table function API can be used to define a table function that can then be called from within DuckDB in the `FROM` clause of a query.

API Reference Overview

```
            duckdb_table_function duckdb_create_table_function();
void duckdb_destroy_table_function(duckdb_table_function *table_function);
void duckdb_table_function_set_name(duckdb_table_function table_function, const char *name);
void duckdb_table_function_add_parameter(duckdb_table_function table_function, duckdb_logical_type type);
void duckdb_table_function_add_named_parameter(duckdb_table_function table_function, const char *name,
duckdb_logical_type type);
```

```
void duckdb_table_function_set_extra_info(duckdb_table_function table_function, void *extra_info, duckdb_
delete_callback_t destroy);
void duckdb_table_function_set_bind(duckdb_table_function table_function, duckdb_table_function_bind_t
bind);
void duckdb_table_function_set_init(duckdb_table_function table_function, duckdb_table_function_init_t
init);
void duckdb_table_function_set_local_init(duckdb_table_function table_function, duckdb_table_function_
init_t init);
void duckdb_table_function_set_function(duckdb_table_function table_function, duckdb_table_function_t
function);
void duckdb_table_function_supports_projection_pushdown(duckdb_table_function table_function, bool
pushdown);
duckdb_state duckdb_register_table_function(duckdb_connection con, duckdb_table_function function);
```

Table Function Bind

```
            void *duckdb_bind_get_extra_info(duckdb_bind_info info);
void duckdb_table_function_get_client_context(duckdb_bind_info info, duckdb_client_context *out_context);
void duckdb_bind_add_result_column(duckdb_bind_info info, const char *name, duckdb_logical_type type);
idx_t duckdb_bind_get_parameter_count(duckdb_bind_info info);
duckdb_value duckdb_bind_get_parameter(duckdb_bind_info info, idx_t index);
duckdb_value duckdb_bind_get_named_parameter(duckdb_bind_info info, const char *name);
void duckdb_bind_set_bind_data(duckdb_bind_info info, void *bind_data, duckdb_delete_callback_t destroy);
void duckdb_bind_set_cardinality(duckdb_bind_info info, idx_t cardinality, bool is_exact);
void duckdb_bind_set_error(duckdb_bind_info info, const char *error);
```

Table Function Init

```
            void *duckdb_init_get_extra_info(duckdb_init_info info);
void *duckdb_init_get_bind_data(duckdb_init_info info);
void duckdb_init_set_init_data(duckdb_init_info info, void *init_data, duckdb_delete_callback_t destroy);
idx_t duckdb_init_get_column_count(duckdb_init_info info);
idx_t duckdb_init_get_column_index(duckdb_init_info info, idx_t column_index);
void duckdb_init_set_max_threads(duckdb_init_info info, idx_t max_threads);
void duckdb_init_set_error(duckdb_init_info info, const char *error);
```

Table Function

```
            void *duckdb_function_get_extra_info(duckdb_function_info info);
void *duckdb_function_get_bind_data(duckdb_function_info info);
void *duckdb_function_get_init_data(duckdb_function_info info);
void *duckdb_function_get_local_init_data(duckdb_function_info info);
void duckdb_function_set_error(duckdb_function_info info, const char *error);
```

`duckdb_create_table_function` Creates a new empty table function.

The return value should be destroyed with `duckdb_destroy_table_function`.

Return Value The table function object.

Syntax

```
            duckdb_table_function duckdb_create_table_function(

);
```

`duckdb_destroy_table_function` Destroys the given table function object.

Syntax

```
            void duckdb_destroy_table_function(
  duckdb_table_function *table_function
);
```

Parameters

- `table_function`: The table function to destroy

`duckdb_table_function_set_name` Sets the name of the given table function.

Syntax

```
            void duckdb_table_function_set_name(
  duckdb_table_function table_function,
  const char *name
);
```

Parameters

- `table_function`: The table function
- `name`: The name of the table function

`duckdb_table_function_add_parameter` Adds a parameter to the table function.

Syntax

```
            void duckdb_table_function_add_parameter(
  duckdb_table_function table_function,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_table_function_add_named_parameter` Adds a named parameter to the table function.

Syntax

```
            void duckdb_table_function_add_named_parameter(
  duckdb_table_function table_function,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `name`: The parameter name.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_table_function_set_extra_info` Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```
            void duckdb_table_function_set_extra_info(
  duckdb_table_function table_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `table_function`: The table function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_table_function_set_bind` Sets the bind function of the table function.

Syntax

```
            void duckdb_table_function_set_bind(
  duckdb_table_function table_function,
  duckdb_table_function_bind_t bind
);
```

Parameters

- `table_function`: The table function
- `bind`: The bind function

`duckdb_table_function_set_init` Sets the init function of the table function.

Syntax

```
            void duckdb_table_function_set_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_local_init` Sets the thread-local init function of the table function.

Syntax

```
            void duckdb_table_function_set_local_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_function` Sets the main function of the table function.

Syntax

```
            void duckdb_table_function_set_function(
  duckdb_table_function table_function,
  duckdb_table_function_t function
);
```

Parameters

- `table_function`: The table function
- `function`: The function

`duckdb_table_function_supports_projection_pushdown` Sets whether or not the given table function supports projection pushdown.

If this is set to true, the system will provide a list of all required columns in the `init` stage through the `duckdb_init_get_column_count` and `duckdb_init_get_column_index` functions. If this is set to false (the default), the system will expect all columns to be projected.

Syntax

```
            void duckdb_table_function_supports_projection_pushdown(
  duckdb_table_function table_function,
  bool pushdown
);
```

Parameters

- `table_function`: The table function
- `pushdown`: True if the table function supports projection pushdown, false otherwise.

`duckdb_register_table_function` Register the table function object within the given connection.

The function requires at least a name, a bind function, an init function and a main function.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_table_function(
  duckdb_connection con,
  duckdb_table_function function
);
```

Parameters

- `con`: The connection to register it in.
- `function`: The function pointer

Return Value Whether or not the registration was successful.

`duckdb_bind_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_table_function_get_client_context` Retrieves the client context of the bind info of a table function.

Syntax

```
            void duckdb_table_function_get_client_context(
  duckdb_bind_info info,
  duckdb_client_context *out_context
);
```

Parameters

- `info`: The bind info object of the table function.
- `out_context`: The client context of the bind info. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_bind_add_result_column` Adds a result column to the output of the table function.

Syntax

```
            void duckdb_bind_add_result_column(
  duckdb_bind_info info,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `info`: The table function's bind info.
- `name`: The column name.
- `type`: The logical column type.

`duckdb_bind_get_parameter_count` Retrieves the number of regular (non-named) parameters to the function.

Syntax

```
            idx_t duckdb_bind_get_parameter_count(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The number of parameters

`duckdb_bind_get_parameter` Retrieves the parameter at the given index.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_parameter(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`: The info object
- `index`: The index of the parameter to get

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_get_named_parameter` Retrieves a named parameter with the given name.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_named_parameter(
  duckdb_bind_info info,
  const char *name
);
```

Parameters

- `info`: The info object
- `name`: The name of the parameter

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_set_bind_data` Sets the user-provided bind data in the bind object of the table function. This object can be retrieved again during execution.

Syntax

```
            void duckdb_bind_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The bind info of the table function.
- `bind_data`: The bind data object.
- `destroy`: The callback to destroy the bind data (if any).

`duckdb_bind_set_cardinality` Sets the cardinality estimate for the table function, used for optimization.

Syntax

```
            void duckdb_bind_set_cardinality(
  duckdb_bind_info info,
  idx_t cardinality,
  bool is_exact
);
```

Parameters

- `info`: The bind data object.
- `is_exact`: Whether or not the cardinality estimate is exact, or an approximation

`duckdb_bind_set_error` Report that an error has occurred while calling bind on a table function.

Syntax

```
            void duckdb_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_init_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_init_get_extra_info(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_init_get_bind_data` Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_init_get_bind_data(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The bind data object

`duckdb_init_set_init_data` Sets the user-provided init data in the init object. This object can be retrieved again during execution.

Syntax

```
            void duckdb_init_set_init_data(
  duckdb_init_info info,
  void *init_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The info object
- `init_data`: The init data object.
- `destroy`: The callback that will be called to destroy the init data (if any)

`duckdb_init_get_column_count` Returns the number of projected columns.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_count(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The number of projected columns.

`duckdb_init_get_column_index` Returns the column index of the projected column at the specified position.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_index(
  duckdb_init_info info,
  idx_t column_index
);
```

Parameters

- `info`: The info object
- `column_index`: The index at which to get the projected column index, from 0..duckdb_init_get_column_count(info)

Return Value The column index of the projected column.

`duckdb_init_set_max_threads`

Sets how many threads can process this table function in parallel (default: 1)

Syntax

```
            void duckdb_init_set_max_threads(
  duckdb_init_info info,
  idx_t max_threads
);
```

Parameters

- `info`: The info object
- `max_threads`: The maximum amount of threads that can process this table function

`duckdb_init_set_error`

Report that an error has occurred while calling init.

Syntax

```
            void duckdb_init_set_error(
  duckdb_init_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_function_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_function_get_bind_data`

Gets the table function's bind data set by `duckdb_bind_set_bind_data`.

Note that the bind data is read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The function info object.

Return Value

The bind data object.

`duckdb_function_get_init_data`

Gets the init data set by `duckdb_init_set_init_data` during the init.

Syntax

```
            void *duckdb_function_get_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value

The init data object

`duckdb_function_get_local_init_data`

Gets the thread-local init data set by `duckdb_init_set_init_data` during the local_init.

Syntax

```
            void *duckdb_function_get_local_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value

The init data object

`duckdb_function_set_error`

Report that an error has occurred while executing the function.

Syntax

```
            void duckdb_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

Replacement Scans

The replacement scan API can be used to register a callback that is called when a table is read that does not exist in the catalog. For example, when a query such as `SELECT * FROM my_table` is executed and `my_table` does not exist, the replacement scan callback will be called with `my_table` as parameter. The replacement scan can then insert a table function with a specific parameter to replace the read of the table.

API Reference Overview

```
            void duckdb_add_replacement_scan(duckdb_database db, duckdb_replacement_callback_t replacement,
            void *extra_data, duckdb_delete_callback_t delete_callback);
void duckdb_replacement_scan_set_function_name(duckdb_replacement_scan_info info, const char *function_
name);
void duckdb_replacement_scan_add_parameter(duckdb_replacement_scan_info info, duckdb_value parameter);
void duckdb_replacement_scan_set_error(duckdb_replacement_scan_info info, const char *error);
```

`duckdb_add_replacement_scan` Add a replacement scan definition to the specified database.

Syntax

```
            void duckdb_add_replacement_scan(
  duckdb_database db,
  duckdb_replacement_callback_t replacement,
  void *extra_data,
  duckdb_delete_callback_t delete_callback
);
```

Parameters

- db: The database object to add the replacement scan to
- `replacement`: The replacement scan callback
- `extra_data`: Extra data that is passed back into the specified callback
- `delete_callback`: The delete callback to call on the extra data, if any

`duckdb_replacement_scan_set_function_name` Sets the replacement function name. If this function is called in the replacement callback, the replacement scan is performed. If it is not called, the replacement callback is not performed.

Syntax

```
            void duckdb_replacement_scan_set_function_name(
  duckdb_replacement_scan_info info,
  const char *function_name
);
```

Parameters

- `info`: The info object
- `function_name`: The function name to substitute.

`duckdb_replacement_scan_add_parameter` Adds a parameter to the replacement scan function.

Syntax

```
            void duckdb_replacement_scan_add_parameter(
  duckdb_replacement_scan_info info,
  duckdb_value parameter
);
```

Parameters

- `info`: The info object
- `parameter`: The parameter to add.

`duckdb_replacement_scan_set_error` Report that an error has occurred while executing the replacement scan.

Syntax

```
            void duckdb_replacement_scan_set_error(
  duckdb_replacement_scan_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

Complete API

This page contains the reference for DuckDB's C API.

> The reference contains several deprecation notices. These concern methods whose long-term availability is not guaranteed as they may be removed in the future. That said, DuckDB's developers plan to carry out deprecations slowly as several of the deprecated methods do not yet have a fully functional alternative. Therefore, they will not be removed before the alternative is available, and even then, there will be a grace period of a few minor versions before removing them. The reason that the methods are already deprecated in v1.0 is to denote that they are not part of the v1.0 stable API, which contains methods that are available long-term.

API Reference Overview

Open Connect

```
            duckdb_instance_cache duckdb_create_instance_cache();
duckdb_state duckdb_get_or_create_from_cache(duckdb_instance_cache instance_cache, const char *path,
duckdb_database *out_database, duckdb_config config, char **out_error);
void duckdb_destroy_instance_cache(duckdb_instance_cache *instance_cache);
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
duckdb_state duckdb_open_ext(const char *path, duckdb_database *out_database, duckdb_config config, char
**out_error);
void duckdb_close(duckdb_database *database);
duckdb_state duckdb_connect(duckdb_database database, duckdb_connection *out_connection);
void duckdb_interrupt(duckdb_connection connection);
duckdb_query_progress_type duckdb_query_progress(duckdb_connection connection);
void duckdb_disconnect(duckdb_connection *connection);
```

```
void duckdb_connection_get_client_context(duckdb_connection connection, duckdb_client_context *out_
context);
void duckdb_connection_get_arrow_options(duckdb_connection connection, duckdb_arrow_options *out_arrow_
options);
idx_t duckdb_client_context_get_connection_id(duckdb_client_context context);
void duckdb_destroy_client_context(duckdb_client_context *context);
void duckdb_destroy_arrow_options(duckdb_arrow_options *arrow_options);
const char *duckdb_library_version();
duckdb_value duckdb_get_table_names(duckdb_connection connection, const char *query, bool qualified);
```

Configuration

```
            duckdb_state duckdb_create_config(duckdb_config *out_config);
size_t duckdb_config_count();
duckdb_state duckdb_get_config_flag(size_t index, const char **out_name, const char **out_description);
duckdb_state duckdb_set_config(duckdb_config config, const char *name, const char *option);
void duckdb_destroy_config(duckdb_config *config);
```

Error Data

```
            duckdb_error_data duckdb_create_error_data(duckdb_error_type type, const char *message);
void duckdb_destroy_error_data(duckdb_error_data *error_data);
duckdb_error_type duckdb_error_data_error_type(duckdb_error_data error_data);
const char *duckdb_error_data_message(duckdb_error_data error_data);
bool duckdb_error_data_has_error(duckdb_error_data error_data);
```

Query Execution

```
            duckdb_state duckdb_query(duckdb_connection connection, const char *query, duckdb_result *out_
result);
void duckdb_destroy_result(duckdb_result *result);
const char *duckdb_column_name(duckdb_result *result, idx_t col);
duckdb_type duckdb_column_type(duckdb_result *result, idx_t col);
duckdb_statement_type duckdb_result_statement_type(duckdb_result result);
duckdb_logical_type duckdb_column_logical_type(duckdb_result *result, idx_t col);
duckdb_arrow_options duckdb_result_get_arrow_options(duckdb_result *result);
idx_t duckdb_column_count(duckdb_result *result);
idx_t duckdb_row_count(duckdb_result *result);
idx_t duckdb_rows_changed(duckdb_result *result);
void *duckdb_column_data(duckdb_result *result, idx_t col);
bool *duckdb_nullmask_data(duckdb_result *result, idx_t col);
const char *duckdb_result_error(duckdb_result *result);
duckdb_error_type duckdb_result_error_type(duckdb_result *result);
```

Result Functions

```
            duckdb_data_chunk duckdb_result_get_chunk(duckdb_result result, idx_t chunk_index);
bool duckdb_result_is_streaming(duckdb_result result);
idx_t duckdb_result_chunk_count(duckdb_result result);
duckdb_result_type duckdb_result_return_type(duckdb_result result);
```

Safe Fetch Functions

```
            bool duckdb_value_boolean(duckdb_result *result, idx_t col, idx_t row);
int8_t duckdb_value_int8(duckdb_result *result, idx_t col, idx_t row);
int16_t duckdb_value_int16(duckdb_result *result, idx_t col, idx_t row);
int32_t duckdb_value_int32(duckdb_result *result, idx_t col, idx_t row);
int64_t duckdb_value_int64(duckdb_result *result, idx_t col, idx_t row);
duckdb_hugeint duckdb_value_hugeint(duckdb_result *result, idx_t col, idx_t row);
```

```
duckdb_uhugeint duckdb_value_uhugeint(duckdb_result *result, idx_t col, idx_t row);
duckdb_decimal duckdb_value_decimal(duckdb_result *result, idx_t col, idx_t row);
uint8_t duckdb_value_uint8(duckdb_result *result, idx_t col, idx_t row);
uint16_t duckdb_value_uint16(duckdb_result *result, idx_t col, idx_t row);
uint32_t duckdb_value_uint32(duckdb_result *result, idx_t col, idx_t row);
uint64_t duckdb_value_uint64(duckdb_result *result, idx_t col, idx_t row);
float duckdb_value_float(duckdb_result *result, idx_t col, idx_t row);
double duckdb_value_double(duckdb_result *result, idx_t col, idx_t row);
duckdb_date duckdb_value_date(duckdb_result *result, idx_t col, idx_t row);
duckdb_time duckdb_value_time(duckdb_result *result, idx_t col, idx_t row);
duckdb_timestamp duckdb_value_timestamp(duckdb_result *result, idx_t col, idx_t row);
duckdb_interval duckdb_value_interval(duckdb_result *result, idx_t col, idx_t row);
char *duckdb_value_varchar(duckdb_result *result, idx_t col, idx_t row);
duckdb_string duckdb_value_string(duckdb_result *result, idx_t col, idx_t row);
char *duckdb_value_varchar_internal(duckdb_result *result, idx_t col, idx_t row);
duckdb_string duckdb_value_string_internal(duckdb_result *result, idx_t col, idx_t row);
duckdb_blob duckdb_value_blob(duckdb_result *result, idx_t col, idx_t row);
bool duckdb_value_is_null(duckdb_result *result, idx_t col, idx_t row);
```

Helpers

```
            void *duckdb_malloc(size_t size);
void duckdb_free(void *ptr);
idx_t duckdb_vector_size();
bool duckdb_string_is_inlined(duckdb_string_t string);
uint32_t duckdb_string_t_length(duckdb_string_t string);
const char *duckdb_string_t_data(duckdb_string_t *string);
```

Date Time Timestamp Helpers

```
            duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
bool duckdb_is_finite_date(duckdb_date date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time_tz duckdb_create_time_tz(int64_t micros, int32_t offset);
duckdb_time_tz_struct duckdb_from_time_tz(duckdb_time_tz micros);
duckdb_time duckdb_to_time(duckdb_time_struct time);
duckdb_timestamp_struct duckdb_from_timestamp(duckdb_timestamp ts);
duckdb_timestamp duckdb_to_timestamp(duckdb_timestamp_struct ts);
bool duckdb_is_finite_timestamp(duckdb_timestamp ts);
bool duckdb_is_finite_timestamp_s(duckdb_timestamp_s ts);
bool duckdb_is_finite_timestamp_ms(duckdb_timestamp_ms ts);
bool duckdb_is_finite_timestamp_ns(duckdb_timestamp_ns ts);
```

Hugeint Helpers

```
            double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
```

Unsigned Hugeint Helpers

```
            double duckdb_uhugeint_to_double(duckdb_uhugeint val);
duckdb_uhugeint duckdb_double_to_uhugeint(double val);
```

Decimal Helpers

```
            duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);
double duckdb_decimal_to_double(duckdb_decimal val);
```

Prepared Statements

```
duckdb_state duckdb_prepare(duckdb_connection connection, const char *query, duckdb_prepared_
statement *out_prepared_statement);
void duckdb_destroy_prepare(duckdb_prepared_statement *prepared_statement);
const char *duckdb_prepare_error(duckdb_prepared_statement prepared_statement);
idx_t duckdb_nparams(duckdb_prepared_statement prepared_statement);
const char *duckdb_parameter_name(duckdb_prepared_statement prepared_statement, idx_t index);
duckdb_type duckdb_param_type(duckdb_prepared_statement prepared_statement, idx_t param_idx);
duckdb_logical_type duckdb_param_logical_type(duckdb_prepared_statement prepared_statement, idx_t param_
idx);
duckdb_state duckdb_clear_bindings(duckdb_prepared_statement prepared_statement);
duckdb_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement statement);
idx_t duckdb_prepared_statement_column_count(duckdb_prepared_statement prepared_statement);
const char *duckdb_prepared_statement_column_name(duckdb_prepared_statement prepared_statement, idx_t col_
idx);
duckdb_logical_type duckdb_prepared_statement_column_logical_type(duckdb_prepared_statement prepared_
statement, idx_t col_idx);
duckdb_type duckdb_prepared_statement_column_type(duckdb_prepared_statement prepared_statement, idx_t col_
idx);
```

Bind Values to Prepared Statements

```
duckdb_state duckdb_bind_value(duckdb_prepared_statement prepared_statement, idx_t param_idx,
duckdb_value val);
duckdb_state duckdb_bind_parameter_index(duckdb_prepared_statement prepared_statement, idx_t *param_idx_
out, const char *name);
duckdb_state duckdb_bind_boolean(duckdb_prepared_statement prepared_statement, idx_t param_idx, bool val);
duckdb_state duckdb_bind_int8(duckdb_prepared_statement prepared_statement, idx_t param_idx, int8_t val);
duckdb_state duckdb_bind_int16(duckdb_prepared_statement prepared_statement, idx_t param_idx, int16_t val);
duckdb_state duckdb_bind_int32(duckdb_prepared_statement prepared_statement, idx_t param_idx, int32_t val);
duckdb_state duckdb_bind_int64(duckdb_prepared_statement prepared_statement, idx_t param_idx, int64_t val);
duckdb_state duckdb_bind_hugeint(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
hugeint val);
duckdb_state duckdb_bind_uhugeint(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
uhugeint val);
duckdb_state duckdb_bind_decimal(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
decimal val);
duckdb_state duckdb_bind_uint8(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint8_t val);
duckdb_state duckdb_bind_uint16(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint16_t
val);
duckdb_state duckdb_bind_uint32(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint32_t
val);
duckdb_state duckdb_bind_uint64(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint64_t
val);
duckdb_state duckdb_bind_float(duckdb_prepared_statement prepared_statement, idx_t param_idx, float val);
duckdb_state duckdb_bind_double(duckdb_prepared_statement prepared_statement, idx_t param_idx, double val);
duckdb_state duckdb_bind_date(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_date
val);
duckdb_state duckdb_bind_time(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_time
val);
duckdb_state duckdb_bind_timestamp(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
timestamp val);
duckdb_state duckdb_bind_timestamp_tz(duckdb_prepared_statement prepared_statement, idx_t param_idx,
duckdb_timestamp val);
duckdb_state duckdb_bind_interval(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
interval val);
duckdb_state duckdb_bind_varchar(duckdb_prepared_statement prepared_statement, idx_t param_idx, const char
*val);
duckdb_state duckdb_bind_varchar_length(duckdb_prepared_statement prepared_statement, idx_t param_idx,
const char *val, idx_t length);
duckdb_state duckdb_bind_blob(duckdb_prepared_statement prepared_statement, idx_t param_idx, const void
*data, idx_t length);
duckdb_state duckdb_bind_null(duckdb_prepared_statement prepared_statement, idx_t param_idx);
```

Execute Prepared Statements

```
            duckdb_state duckdb_execute_prepared(duckdb_prepared_statement prepared_statement, duckdb_
result *out_result);
duckdb_state duckdb_execute_prepared_streaming(duckdb_prepared_statement prepared_statement, duckdb_result
*out_result);
```

Extract Statements

```
            idx_t duckdb_extract_statements(duckdb_connection connection, const char *query, duckdb_
extracted_statements *out_extracted_statements);
duckdb_state duckdb_prepare_extracted_statement(duckdb_connection connection, duckdb_extracted_statements
extracted_statements, idx_t index, duckdb_prepared_statement *out_prepared_statement);
const char *duckdb_extract_statements_error(duckdb_extracted_statements extracted_statements);
void duckdb_destroy_extracted(duckdb_extracted_statements *extracted_statements);
```

Pending Result Interface

```
            duckdb_state duckdb_pending_prepared(duckdb_prepared_statement prepared_statement, duckdb_
pending_result *out_result);
duckdb_state duckdb_pending_prepared_streaming(duckdb_prepared_statement prepared_statement, duckdb_
pending_result *out_result);
void duckdb_destroy_pending(duckdb_pending_result *pending_result);
const char *duckdb_pending_error(duckdb_pending_result pending_result);
duckdb_pending_state duckdb_pending_execute_task(duckdb_pending_result pending_result);
duckdb_pending_state duckdb_pending_execute_check_state(duckdb_pending_result pending_result);
duckdb_state duckdb_execute_pending(duckdb_pending_result pending_result, duckdb_result *out_result);
bool duckdb_pending_execution_is_finished(duckdb_pending_state pending_state);
```

Value Interface

```
            void duckdb_destroy_value(duckdb_value *value);
duckdb_value duckdb_create_varchar(const char *text);
duckdb_value duckdb_create_varchar_length(const char *text, idx_t length);
duckdb_value duckdb_create_bool(bool input);
duckdb_value duckdb_create_int8(int8_t input);
duckdb_value duckdb_create_uint8(uint8_t input);
duckdb_value duckdb_create_int16(int16_t input);
duckdb_value duckdb_create_uint16(uint16_t input);
duckdb_value duckdb_create_int32(int32_t input);
duckdb_value duckdb_create_uint32(uint32_t input);
duckdb_value duckdb_create_uint64(uint64_t input);
duckdb_value duckdb_create_int64(int64_t val);
duckdb_value duckdb_create_hugeint(duckdb_hugeint input);
duckdb_value duckdb_create_uhugeint(duckdb_uhugeint input);
duckdb_value duckdb_create_bignum(duckdb_bignum input);
duckdb_value duckdb_create_decimal(duckdb_decimal input);
duckdb_value duckdb_create_float(float input);
duckdb_value duckdb_create_double(double input);
duckdb_value duckdb_create_date(duckdb_date input);
duckdb_value duckdb_create_time(duckdb_time input);
duckdb_value duckdb_create_time_ns(duckdb_time_ns input);
duckdb_value duckdb_create_time_tz_value(duckdb_time_tz value);
duckdb_value duckdb_create_timestamp(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_tz(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_s(duckdb_timestamp_s input);
duckdb_value duckdb_create_timestamp_ms(duckdb_timestamp_ms input);
duckdb_value duckdb_create_timestamp_ns(duckdb_timestamp_ns input);
duckdb_value duckdb_create_interval(duckdb_interval input);
duckdb_value duckdb_create_blob(const uint8_t *data, idx_t length);
duckdb_value duckdb_create_bit(duckdb_bit input);
```

```
duckdb_value duckdb_create_uuid(duckdb_uhugeint input);
bool duckdb_get_bool(duckdb_value val);
int8_t duckdb_get_int8(duckdb_value val);
uint8_t duckdb_get_uint8(duckdb_value val);
int16_t duckdb_get_int16(duckdb_value val);
uint16_t duckdb_get_uint16(duckdb_value val);
int32_t duckdb_get_int32(duckdb_value val);
uint32_t duckdb_get_uint32(duckdb_value val);
int64_t duckdb_get_int64(duckdb_value val);
uint64_t duckdb_get_uint64(duckdb_value val);
duckdb_hugeint duckdb_get_hugeint(duckdb_value val);
duckdb_uhugeint duckdb_get_uhugeint(duckdb_value val);
duckdb_bignum duckdb_get_bignum(duckdb_value val);
duckdb_decimal duckdb_get_decimal(duckdb_value val);
float duckdb_get_float(duckdb_value val);
double duckdb_get_double(duckdb_value val);
duckdb_date duckdb_get_date(duckdb_value val);
duckdb_time duckdb_get_time(duckdb_value val);
duckdb_time_ns duckdb_get_time_ns(duckdb_value val);
duckdb_time_tz duckdb_get_time_tz(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp_tz(duckdb_value val);
duckdb_timestamp_s duckdb_get_timestamp_s(duckdb_value val);
duckdb_timestamp_ms duckdb_get_timestamp_ms(duckdb_value val);
duckdb_timestamp_ns duckdb_get_timestamp_ns(duckdb_value val);
duckdb_interval duckdb_get_interval(duckdb_value val);
duckdb_logical_type duckdb_get_value_type(duckdb_value val);
duckdb_blob duckdb_get_blob(duckdb_value val);
duckdb_bit duckdb_get_bit(duckdb_value val);
duckdb_uhugeint duckdb_get_uuid(duckdb_value val);
char *duckdb_get_varchar(duckdb_value value);
duckdb_value duckdb_create_struct_value(duckdb_logical_type type, duckdb_value *values);
duckdb_value duckdb_create_list_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
duckdb_value duckdb_create_array_value(duckdb_logical_type type, duckdb_value *values, idx_t value_count);
duckdb_value duckdb_create_map_value(duckdb_logical_type map_type, duckdb_value *keys, duckdb_value
*values, idx_t entry_count);
duckdb_value duckdb_create_union_value(duckdb_logical_type union_type, idx_t tag_index, duckdb_value
value);
idx_t duckdb_get_map_size(duckdb_value value);
duckdb_value duckdb_get_map_key(duckdb_value value, idx_t index);
duckdb_value duckdb_get_map_value(duckdb_value value, idx_t index);
bool duckdb_is_null_value(duckdb_value value);
duckdb_value duckdb_create_null_value();
idx_t duckdb_get_list_size(duckdb_value value);
duckdb_value duckdb_get_list_child(duckdb_value value, idx_t index);
duckdb_value duckdb_create_enum_value(duckdb_logical_type type, uint64_t value);
uint64_t duckdb_get_enum_value(duckdb_value value);
duckdb_value duckdb_get_struct_child(duckdb_value value, idx_t index);
char *duckdb_value_to_string(duckdb_value value);
```

Logical Type Interface

```
            duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
char *duckdb_logical_type_get_alias(duckdb_logical_type type);
void duckdb_logical_type_set_alias(duckdb_logical_type type, const char *alias);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_array_type(duckdb_logical_type type, idx_t array_size);
duckdb_logical_type duckdb_create_map_type(duckdb_logical_type key_type, duckdb_logical_type value_type);
duckdb_logical_type duckdb_create_union_type(duckdb_logical_type *member_types, const char **member_names,
idx_t member_count);
duckdb_logical_type duckdb_create_struct_type(duckdb_logical_type *member_types, const char **member_
names, idx_t member_count);
duckdb_logical_type duckdb_create_enum_type(const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_decimal_type(uint8_t width, uint8_t scale);
```

```
duckdb_type duckdb_get_type_id(duckdb_logical_type type);
uint8_t duckdb_decimal_width(duckdb_logical_type type);
uint8_t duckdb_decimal_scale(duckdb_logical_type type);
duckdb_type duckdb_decimal_internal_type(duckdb_logical_type type);
duckdb_type duckdb_enum_internal_type(duckdb_logical_type type);
uint32_t duckdb_enum_dictionary_size(duckdb_logical_type type);
char *duckdb_enum_dictionary_value(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_array_type_child_type(duckdb_logical_type type);
idx_t duckdb_array_type_array_size(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_value_type(duckdb_logical_type type);
idx_t duckdb_struct_type_child_count(duckdb_logical_type type);
char *duckdb_struct_type_child_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_struct_type_child_type(duckdb_logical_type type, idx_t index);
idx_t duckdb_union_type_member_count(duckdb_logical_type type);
char *duckdb_union_type_member_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_union_type_member_type(duckdb_logical_type type, idx_t index);
void duckdb_destroy_logical_type(duckdb_logical_type *type);
duckdb_state duckdb_register_logical_type(duckdb_connection con, duckdb_logical_type type, duckdb_create_
type_info info);
```

Data Chunk Interface

```
            duckdb_data_chunk duckdb_create_data_chunk(duckdb_logical_type *types, idx_t column_count);
void duckdb_destroy_data_chunk(duckdb_data_chunk *chunk);
void duckdb_data_chunk_reset(duckdb_data_chunk chunk);
idx_t duckdb_data_chunk_get_column_count(duckdb_data_chunk chunk);
duckdb_vector duckdb_data_chunk_get_vector(duckdb_data_chunk chunk, idx_t col_idx);
idx_t duckdb_data_chunk_get_size(duckdb_data_chunk chunk);
void duckdb_data_chunk_set_size(duckdb_data_chunk chunk, idx_t size);
```

Vector Interface

```
            duckdb_vector duckdb_create_vector(duckdb_logical_type type, idx_t capacity);
void duckdb_destroy_vector(duckdb_vector *vector);
duckdb_logical_type duckdb_vector_get_column_type(duckdb_vector vector);
void *duckdb_vector_get_data(duckdb_vector vector);
uint64_t *duckdb_vector_get_validity(duckdb_vector vector);
void duckdb_vector_ensure_validity_writable(duckdb_vector vector);
void duckdb_vector_assign_string_element(duckdb_vector vector, idx_t index, const char *str);
void duckdb_vector_assign_string_element_len(duckdb_vector vector, idx_t index, const char *str, idx_t str_
len);
duckdb_vector duckdb_list_vector_get_child(duckdb_vector vector);
idx_t duckdb_list_vector_get_size(duckdb_vector vector);
duckdb_state duckdb_list_vector_set_size(duckdb_vector vector, idx_t size);
duckdb_state duckdb_list_vector_reserve(duckdb_vector vector, idx_t required_capacity);
duckdb_vector duckdb_struct_vector_get_child(duckdb_vector vector, idx_t index);
duckdb_vector duckdb_array_vector_get_child(duckdb_vector vector);
void duckdb_slice_vector(duckdb_vector vector, duckdb_selection_vector sel, idx_t len);
void duckdb_vector_copy_sel(duckdb_vector src, duckdb_vector dst, duckdb_selection_vector sel, idx_t src_
count, idx_t src_offset, idx_t dst_offset);
void duckdb_vector_reference_value(duckdb_vector vector, duckdb_value value);
void duckdb_vector_reference_vector(duckdb_vector to_vector, duckdb_vector from_vector);
```

Validity Mask Functions

```
            bool duckdb_validity_row_is_valid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_validity(uint64_t *validity, idx_t row, bool valid);
void duckdb_validity_set_row_invalid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_valid(uint64_t *validity, idx_t row);
```

Scalar Functions

```
            duckdb_scalar_function duckdb_create_scalar_function();
void duckdb_destroy_scalar_function(duckdb_scalar_function *scalar_function);
void duckdb_scalar_function_set_name(duckdb_scalar_function scalar_function, const char *name);
void duckdb_scalar_function_set_varargs(duckdb_scalar_function scalar_function, duckdb_logical_type type);
void duckdb_scalar_function_set_special_handling(duckdb_scalar_function scalar_function);
void duckdb_scalar_function_set_volatile(duckdb_scalar_function scalar_function);
void duckdb_scalar_function_add_parameter(duckdb_scalar_function scalar_function, duckdb_logical_type
type);
void duckdb_scalar_function_set_return_type(duckdb_scalar_function scalar_function, duckdb_logical_type
type);
void duckdb_scalar_function_set_extra_info(duckdb_scalar_function scalar_function, void *extra_info,
duckdb_delete_callback_t destroy);
void duckdb_scalar_function_set_bind(duckdb_scalar_function scalar_function, duckdb_scalar_function_bind_t
bind);
void duckdb_scalar_function_set_bind_data(duckdb_bind_info info, void *bind_data, duckdb_delete_callback_t
destroy);
void duckdb_scalar_function_set_bind_data_copy(duckdb_bind_info info, duckdb_copy_callback_t copy);
void duckdb_scalar_function_bind_set_error(duckdb_bind_info info, const char *error);
void duckdb_scalar_function_set_function(duckdb_scalar_function scalar_function, duckdb_scalar_function_t
function);
duckdb_state duckdb_register_scalar_function(duckdb_connection con, duckdb_scalar_function scalar_
function);
void *duckdb_scalar_function_get_extra_info(duckdb_function_info info);
void *duckdb_scalar_function_bind_get_extra_info(duckdb_bind_info info);
void *duckdb_scalar_function_get_bind_data(duckdb_function_info info);
void duckdb_scalar_function_get_client_context(duckdb_bind_info info, duckdb_client_context *out_context);
void duckdb_scalar_function_set_error(duckdb_function_info info, const char *error);
duckdb_scalar_function_set duckdb_create_scalar_function_set(const char *name);
void duckdb_destroy_scalar_function_set(duckdb_scalar_function_set *scalar_function_set);
duckdb_state duckdb_add_scalar_function_to_set(duckdb_scalar_function_set set, duckdb_scalar_function
function);
duckdb_state duckdb_register_scalar_function_set(duckdb_connection con, duckdb_scalar_function_set set);
idx_t duckdb_scalar_function_bind_get_argument_count(duckdb_bind_info info);
duckdb_expression duckdb_scalar_function_bind_get_argument(duckdb_bind_info info, idx_t index);
```

Selection Vector Interface

```
            duckdb_selection_vector duckdb_create_selection_vector(idx_t size);
void duckdb_destroy_selection_vector(duckdb_selection_vector sel);
sel_t *duckdb_selection_vector_get_data_ptr(duckdb_selection_vector sel);
```

Aggregate Functions

```
            duckdb_aggregate_function duckdb_create_aggregate_function();
void duckdb_destroy_aggregate_function(duckdb_aggregate_function *aggregate_function);
void duckdb_aggregate_function_set_name(duckdb_aggregate_function aggregate_function, const char *name);
void duckdb_aggregate_function_add_parameter(duckdb_aggregate_function aggregate_function, duckdb_logical_
type type);
void duckdb_aggregate_function_set_return_type(duckdb_aggregate_function aggregate_function, duckdb_
logical_type type);
void duckdb_aggregate_function_set_functions(duckdb_aggregate_function aggregate_function, duckdb_
aggregate_state_size state_size, duckdb_aggregate_init_t state_init, duckdb_aggregate_update_t update,
duckdb_aggregate_combine_t combine, duckdb_aggregate_finalize_t finalize);
void duckdb_aggregate_function_set_destructor(duckdb_aggregate_function aggregate_function, duckdb_
aggregate_destroy_t destroy);
duckdb_state duckdb_register_aggregate_function(duckdb_connection con, duckdb_aggregate_function
aggregate_function);
void duckdb_aggregate_function_set_special_handling(duckdb_aggregate_function aggregate_function);
void duckdb_aggregate_function_set_extra_info(duckdb_aggregate_function aggregate_function, void *extra_
info, duckdb_delete_callback_t destroy);
```

```
void *duckdb_aggregate_function_get_extra_info(duckdb_function_info info);
void duckdb_aggregate_function_set_error(duckdb_function_info info, const char *error);
duckdb_aggregate_function_set duckdb_create_aggregate_function_set(const char *name);
void duckdb_destroy_aggregate_function_set(duckdb_aggregate_function_set *aggregate_function_set);
duckdb_state duckdb_add_aggregate_function_to_set(duckdb_aggregate_function_set set, duckdb_aggregate_
function function);
duckdb_state duckdb_register_aggregate_function_set(duckdb_connection con, duckdb_aggregate_function_set
set);
```

Table Functions

```
            duckdb_table_function duckdb_create_table_function();
void duckdb_destroy_table_function(duckdb_table_function *table_function);
void duckdb_table_function_set_name(duckdb_table_function table_function, const char *name);
void duckdb_table_function_add_parameter(duckdb_table_function table_function, duckdb_logical_type type);
void duckdb_table_function_add_named_parameter(duckdb_table_function table_function, const char *name,
duckdb_logical_type type);
void duckdb_table_function_set_extra_info(duckdb_table_function table_function, void *extra_info, duckdb_
delete_callback_t destroy);
void duckdb_table_function_set_bind(duckdb_table_function table_function, duckdb_table_function_bind_t
bind);
void duckdb_table_function_set_init(duckdb_table_function table_function, duckdb_table_function_init_t
init);
void duckdb_table_function_set_local_init(duckdb_table_function table_function, duckdb_table_function_
init_t init);
void duckdb_table_function_set_function(duckdb_table_function table_function, duckdb_table_function_t
function);
void duckdb_table_function_supports_projection_pushdown(duckdb_table_function table_function, bool
pushdown);
duckdb_state duckdb_register_table_function(duckdb_connection con, duckdb_table_function function);
```

Table Function Bind

```
            void *duckdb_bind_get_extra_info(duckdb_bind_info info);
void duckdb_table_function_get_client_context(duckdb_bind_info info, duckdb_client_context *out_context);
void duckdb_bind_add_result_column(duckdb_bind_info info, const char *name, duckdb_logical_type type);
idx_t duckdb_bind_get_parameter_count(duckdb_bind_info info);
duckdb_value duckdb_bind_get_parameter(duckdb_bind_info info, idx_t index);
duckdb_value duckdb_bind_get_named_parameter(duckdb_bind_info info, const char *name);
void duckdb_bind_set_bind_data(duckdb_bind_info info, void *bind_data, duckdb_delete_callback_t destroy);
void duckdb_bind_set_cardinality(duckdb_bind_info info, idx_t cardinality, bool is_exact);
void duckdb_bind_set_error(duckdb_bind_info info, const char *error);
```

Table Function Init

```
            void *duckdb_init_get_extra_info(duckdb_init_info info);
void *duckdb_init_get_bind_data(duckdb_init_info info);
void duckdb_init_set_init_data(duckdb_init_info info, void *init_data, duckdb_delete_callback_t destroy);
idx_t duckdb_init_get_column_count(duckdb_init_info info);
idx_t duckdb_init_get_column_index(duckdb_init_info info, idx_t column_index);
void duckdb_init_set_max_threads(duckdb_init_info info, idx_t max_threads);
void duckdb_init_set_error(duckdb_init_info info, const char *error);
```

Table Function

```
            void *duckdb_function_get_extra_info(duckdb_function_info info);
void *duckdb_function_get_bind_data(duckdb_function_info info);
void *duckdb_function_get_init_data(duckdb_function_info info);
void *duckdb_function_get_local_init_data(duckdb_function_info info);
void duckdb_function_set_error(duckdb_function_info info, const char *error);
```

Replacement Scans

```
            void duckdb_add_replacement_scan(duckdb_database db, duckdb_replacement_callback_t replacement,
            void *extra_data, duckdb_delete_callback_t delete_callback);
void duckdb_replacement_scan_set_function_name(duckdb_replacement_scan_info info, const char *function_
name);
void duckdb_replacement_scan_add_parameter(duckdb_replacement_scan_info info, duckdb_value parameter);
void duckdb_replacement_scan_set_error(duckdb_replacement_scan_info info, const char *error);
```

Profiling Info

```
            duckdb_profiling_info duckdb_get_profiling_info(duckdb_connection connection);
duckdb_value duckdb_profiling_info_get_value(duckdb_profiling_info info, const char *key);
duckdb_value duckdb_profiling_info_get_metrics(duckdb_profiling_info info);
idx_t duckdb_profiling_info_get_child_count(duckdb_profiling_info info);
duckdb_profiling_info duckdb_profiling_info_get_child(duckdb_profiling_info info, idx_t index);
```

Appender

```
            duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const
char *table, duckdb_appender *out_appender);
duckdb_state duckdb_appender_create_ext(duckdb_connection connection, const char *catalog, const char
*schema, const char *table, duckdb_appender *out_appender);
duckdb_state duckdb_appender_create_query(duckdb_connection connection, const char *query, idx_t column_
count, duckdb_logical_type *types, const char *table_name, const char **column_names, duckdb_appender *out_
appender);
idx_t duckdb_appender_column_count(duckdb_appender appender);
duckdb_logical_type duckdb_appender_column_type(duckdb_appender appender, idx_t col_idx);
const char *duckdb_appender_error(duckdb_appender appender);
duckdb_error_data duckdb_appender_error_data(duckdb_appender appender);
duckdb_state duckdb_appender_flush(duckdb_appender appender);
duckdb_state duckdb_appender_close(duckdb_appender appender);
duckdb_state duckdb_appender_destroy(duckdb_appender *appender);
duckdb_state duckdb_appender_add_column(duckdb_appender appender, const char *name);
duckdb_state duckdb_appender_clear_columns(duckdb_appender appender);
duckdb_state duckdb_appender_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_append_default(duckdb_appender appender);
duckdb_state duckdb_append_default_to_chunk(duckdb_appender appender, duckdb_data_chunk chunk, idx_t col,
idx_t row);
duckdb_state duckdb_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_append_int32(duckdb_appender appender, int32_t value);
duckdb_state duckdb_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_append_hugeint(duckdb_appender appender, duckdb_hugeint value);
duckdb_state duckdb_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_append_uhugeint(duckdb_appender appender, duckdb_uhugeint value);
duckdb_state duckdb_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_append_date(duckdb_appender appender, duckdb_date value);
duckdb_state duckdb_append_time(duckdb_appender appender, duckdb_time value);
duckdb_state duckdb_append_timestamp(duckdb_appender appender, duckdb_timestamp value);
duckdb_state duckdb_append_interval(duckdb_appender appender, duckdb_interval value);
duckdb_state duckdb_append_varchar(duckdb_appender appender, const char *val);
duckdb_state duckdb_append_varchar_length(duckdb_appender appender, const char *val, idx_t length);
duckdb_state duckdb_append_blob(duckdb_appender appender, const void *data, idx_t length);
duckdb_state duckdb_append_null(duckdb_appender appender);
duckdb_state duckdb_append_value(duckdb_appender appender, duckdb_value value);
duckdb_state duckdb_append_data_chunk(duckdb_appender appender, duckdb_data_chunk chunk);
```

Table Description

```
duckdb_state duckdb_table_description_create(duckdb_connection connection, const char *schema,
const char *table, duckdb_table_description *out);
duckdb_state duckdb_table_description_create_ext(duckdb_connection connection, const char *catalog, const
char *schema, const char *table, duckdb_table_description *out);
void duckdb_table_description_destroy(duckdb_table_description *table_description);
const char *duckdb_table_description_error(duckdb_table_description table_description);
duckdb_state duckdb_column_has_default(duckdb_table_description table_description, idx_t index, bool *out);
char *duckdb_table_description_get_column_name(duckdb_table_description table_description, idx_t index);
```

Arrow Interface

```
duckdb_error_data duckdb_to_arrow_schema(duckdb_arrow_options arrow_options, duckdb_logical_
type *types, const char **names, idx_t column_count, struct ArrowSchema *out_schema);
duckdb_error_data duckdb_data_chunk_to_arrow(duckdb_arrow_options arrow_options, duckdb_data_chunk chunk,
struct ArrowArray *out_arrow_array);
duckdb_error_data duckdb_schema_from_arrow(duckdb_connection connection, struct ArrowSchema *schema,
duckdb_arrow_converted_schema *out_types);
duckdb_error_data duckdb_data_chunk_from_arrow(duckdb_connection connection, struct ArrowArray *arrow_
array, duckdb_arrow_converted_schema converted_schema, duckdb_data_chunk *out_chunk);
void duckdb_destroy_arrow_converted_schema(duckdb_arrow_converted_schema *arrow_converted_schema);
duckdb_state duckdb_query_arrow(duckdb_connection connection, const char *query, duckdb_arrow *out_result);
duckdb_state duckdb_query_arrow_schema(duckdb_arrow result, duckdb_arrow_schema *out_schema);
duckdb_state duckdb_prepared_arrow_schema(duckdb_prepared_statement prepared, duckdb_arrow_schema *out_
schema);
void duckdb_result_arrow_array(duckdb_result result, duckdb_data_chunk chunk, duckdb_arrow_array *out_
array);
duckdb_state duckdb_query_arrow_array(duckdb_arrow result, duckdb_arrow_array *out_array);
idx_t duckdb_arrow_column_count(duckdb_arrow result);
idx_t duckdb_arrow_row_count(duckdb_arrow result);
idx_t duckdb_arrow_rows_changed(duckdb_arrow result);
const char *duckdb_query_arrow_error(duckdb_arrow result);
void duckdb_destroy_arrow(duckdb_arrow *result);
void duckdb_destroy_arrow_stream(duckdb_arrow_stream *stream_p);
duckdb_state duckdb_execute_prepared_arrow(duckdb_prepared_statement prepared_statement, duckdb_arrow
*out_result);
duckdb_state duckdb_arrow_scan(duckdb_connection connection, const char *table_name, duckdb_arrow_stream
arrow);
duckdb_state duckdb_arrow_array_scan(duckdb_connection connection, const char *table_name, duckdb_arrow_
schema arrow_schema, duckdb_arrow_array arrow_array, duckdb_arrow_stream *out_stream);
```

Threading Information

```
void duckdb_execute_tasks(duckdb_database database, idx_t max_tasks);
duckdb_task_state duckdb_create_task_state(duckdb_database database);
void duckdb_execute_tasks_state(duckdb_task_state state);
idx_t duckdb_execute_n_tasks_state(duckdb_task_state state, idx_t max_tasks);
void duckdb_finish_execution(duckdb_task_state state);
bool duckdb_task_state_is_finished(duckdb_task_state state);
void duckdb_destroy_task_state(duckdb_task_state state);
bool duckdb_execution_is_finished(duckdb_connection con);
```

Streaming Result Interface

```
duckdb_data_chunk duckdb_stream_fetch_chunk(duckdb_result result);
duckdb_data_chunk duckdb_fetch_chunk(duckdb_result result);
```

Cast Functions

```
            duckdb_cast_function duckdb_create_cast_function();
void duckdb_cast_function_set_source_type(duckdb_cast_function cast_function, duckdb_logical_type source_
type);
void duckdb_cast_function_set_target_type(duckdb_cast_function cast_function, duckdb_logical_type target_
type);
void duckdb_cast_function_set_implicit_cast_cost(duckdb_cast_function cast_function, int64_t cost);
void duckdb_cast_function_set_function(duckdb_cast_function cast_function, duckdb_cast_function_t
function);
void duckdb_cast_function_set_extra_info(duckdb_cast_function cast_function, void *extra_info, duckdb_
delete_callback_t destroy);
void *duckdb_cast_function_get_extra_info(duckdb_function_info info);
duckdb_cast_mode duckdb_cast_function_get_cast_mode(duckdb_function_info info);
void duckdb_cast_function_set_error(duckdb_function_info info, const char *error);
void duckdb_cast_function_set_row_error(duckdb_function_info info, const char *error, idx_t row, duckdb_
vector output);
duckdb_state duckdb_register_cast_function(duckdb_connection con, duckdb_cast_function cast_function);
void duckdb_destroy_cast_function(duckdb_cast_function *cast_function);
```

Expression Interface

```
            void duckdb_destroy_expression(duckdb_expression *expr);
duckdb_logical_type duckdb_expression_return_type(duckdb_expression expr);
bool duckdb_expression_is_foldable(duckdb_expression expr);
duckdb_error_data duckdb_expression_fold(duckdb_client_context context, duckdb_expression expr, duckdb_
value *out_value);
```

`duckdb_create_instance_cache` Creates a new database instance cache. The instance cache is necessary if a client/program (re)opens multiple databases to the same file within the same process. Must be destroyed with 'duckdb_destroy_instance_cache'.

Return Value

The database instance cache.

Syntax

```
            duckdb_instance_cache duckdb_create_instance_cache(

);
```

`duckdb_get_or_create_from_cache` Creates a new database instance in the instance cache, or retrieves an existing database instance. Must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_get_or_create_from_cache(
  duckdb_instance_cache instance_cache,
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `instance_cache`: The instance cache in which to create the database, or from which to take the database.
- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open or retrieve an in-memory database.
- `out_database`: The resulting cached database.

- `config`: (Optional) configuration used to create the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_instance_cache` Destroys an existing database instance cache and de-allocates its memory.

Syntax

```
            void duckdb_destroy_instance_cache(
  duckdb_instance_cache *instance_cache
);
```

Parameters

- `instance_cache`: The instance cache to destroy.

`duckdb_open` Creates a new database or opens an existing database file stored at the given path. If no path is given a new in-memory database is created instead. The database must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_open(
  const char *path,
  duckdb_database *out_database
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_open_ext` Extended version of duckdb_open. Creates a new database or opens an existing database file stored at the given path. The database must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_open_ext(
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.
- `config`: (Optional) configuration used to start up the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value DuckDBSuccess on success or DuckDBError on failure.

duckdb_close Closes the specified database and de-allocates all memory allocated for that database. This should be called after you are done with any database allocated through duckdb_open or duckdb_open_ext. Note that failing to call duckdb_close (in case of e.g., a program crash) will not cause data corruption. Still, it is recommended to always correctly close a database object after you are done with it.

Syntax

```
            void duckdb_close(
  duckdb_database *database
);
```

Parameters

- database: The database object to shut down.

duckdb_connect Opens a connection to a database. Connections are required to query the database, and store transactional state associated with the connection. The instantiated connection should be closed using 'duckdb_disconnect'.

Syntax

```
            duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```

Parameters

- database: The database file to connect to.
- out_connection: The result connection object.

Return Value DuckDBSuccess on success or DuckDBError on failure.

duckdb_interrupt Interrupt running query

Syntax

```
            void duckdb_interrupt(
  duckdb_connection connection
);
```

Parameters

- connection: The connection to interrupt

duckdb_query_progress Get progress of the running query

Syntax

```
            duckdb_query_progress_type duckdb_query_progress(
  duckdb_connection connection
);
```

Parameters

- `connection`: The working connection

Return Value -1 if no progress or a percentage of the progress

`duckdb_disconnect`

Closes the specified connection and de-allocates all memory allocated for that connection.

Syntax

```
            void duckdb_disconnect(
  duckdb_connection *connection
);
```

Parameters

- `connection`: The connection to close.

`duckdb_connection_get_client_context`

Retrieves the client context of the connection.

Syntax

```
            void duckdb_connection_get_client_context(
  duckdb_connection connection,
  duckdb_client_context *out_context
);
```

Parameters

- `connection`: The connection.
- `out_context`: The client context of the connection. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_connection_get_arrow_options`

Retrieves the arrow options of the connection.

Syntax

```
            void duckdb_connection_get_arrow_options(
  duckdb_connection connection,
  duckdb_arrow_options *out_arrow_options
);
```

Parameters

- `connection`: The connection.

`duckdb_client_context_get_connection_id`

Returns the connection id of the client context.

Syntax

```
            idx_t duckdb_client_context_get_connection_id(
  duckdb_client_context context
);
```

Parameters

- `context`: The client context.

Return Value The connection id of the client context.

`duckdb_destroy_client_context`

Destroys the client context and deallocates its memory.

Syntax

```
            void duckdb_destroy_client_context(
  duckdb_client_context *context
);
```

Parameters

- `context`: The client context to destroy.

`duckdb_destroy_arrow_options`

Destroys the arrow options and deallocates its memory.

Syntax

```
            void duckdb_destroy_arrow_options(
  duckdb_arrow_options *arrow_options
);
```

Parameters

- `arrow_options`: The arrow options to destroy.

`duckdb_library_version`

Returns the version of the linked DuckDB, with a version postfix for dev versions

Usually used for developing C extensions that must return this for a compatibility check.

Syntax

```
            const char *duckdb_library_version(

);
```

`duckdb_get_table_names`

Get the list of (fully qualified) table names of the query.

Syntax

```
            duckdb_value duckdb_get_table_names(
  duckdb_connection connection,
  const char *query,
  bool qualified
);
```

Parameters

- `connection`: The connection for which to get the table names.
- `query`: The query for which to get the table names.
- `qualified`: Returns fully qualified table names (catalog.schema.table), if set to true, else only the (not escaped) table names.

Return Value A duckdb_value of type VARCHAR[] containing the (fully qualified) table names of the query. Must be destroyed with duckdb_destroy_value.

`duckdb_create_config` Initializes an empty configuration object that can be used to provide start-up options for the DuckDB instance through `duckdb_open_ext`. The duckdb_config must be destroyed using 'duckdb_destroy_config'

This will always succeed unless there is a malloc failure.

Note that `duckdb_destroy_config` should always be called on the resulting config, even if the function returns `DuckDBError`.

Syntax

```
            duckdb_state duckdb_create_config(
  duckdb_config *out_config
);
```

Parameters

- `out_config`: The result configuration object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_config_count` This returns the total amount of configuration options available for usage with `duckdb_get_config_flag`.

This should not be called in a loop as it internally loops over all the options.

Return Value The amount of config options available.

Syntax

```
            size_t duckdb_config_count(

);
```

`duckdb_get_config_flag` Obtains a human-readable name and description of a specific configuration option. This can be used to e.g. display configuration options. This will succeed unless `index` is out of range (i.e., `>= duckdb_config_count`).

The result name or description MUST NOT be freed.

Syntax

```
            duckdb_state duckdb_get_config_flag(
  size_t index,
  const char **out_name,
  const char **out_description
);
```

Parameters

- `index`: The index of the configuration option (between 0 and `duckdb_config_count`)
- `out_name`: A name of the configuration flag.
- `out_description`: A description of the configuration flag.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_set_config` Sets the specified option for the specified configuration. The configuration option is indicated by name. To obtain a list of config options, see `duckdb_get_config_flag`.

In the source code, configuration options are defined in `config.cpp`.

This can fail if either the name is invalid, or if the value provided for the option is invalid.

Syntax

```
            duckdb_state duckdb_set_config(
  duckdb_config config,
  const char *name,
  const char *option
);
```

Parameters

- `config`: The configuration object to set the option on.
- `name`: The name of the configuration flag to set.
- `option`: The value to set the configuration flag to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_config` Destroys the specified configuration object and de-allocates all memory allocated for the object.

Syntax

```
            void duckdb_destroy_config(
  duckdb_config *config
);
```

Parameters

- `config`: The configuration object to destroy.

`duckdb_create_error_data` Creates duckdb_error_data. Must be destroyed with `duckdb_destroy_error_data`.

Syntax

```
            duckdb_error_data duckdb_create_error_data(
  duckdb_error_type type,
  const char *message
);
```

Parameters

- `type`: The error type.
- `message`: The error message.

Return Value The error data.

`duckdb_destroy_error_data` Destroys the error data and deallocates its memory.

Syntax

```
            void duckdb_destroy_error_data(
  duckdb_error_data *error_data
);
```

Parameters

- `error_data`: The error data to destroy.

`duckdb_error_data_error_type` Returns the duckdb_error_type of the error data.

Syntax

```
            duckdb_error_type duckdb_error_data_error_type(
  duckdb_error_data error_data
);
```

Parameters

- `error_data`: The error data.

Return Value The error type.

`duckdb_error_data_message` Returns the error message of the error data. Must not be freed.

Syntax

```
            const char *duckdb_error_data_message(
  duckdb_error_data error_data
);
```

Parameters

- `error_data`: The error data.

Return Value The error message.

`duckdb_error_data_has_error` Returns whether the error data contains an error or not.

Syntax

```
            bool duckdb_error_data_has_error(
  duckdb_error_data error_data
);
```

Parameters

- `error_data`: The error data.

Return Value True, if the error data contains an exception, else false.

`duckdb_query` Executes a SQL query within a connection and stores the full (materialized) result in the out_result pointer. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_result_error`.

Note that after running `duckdb_query`, `duckdb_destroy_result` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
            duckdb_state duckdb_query(
  duckdb_connection connection,
  const char *query,
  duckdb_result *out_result
);
```

Parameters

- `connection`: The connection to perform the query in.
- `query`: The SQL query to run.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_result` Closes the result and de-allocates all memory allocated for that result.

Syntax

```
            void duckdb_destroy_result(
  duckdb_result *result
);
```

Parameters

- `result`: The result to destroy.

`duckdb_column_name` Returns the column name of the specified column. The result should not need to be freed; the column names will automatically be destroyed when the result is destroyed.

Returns `NULL` if the column is out of range.

Syntax

```
            const char *duckdb_column_name(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column name from.
- `col`: The column index.

Return Value The column name of the specified column.

`duckdb_column_type` Returns the column type of the specified column.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
            duckdb_type duckdb_column_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The column type of the specified column.

`duckdb_result_statement_type` Returns the statement type of the statement that was executed

Syntax

```
            duckdb_statement_type duckdb_result_statement_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the statement type from.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_column_logical_type` Returns the logical column type of the specified column.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Returns `NULL` if the column is out of range.

Syntax

```
            duckdb_logical_type duckdb_column_logical_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The logical column type of the specified column.

`duckdb_result_get_arrow_options` Returns the arrow options associated with the given result. These options are definitions of how the arrow arrays/schema should be produced.

Syntax

```
            duckdb_arrow_options duckdb_result_get_arrow_options(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch arrow options from.

Return Value The arrow options associated with the given result. This must be destroyed with `duckdb_destroy_arrow_options`.

`duckdb_column_count`

Returns the number of columns present in a the result object.

Syntax

```
            idx_t duckdb_column_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of columns present in the result object.

`duckdb_row_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows present in the result object.

Syntax

```
            idx_t duckdb_row_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows present in the result object.

`duckdb_rows_changed`

Returns the number of rows changed by the query stored in the result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
            idx_t duckdb_rows_changed(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows changed.

`duckdb_column_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the data of a specific column of a result in columnar format.

The function returns a dense array which contains the result data. The exact type stored in the array depends on the corresponding duckdb_type (as provided by `duckdb_column_type`). For the exact type by which the data should be accessed, see the comments in the types section or the DUCKDB_TYPE enum.

For example, for a column of type `DUCKDB_TYPE_INTEGER`, rows can be accessed in the following manner:

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

Syntax

```
           void *duckdb_column_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column data from.
- `col`: The column index.

Return Value The column data of the specified column.

`duckdb_nullmask_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the nullmask of a specific column of a result in columnar format. The nullmask indicates for every row whether or not the corresponding row is `NULL`. If a row is `NULL`, the values present in the array provided by `duckdb_column_data` are undefined.

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
bool *nullmask = duckdb_nullmask_data(&result, 0);
if (nullmask[row]) {
    printf("Data for row %d: NULL\n", row);
} else {
    printf("Data for row %d: %d\n", row, data[row]);
}
```

Syntax

```
           bool *duckdb_nullmask_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the nullmask from.
- `col`: The column index.

Return Value The nullmask of the specified column.

`duckdb_result_error` Returns the error message contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_result` is called.

Syntax

```
            const char *duckdb_result_error(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error of the result.

`duckdb_result_error_type` Returns the result error type contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

Syntax

```
            duckdb_error_type duckdb_result_error_type(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error type of the result.

`duckdb_result_get_chunk`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetches a data chunk from the duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function supersedes all `duckdb_value` functions, as well as the `duckdb_column_data` and `duckdb_nullmask_data` functions. It results in significantly better performance, and should be preferred in newer code-bases.

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions).

Use `duckdb_result_chunk_count` to figure out how many chunks there are in the result.

Syntax

```
            duckdb_data_chunk duckdb_result_get_chunk(
  duckdb_result result,
  idx_t chunk_index
);
```

Parameters

- `result`: The result object to fetch the data chunk from.
- `chunk_index`: The chunk index to fetch from.

Return Value The resulting data chunk. Returns `NULL` if the chunk index is out of bounds.

`duckdb_result_is_streaming`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Checks if the type of the internal result is StreamQueryResult.

Syntax

```
            bool duckdb_result_is_streaming(
  duckdb_result result
);
```

Parameters

- `result`: The result object to check.

Return Value Whether or not the result object is of the type StreamQueryResult

`duckdb_result_chunk_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of data chunks present in the result.

Syntax

```
            idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value Number of data chunks present in the result.

`duckdb_result_return_type` Returns the return_type of the given result, or DUCKDB_RETURN_TYPE_INVALID on error

Syntax

```
            duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value The return_type

`duckdb_value_boolean`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The boolean value at the specified location, or false if the value cannot be converted.

Syntax

```
            bool duckdb_value_boolean(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_int8`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int8_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int8_t duckdb_value_int8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_int16`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int16_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int16_t duckdb_value_int16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_int32`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int32_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int32_t duckdb_value_int32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_int64

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int64_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int64_t duckdb_value_int64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_hugeint

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_hugeint value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_hugeint duckdb_value_hugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uhugeint

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_uhugeint value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_uhugeint duckdb_value_uhugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_decimal

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_decimal value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_decimal duckdb_value_decimal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint8

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint8_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint8_t duckdb_value_uint8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint16

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint16_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint16_t duckdb_value_uint16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint32

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint32_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint32_t duckdb_value_uint32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint64

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint64_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint64_t duckdb_value_uint64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_float

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The float value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            float duckdb_value_float(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_double

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The double value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            double duckdb_value_double(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_date

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_date value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_date duckdb_value_date(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_time

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_time value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_time duckdb_value_time(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_timestamp

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_timestamp value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_timestamp duckdb_value_timestamp(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_interval

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_interval value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_interval duckdb_value_interval(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_varchar

> **Deprecated.** This method has been deprecated. Use duckdb_value_string instead. This function does not work correctly if the string contains null bytes.

Return Value The text value at the specified location as a null-terminated string, or nullptr if the value cannot be converted. The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_value_varchar(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_string

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

No support for nested types, and for other complex types. The resulting field "string.data" must be freed with `duckdb_free`.

Return Value The string value at the specified location. Attempts to cast the result value to string.

Syntax

```
            duckdb_string duckdb_value_string(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_varchar_internal

> **Deprecated.** This method has been deprecated. Use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

Return Value The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

Syntax

```
            char *duckdb_value_varchar_internal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_string_internal

> **Deprecated.** This method has been deprecated. Use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

Return Value The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

Syntax

```
            duckdb_string duckdb_value_string_internal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_blob

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_blob value at the specified location. Returns a blob with blob.data set to nullptr if the value cannot be converted. The resulting field "blob.data" must be freed with `duckdb_free`.

Syntax

```
           duckdb_blob duckdb_value_blob(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_is_null`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value Returns true if the value at the specified index is NULL, and false otherwise.

Syntax

```
           bool duckdb_value_is_null(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_malloc` Allocate `size` bytes of memory using the duckdb internal malloc function. Any memory allocated in this manner should be freed using `duckdb_free`.

Syntax

```
           void *duckdb_malloc(
  size_t size
);
```

Parameters

- `size`: The number of bytes to allocate.

Return Value A pointer to the allocated memory region.

`duckdb_free` Free a value returned from `duckdb_malloc`, `duckdb_value_varchar`, `duckdb_value_blob`, or `duckdb_value_string`.

Syntax

```
           void duckdb_free(
  void *ptr
);
```

Parameters

- `ptr`: The memory region to de-allocate.

`duckdb_vector_size` The internal vector size used by DuckDB. This is the amount of tuples that will fit into a data chunk created by `duckdb_create_data_chunk`.

Return Value The vector size.

Syntax

```
            idx_t duckdb_vector_size(

);
```

`duckdb_string_is_inlined` Whether or not the duckdb_string_t value is inlined. This means that the data of the string does not have a separate allocation.

Syntax

```
            bool duckdb_string_is_inlined(
  duckdb_string_t string
);
```

`duckdb_string_t_length` Get the string length of a string_t

Syntax

```
            uint32_t duckdb_string_t_length(
  duckdb_string_t string
);
```

Parameters

- `string`: The string to get the length of.

Return Value The length.

`duckdb_string_t_data` Get a pointer to the string data of a string_t

Syntax

```
            const char *duckdb_string_t_data(
  duckdb_string_t *string
);
```

Parameters

- `string`: The string to get the pointer to.

Return Value The pointer.

`duckdb_from_date` Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

Syntax

```
            duckdb_date_struct duckdb_from_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value The `duckdb_date_struct` with the decomposed elements.

`duckdb_to_date` Re-compose a `duckdb_date` from year, month and date (`duckdb_date_struct`).

Syntax

```
            duckdb_date duckdb_to_date(
  duckdb_date_struct date
);
```

Parameters

- `date`: The year, month and date stored in a `duckdb_date_struct`.

Return Value The `duckdb_date` element.

`duckdb_is_finite_date` Test a `duckdb_date` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value True if the date is finite, false if it is ±infinity.

`duckdb_from_time` Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

Syntax

```
            duckdb_time_struct duckdb_from_time(
  duckdb_time time
);
```

Parameters

- `time`: The time object, as obtained from a `DUCKDB_TYPE_TIME` column.

Return Value The `duckdb_time_struct` with the decomposed elements.

`duckdb_create_time_tz` Create a `duckdb_time_tz` object from micros and a timezone offset.

Syntax

```
           duckdb_time_tz duckdb_create_time_tz(
  int64_t micros,
  int32_t offset
);
```

Parameters

- `micros`: The microsecond component of the time.
- `offset`: The timezone offset component of the time.

Return Value The `duckdb_time_tz` element.

`duckdb_from_time_tz` Decompose a TIME_TZ objects into micros and a timezone offset.

Use `duckdb_from_time` to further decompose the micros into hour, minute, second and microsecond.

Syntax

```
           duckdb_time_tz_struct duckdb_from_time_tz(
  duckdb_time_tz micros
);
```

Parameters

- `micros`: The time object, as obtained from a `DUCKDB_TYPE_TIME_TZ` column.

`duckdb_to_time` Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

Syntax

```
           duckdb_time duckdb_to_time(
  duckdb_time_struct time
);
```

Parameters

- `time`: The hour, minute, second and microsecond in a `duckdb_time_struct`.

Return Value The `duckdb_time` element.

`duckdb_from_timestamp` Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

Syntax

```
           duckdb_timestamp_struct duckdb_from_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The ts object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value The `duckdb_timestamp_struct` with the decomposed elements.

`duckdb_to_timestamp`

Re-compose a `duckdb_timestamp` from a duckdb_timestamp_struct.

Syntax

```
            duckdb_timestamp duckdb_to_timestamp(
  duckdb_timestamp_struct ts
);
```

Parameters

- `ts`: The de-composed elements in a `duckdb_timestamp_struct`.

Return Value The `duckdb_timestamp` element.

`duckdb_is_finite_timestamp`

Test a `duckdb_timestamp` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The duckdb_timestamp object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_s`

Test a `duckdb_timestamp_s` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_s(
  duckdb_timestamp_s ts
);
```

Parameters

- `ts`: The duckdb_timestamp_s object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_S` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ms`

Test a `duckdb_timestamp_ms` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ms(
  duckdb_timestamp_ms ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ms object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_MS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ns` Test a `duckdb_timestamp_ns` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ns(
  duckdb_timestamp_ns ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ns object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_NS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_hugeint_to_double` Converts a duckdb_hugeint object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

Syntax

```
            double duckdb_hugeint_to_double(
  duckdb_hugeint val
);
```

Parameters

- `val`: The hugeint value.

Return Value The converted `double` element.

`duckdb_double_to_hugeint` Converts a double value to a duckdb_hugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
            duckdb_hugeint duckdb_double_to_hugeint(
  double val
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_hugeint` element.

`duckdb_uhugeint_to_double` Converts a duckdb_uhugeint object (as obtained from a `DUCKDB_TYPE_UHUGEINT` column) into a double.

Syntax

```
            double duckdb_uhugeint_to_double(
  duckdb_uhugeint val
);
```

Parameters

- `val`: The uhugeint value.

Return Value The converted `double` element.

`duckdb_double_to_uhugeint` Converts a double value to a duckdb_uhugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
            duckdb_uhugeint duckdb_double_to_uhugeint(
  double val
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_uhugeint` element.

`duckdb_double_to_decimal` Converts a double value to a duckdb_decimal object.

If the conversion fails because the double value is too big, or the width/scale are invalid the result will be 0.

Syntax

```
            duckdb_decimal duckdb_double_to_decimal(
  double val,
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_decimal` element.

`duckdb_decimal_to_double` Converts a duckdb_decimal object (as obtained from a `DUCKDB_TYPE_DECIMAL` column) into a double.

Syntax

```
            double duckdb_decimal_to_double(
  duckdb_decimal val
);
```

Parameters

- `val`: The decimal value.

Return Value The converted `double` element.

`duckdb_prepare` Create a prepared statement object from a query.

Note that after calling `duckdb_prepare`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
            duckdb_state duckdb_prepare(
  duckdb_connection connection,
  const char *query,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`: The connection object
- `query`: The SQL query to prepare
- `out_prepared_statement`: The resulting prepared statement object

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_prepare` Closes the prepared statement and de-allocates all memory allocated for the statement.

Syntax

```
            void duckdb_destroy_prepare(
  duckdb_prepared_statement *prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to destroy.

`duckdb_prepare_error` Returns the error message associated with the given prepared statement. If the prepared statement has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_prepare` is called.

Syntax

```
            const char *duckdb_prepare_error(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_nparams` Returns the number of parameters that can be provided to the given prepared statement.

Returns 0 if the query was not successfully prepared.

Syntax

```
            idx_t duckdb_nparams(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the number of parameters for.

`duckdb_parameter_name` Returns the name used to identify the parameter The returned string should be freed using `duckdb_free`.

Returns NULL if the index is out of range for the provided prepared statement.

Syntax

```
            const char *duckdb_parameter_name(
  duckdb_prepared_statement prepared_statement,
  idx_t index
);
```

Parameters

- `prepared_statement`: The prepared statement for which to get the parameter name from.

`duckdb_param_type` Returns the parameter type for the parameter at the given index.

Returns `DUCKDB_TYPE_INVALID` if the parameter index is out of range or the statement was not successfully prepared.

Syntax

```
            duckdb_type duckdb_param_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The parameter type

`duckdb_param_logical_type` Returns the logical type for the parameter at the given index.

Returns `nullptr` if the parameter index is out of range or the statement was not successfully prepared.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
           duckdb_logical_type duckdb_param_logical_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The logical type of the parameter

`duckdb_clear_bindings` Clear the params bind to the prepared statement.

Syntax

```
           duckdb_state duckdb_clear_bindings(
  duckdb_prepared_statement prepared_statement
);
```

`duckdb_prepared_statement_type` Returns the statement type of the statement to be executed

Syntax

```
           duckdb_statement_type duckdb_prepared_statement_type(
  duckdb_prepared_statement statement
);
```

Parameters

- `statement`: The prepared statement.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_prepared_statement_column_count` Returns the number of columns present in a the result of the prepared statement. If any of the column types are invalid, the result will be 1.

Syntax

```
            idx_t duckdb_prepared_statement_column_count(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement.

Return Value The number of columns present in the result of the prepared statement.

`duckdb_prepared_statement_column_name` Returns the name of the specified column of the result of the prepared_statement. The returned string should be freed using `duckdb_free`.

Returns `nullptr` if the column is out of range.

Syntax

```
            const char *duckdb_prepared_statement_column_name(
  duckdb_prepared_statement prepared_statement,
  idx_t col_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `col_idx`: The column index.

Return Value The column name of the specified column.

`duckdb_prepared_statement_column_logical_type` Returns the column type of the specified column of the result of the prepared_statement.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range. The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_prepared_statement_column_logical_type(
  duckdb_prepared_statement prepared_statement,
  idx_t col_idx
);
```

Parameters

- `prepared_statement`: The prepared statement to fetch the column type from.
- `col_idx`: The column index.

Return Value The logical type of the specified column.

`duckdb_prepared_statement_column_type` Returns the column type of the specified column of the result of the prepared_statement.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
           duckdb_type duckdb_prepared_statement_column_type(
  duckdb_prepared_statement prepared_statement,
  idx_t col_idx
);
```

Parameters

- `prepared_statement`: The prepared statement to fetch the column type from.
- `col_idx`: The column index.

Return Value The type of the specified column.

`duckdb_bind_value`

Binds a value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_value(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_value val
);
```

`duckdb_bind_parameter_index`

Retrieve the index of the parameter for the prepared statement, identified by name

Syntax

```
           duckdb_state duckdb_bind_parameter_index(
  duckdb_prepared_statement prepared_statement,
  idx_t *param_idx_out,
  const char *name
);
```

`duckdb_bind_boolean`

Binds a bool value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_boolean(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  bool val
);
```

`duckdb_bind_int8`

Binds an int8_t value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_int8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int8_t val
);
```

`duckdb_bind_int16`

Binds an int16_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int16_t val
);
```

`duckdb_bind_int32` Binds an int32_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int32_t val
);
```

`duckdb_bind_int64` Binds an int64_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int64_t val
);
```

`duckdb_bind_hugeint` Binds a duckdb_hugeint value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_hugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_hugeint val
);
```

`duckdb_bind_uhugeint` Binds a duckdb_uhugeint value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_uhugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_uhugeint val
);
```

`duckdb_bind_decimal` Binds a duckdb_decimal value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_decimal(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_decimal val
);
```

`duckdb_bind_uint8` Binds a uint8_t value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_uint8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint8_t val
);
```

`duckdb_bind_uint16` Binds a uint16_t value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_uint16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint16_t val
);
```

`duckdb_bind_uint32` Binds a uint32_t value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_uint32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint32_t val
);
```

`duckdb_bind_uint64` Binds a uint64_t value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_uint64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint64_t val
);
```

`duckdb_bind_float` Binds a float value to the prepared statement at the specified index.

Syntax

```
           duckdb_state duckdb_bind_float(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  float val
);
```

`duckdb_bind_double` Binds a double value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_double(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  double val
);
```

`duckdb_bind_date` Binds a duckdb_date value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_date(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_date val
);
```

`duckdb_bind_time` Binds a duckdb_time value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_time(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_time val
);
```

`duckdb_bind_timestamp` Binds a duckdb_timestamp value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_timestamp(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_timestamp val
);
```

`duckdb_bind_timestamp_tz` Binds a duckdb_timestamp value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_timestamp_tz(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_timestamp val
);
```

`duckdb_bind_interval` Binds a duckdb_interval value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_interval(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_interval val
);
```

`duckdb_bind_varchar` Binds a null-terminated varchar value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_varchar(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val
);
```

`duckdb_bind_varchar_length` Binds a varchar value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_varchar_length(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val,
  idx_t length
);
```

`duckdb_bind_blob` Binds a blob value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_blob(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const void *data,
  idx_t length
);
```

`duckdb_bind_null` Binds a NULL value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_null(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

`duckdb_execute_prepared` Executes the prepared statement with the given bound parameters, and returns a materialized query result.

This method can be called multiple times for each prepared statement, and the parameters can be modified between calls to this function.

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
            duckdb_state duckdb_execute_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_execute_prepared_streaming`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns an optionally-streaming query result. To determine if the resulting query was in fact streamed, use `duckdb_result_is_streaming`

This method can be called multiple times for each prepared statement, and the parameters can be modified between calls to this function.

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
           duckdb_state duckdb_execute_prepared_streaming(
  duckdb_prepared_statement prepared_statement,
  duckdb_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_extract_statements` Extract all statements from a query. Note that after calling `duckdb_extract_statements`, the extracted statements should always be destroyed using `duckdb_destroy_extracted`, even if no statements were extracted.

If the extract fails, `duckdb_extract_statements_error` can be called to obtain the reason why the extract failed.

Syntax

```
           idx_t duckdb_extract_statements(
  duckdb_connection connection,
  const char *query,
  duckdb_extracted_statements *out_extracted_statements
);
```

Parameters

- `connection`: The connection object
- `query`: The SQL query to extract
- `out_extracted_statements`: The resulting extracted statements object

Return Value The number of extracted statements or 0 on failure.

`duckdb_prepare_extracted_statement` Prepare an extracted statement. Note that after calling `duckdb_prepare_extracted_statement`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
            duckdb_state duckdb_prepare_extracted_statement(
  duckdb_connection connection,
  duckdb_extracted_statements extracted_statements,
  idx_t index,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`: The connection object
- `extracted_statements`: The extracted statements object
- `index`: The index of the extracted statement to prepare
- `out_prepared_statement`: The resulting prepared statement object

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_extract_statements_error` Returns the error message contained within the extracted statements. The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_extracted` is called.

Syntax

```
            const char *duckdb_extract_statements_error(
  duckdb_extracted_statements extracted_statements
);
```

Parameters

- `extracted_statements`: The extracted statements to fetch the error from.

Return Value The error of the extracted statements.

`duckdb_destroy_extracted` De-allocates all memory allocated for the extracted statements.

Syntax

```
            void duckdb_destroy_extracted(
  duckdb_extracted_statements *extracted_statements
);
```

Parameters

- `extracted_statements`: The extracted statements to destroy.

`duckdb_pending_prepared` Executes the prepared statement with the given bound parameters, and returns a pending result. The pending result represents an intermediate structure for a query that is not yet fully executed. The pending result can be used to incrementally execute a query, returning control to the client between tasks.

Note that after calling `duckdb_pending_prepared`, the pending result should always be destroyed using `duckdb_destroy_pending`, even if this function returns DuckDBError.

Syntax

```
            duckdb_state duckdb_pending_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_pending_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The pending query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_pending_prepared_streaming`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns a pending result. This pending result will create a streaming duckdb_result when executed. The pending result represents an intermediate structure for a query that is not yet fully executed.

Note that after calling `duckdb_pending_prepared_streaming`, the pending result should always be destroyed using `duckdb_destroy_pending`, even if this function returns DuckDBError.

Syntax

```
            duckdb_state duckdb_pending_prepared_streaming(
  duckdb_prepared_statement prepared_statement,
  duckdb_pending_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The pending query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_pending` Closes the pending result and de-allocates all memory allocated for the result.

Syntax

```
            void duckdb_destroy_pending(
  duckdb_pending_result *pending_result
);
```

Parameters

- `pending_result`: The pending result to destroy.

`duckdb_pending_error` Returns the error message contained within the pending result.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_pending` is called.

Syntax

```
            const char *duckdb_pending_error(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`: The pending result to fetch the error from.

Return Value The error of the pending result.

`duckdb_pending_execute_task` Executes a single task within the query, returning whether or not the query is ready.

If this returns DUCKDB_PENDING_RESULT_READY, the duckdb_execute_pending function can be called to obtain the result. If this returns DUCKDB_PENDING_RESULT_NOT_READY, the duckdb_pending_execute_task function should be called again. If this returns DUCKDB_PENDING_ERROR, an error occurred during execution.

The error message can be obtained by calling duckdb_pending_error on the pending_result.

Syntax

```
            duckdb_pending_state duckdb_pending_execute_task(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`: The pending result to execute a task within.

Return Value The state of the pending result after the execution.

`duckdb_pending_execute_check_state` If this returns DUCKDB_PENDING_RESULT_READY, the duckdb_execute_pending function can be called to obtain the result. If this returns DUCKDB_PENDING_RESULT_NOT_READY, the duckdb_pending_execute_check_state function should be called again. If this returns DUCKDB_PENDING_ERROR, an error occurred during execution.

The error message can be obtained by calling duckdb_pending_error on the pending_result.

Syntax

```
            duckdb_pending_state duckdb_pending_execute_check_state(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`: The pending result.

Return Value The state of the pending result.

`duckdb_execute_pending` Fully execute a pending query result, returning the final query result.

If duckdb_pending_execute_task has been called until DUCKDB_PENDING_RESULT_READY was returned, this will return fast. Otherwise, all remaining tasks must be executed first.

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
            duckdb_state duckdb_execute_pending(
  duckdb_pending_result pending_result,
  duckdb_result *out_result
);
```

Parameters

- `pending_result`: The pending result to execute.
- `out_result`: The result object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_pending_execution_is_finished` Returns whether a duckdb_pending_state is finished executing. For example if `pending_state` is DUCKDB_PENDING_RESULT_READY, this function will return true.

Syntax

```
            bool duckdb_pending_execution_is_finished(
  duckdb_pending_state pending_state
);
```

Parameters

- `pending_state`: The pending state on which to decide whether to finish execution.

Return Value Boolean indicating pending execution should be considered finished.

`duckdb_destroy_value` Destroys the value and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_value(
  duckdb_value *value
);
```

Parameters

- `value`: The value to destroy.

`duckdb_create_varchar` Creates a value from a null-terminated string

Syntax

```
            duckdb_value duckdb_create_varchar(
  const char *text
);
```

Parameters

- `text`: The null-terminated string

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varchar_length` Creates a value from a string

Syntax

```
            duckdb_value duckdb_create_varchar_length(
  const char *text,
  idx_t length
);
```

Parameters

- `text`: The text
- `length`: The length of the text

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bool` Creates a value from a boolean

Syntax

```
            duckdb_value duckdb_create_bool(
  bool input
);
```

Parameters

- `input`: The boolean value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int8` Creates a value from an int8_t (a tinyint)

Syntax

```
            duckdb_value duckdb_create_int8(
  int8_t input
);
```

Parameters

- `input`: The tinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint8` Creates a value from a uint8_t (a utinyint)

Syntax

```
            duckdb_value duckdb_create_uint8(
  uint8_t input
);
```

Parameters

- `input`: The utinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int16` Creates a value from an int16_t (a smallint)

Syntax

```
            duckdb_value duckdb_create_int16(
  int16_t input
);
```

Parameters

- `input`: The smallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint16` Creates a value from a uint16_t (a usmallint)

Syntax

```
            duckdb_value duckdb_create_uint16(
  uint16_t input
);
```

Parameters

- `input`: The usmallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int32` Creates a value from an int32_t (an integer)

Syntax

```
            duckdb_value duckdb_create_int32(
  int32_t input
);
```

Parameters

- `input`: The integer value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint32` Creates a value from a uint32_t (a uinteger)

Syntax

```
duckdb_value duckdb_create_uint32(
  uint32_t input
);
```

Parameters

- `input`: The uinteger value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint64` Creates a value from a uint64_t (a ubigint)

Syntax

```
duckdb_value duckdb_create_uint64(
  uint64_t input
);
```

Parameters

- `input`: The ubigint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int64` Creates a value from an int64

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_int64(
  int64_t val
);
```

`duckdb_create_hugeint` Creates a value from a hugeint

Syntax

```
duckdb_value duckdb_create_hugeint(
  duckdb_hugeint input
);
```

Parameters

- `input`: The hugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uhugeint` Creates a value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uhugeint(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The uhugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bignum` Creates a BIGNUM value from a duckdb_bignum

Syntax

```
            duckdb_value duckdb_create_bignum(
  duckdb_bignum input
);
```

Parameters

- `input`: The duckdb_bignum value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_decimal` Creates a DECIMAL value from a duckdb_decimal

Syntax

```
            duckdb_value duckdb_create_decimal(
  duckdb_decimal input
);
```

Parameters

- `input`: The duckdb_decimal value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_float` Creates a value from a float

Syntax

```
            duckdb_value duckdb_create_float(
  float input
);
```

Parameters

- `input`: The float value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_double` Creates a value from a double

Syntax

```
            duckdb_value duckdb_create_double(
  double input
);
```

Parameters

- `input`: The double value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_date` Creates a value from a date

Syntax

```
            duckdb_value duckdb_create_date(
  duckdb_date input
);
```

Parameters

- `input`: The date value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time` Creates a value from a time

Syntax

```
            duckdb_value duckdb_create_time(
  duckdb_time input
);
```

Parameters

- `input`: The time value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time_ns` Creates a value from a time_ns

Syntax

```
            duckdb_value duckdb_create_time_ns(
  duckdb_time_ns input
);
```

Parameters

- `input`: The time value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time_tz_value` Creates a value from a time_tz. Not to be confused with `duckdb_create_time_tz`, which creates a duckdb_time_tz_t.

Syntax

```
            duckdb_value duckdb_create_time_tz_value(
  duckdb_time_tz value
);
```

Parameters

- `value`: The time_tz value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp` Creates a TIMESTAMP value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_tz` Creates a TIMESTAMP_TZ value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp_tz(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_s` Creates a TIMESTAMP_S value from a duckdb_timestamp_s

Syntax

```
            duckdb_value duckdb_create_timestamp_s(
  duckdb_timestamp_s input
);
```

Parameters

- `input`: The duckdb_timestamp_s value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ms` Creates a TIMESTAMP_MS value from a duckdb_timestamp_ms

Syntax

```
            duckdb_value duckdb_create_timestamp_ms(
  duckdb_timestamp_ms input
);
```

Parameters

- `input`: The duckdb_timestamp_ms value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ns` Creates a TIMESTAMP_NS value from a duckdb_timestamp_ns

Syntax

```
            duckdb_value duckdb_create_timestamp_ns(
  duckdb_timestamp_ns input
);
```

Parameters

- `input`: The duckdb_timestamp_ns value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_interval` Creates a value from an interval

Syntax

```
            duckdb_value duckdb_create_interval(
  duckdb_interval input
);
```

Parameters

- `input`: The interval value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_blob` Creates a value from a blob

Syntax

```
            duckdb_value duckdb_create_blob(
  const uint8_t *data,
  idx_t length
);
```

Parameters

- `data`: The blob data
- `length`: The length of the blob data

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bit` Creates a BIT value from a duckdb_bit

Syntax

```
            duckdb_value duckdb_create_bit(
  duckdb_bit input
);
```

Parameters

- `input`: The duckdb_bit value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uuid` Creates a UUID value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uuid(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The duckdb_uhugeint containing the UUID

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_get_bool` Returns the boolean value of the given value.

Syntax

```
            bool duckdb_get_bool(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a boolean

Return Value A boolean, or false if the value cannot be converted

`duckdb_get_int8` Returns the int8_t value of the given value.

Syntax

```
            int8_t duckdb_get_int8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a tinyint

Return Value A int8_t, or MinValue if the value cannot be converted

`duckdb_get_uint8` Returns the uint8_t value of the given value.

Syntax

```
            uint8_t duckdb_get_uint8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a utinyint

Return Value A uint8_t, or MinValue if the value cannot be converted

`duckdb_get_int16` Returns the int16_t value of the given value.

Syntax

```
            int16_t duckdb_get_int16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a smallint

Return Value A int16_t, or MinValue if the value cannot be converted

`duckdb_get_uint16` Returns the uint16_t value of the given value.

Syntax

```
            uint16_t duckdb_get_uint16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a usmallint

Return Value A uint16_t, or MinValue if the value cannot be converted

`duckdb_get_int32` Returns the int32_t value of the given value.

Syntax

```
            int32_t duckdb_get_int32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing an integer

Return Value A int32_t, or MinValue if the value cannot be converted

`duckdb_get_uint32` Returns the uint32_t value of the given value.

Syntax

```
            uint32_t duckdb_get_uint32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uinteger

Return Value A uint32_t, or MinValue if the value cannot be converted

`duckdb_get_int64` Returns the int64_t value of the given value.

Syntax

```
            int64_t duckdb_get_int64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a bigint

Return Value A int64_t, or MinValue if the value cannot be converted

`duckdb_get_uint64` Returns the uint64_t value of the given value.

Syntax

```
            uint64_t duckdb_get_uint64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a ubigint

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_hugeint` Returns the hugeint value of the given value.

Syntax

```
            duckdb_hugeint duckdb_get_hugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a hugeint

Return Value A duckdb_hugeint, or MinValue if the value cannot be converted

`duckdb_get_uhugeint` Returns the uhugeint value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uhugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uhugeint

Return Value A duckdb_uhugeint, or MinValue if the value cannot be converted

`duckdb_get_bignum` Returns the duckdb_bignum value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_bignum duckdb_get_bignum(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a BIGNUM

Return Value A duckdb_bignum. The `data` field must be destroyed with `duckdb_free`.

`duckdb_get_decimal`

Returns the duckdb_decimal value of the given value.

Syntax

```
            duckdb_decimal duckdb_get_decimal(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a DECIMAL

Return Value A duckdb_decimal, or MinValue if the value cannot be converted

`duckdb_get_float`

Returns the float value of the given value.

Syntax

```
            float duckdb_get_float(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a float

Return Value A float, or NAN if the value cannot be converted

`duckdb_get_double`

Returns the double value of the given value.

Syntax

```
            double duckdb_get_double(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a double

Return Value A double, or NAN if the value cannot be converted

`duckdb_get_date` Returns the date value of the given value.

Syntax

```
            duckdb_date duckdb_get_date(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a date

Return Value A duckdb_date, or MinValue if the value cannot be converted

`duckdb_get_time` Returns the time value of the given value.

Syntax

```
            duckdb_time duckdb_get_time(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time

Return Value A duckdb_time, or MinValue if the value cannot be converted

`duckdb_get_time_ns` Returns the time_ns value of the given value.

Syntax

```
            duckdb_time_ns duckdb_get_time_ns(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time_ns

Return Value A duckdb_time_ns, or MinValue if the value cannot be converted

`duckdb_get_time_tz` Returns the time_tz value of the given value.

Syntax

```
            duckdb_time_tz duckdb_get_time_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time_tz

Return Value A duckdb_time_tz, or MinValue if the value cannot be converted

`duckdb_get_timestamp` Returns the TIMESTAMP value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_tz` Returns the TIMESTAMP_TZ value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_TZ

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_s` Returns the duckdb_timestamp_s value of the given value.

Syntax

```
            duckdb_timestamp_s duckdb_get_timestamp_s(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_S

Return Value A duckdb_timestamp_s, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ms` Returns the duckdb_timestamp_ms value of the given value.

Syntax

```
            duckdb_timestamp_ms duckdb_get_timestamp_ms(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_MS

Return Value A duckdb_timestamp_ms, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ns` Returns the duckdb_timestamp_ns value of the given value.

Syntax

```
            duckdb_timestamp_ns duckdb_get_timestamp_ns(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_NS

Return Value A duckdb_timestamp_ns, or MinValue if the value cannot be converted

`duckdb_get_interval` Returns the interval value of the given value.

Syntax

```
            duckdb_interval duckdb_get_interval(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a interval

Return Value A duckdb_interval, or MinValue if the value cannot be converted

`duckdb_get_value_type` Returns the type of the given value. The type is valid as long as the value is not destroyed. The type itself must not be destroyed.

Syntax

```
            duckdb_logical_type duckdb_get_value_type(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value

Return Value A duckdb_logical_type.

`duckdb_get_blob` Returns the blob value of the given value.

Syntax

```
            duckdb_blob duckdb_get_blob(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a blob

Return Value A duckdb_blob

`duckdb_get_bit` Returns the duckdb_bit value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_bit duckdb_get_bit(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a BIT

Return Value A duckdb_bit

`duckdb_get_uuid` Returns a duckdb_uhugeint representing the UUID value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uuid(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a UUID

Return Value A duckdb_uhugeint representing the UUID value

`duckdb_get_varchar` Obtains a string representation of the given value. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_get_varchar(
  duckdb_value value
);
```

Parameters

- `value`: The value

Return Value The string value. This must be destroyed with `duckdb_free`.

`duckdb_create_struct_value` Creates a struct value from a type and an array of values. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_struct_value(
  duckdb_logical_type type,
  duckdb_value *values
);
```

Parameters

- `type`: The type of the struct
- `values`: The values for the struct fields

Return Value The struct value, or nullptr, if any child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_list_value` Creates a list value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_list_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the list
- `values`: The values for the list
- `value_count`: The number of values in the list

Return Value The list value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_array_value` Creates an array value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_array_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the array
- `values`: The values for the array
- `value_count`: The number of values in the array

Return Value The array value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_map_value` Creates a map value from a map type and two arrays, one for the keys and one for the values, each of length `entry_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
           duckdb_value duckdb_create_map_value(
  duckdb_logical_type map_type,
  duckdb_value *keys,
  duckdb_value *values,
  idx_t entry_count
);
```

Parameters

- `map_type`: The map type
- `keys`: The keys of the map
- `values`: The values of the map
- `entry_count`: The number of entries (key-value pairs) in the map

Return Value The map value, or nullptr, if the parameters are invalid.

`duckdb_create_union_value` Creates a union value from a union type, a tag index, and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
           duckdb_value duckdb_create_union_value(
  duckdb_logical_type union_type,
  idx_t tag_index,
  duckdb_value value
);
```

Parameters

- `union_type`: The union type
- `tag_index`: The index of the tag of the union
- `value`: The value of the union for that tag

Return Value The union value, or nullptr, if the parameters are invalid.

`duckdb_get_map_size` Returns the number of elements in a MAP value.

Syntax

```
            idx_t duckdb_get_map_size(
  duckdb_value value
);
```

Parameters

- `value`: The MAP value.

Return Value The number of elements in the map.

`duckdb_get_map_key` Returns the MAP key at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_key(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the key.

Return Value The key as a duckdb_value.

`duckdb_get_map_value` Returns the MAP value at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_value(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the value.

Return Value The value as a duckdb_value.

`duckdb_is_null_value` Returns whether the value's type is SQLNULL or not.

Syntax

```
            bool duckdb_is_null_value(
  duckdb_value value
);
```

Parameters

- `value`: The value to check.

Return Value True, if the value's type is SQLNULL, otherwise false.

`duckdb_create_null_value` Creates a value of type SQLNULL.

Return Value The duckdb_value representing SQLNULL. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_null_value(

);
```

`duckdb_get_list_size` Returns the number of elements in a LIST value.

Syntax

```
idx_t duckdb_get_list_size(
  duckdb_value value
);
```

Parameters

- `value`: The LIST value.

Return Value The number of elements in the list.

`duckdb_get_list_child` Returns the LIST child at index as a duckdb_value.

Syntax

```
duckdb_value duckdb_get_list_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The LIST value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_create_enum_value` Creates an enum value from a type and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_enum_value(
  duckdb_logical_type type,
  uint64_t value
);
```

Parameters

- `type`: The type of the enum
- `value`: The value for the enum

Return Value The enum value, or nullptr.

`duckdb_get_enum_value`

Returns the enum value of the given value.

Syntax

```
            uint64_t duckdb_get_enum_value(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value containing an enum

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_struct_child`

Returns the STRUCT child at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_struct_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The STRUCT value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_value_to_string`

Returns the SQL string representation of the given value.

Syntax

```
            char *duckdb_value_to_string(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value.

Return Value The SQL string representation as a null-terminated string. The result must be freed with `duckdb_free`.

`duckdb_create_logical_type` Creates a `duckdb_logical_type` from a primitive type. The resulting logical type must be destroyed with `duckdb_destroy_logical_type`.

Returns an invalid logical type, if type is: `DUCKDB_TYPE_INVALID`, `DUCKDB_TYPE_DECIMAL`, `DUCKDB_TYPE_ENUM`, `DUCKDB_TYPE_LIST`, `DUCKDB_TYPE_STRUCT`, `DUCKDB_TYPE_MAP`, `DUCKDB_TYPE_ARRAY`, or `DUCKDB_TYPE_UNION`.

Syntax

```
            duckdb_logical_type duckdb_create_logical_type(
  duckdb_type type
);
```

Parameters

- `type`: The primitive type to create.

Return Value The logical type.

`duckdb_logical_type_get_alias` Returns the alias of a duckdb_logical_type, if set, else `nullptr`. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_logical_type_get_alias(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type

Return Value The alias or `nullptr`

`duckdb_logical_type_set_alias` Sets the alias of a duckdb_logical_type.

Syntax

```
            void duckdb_logical_type_set_alias(
  duckdb_logical_type type,
  const char *alias
);
```

Parameters

- `type`: The logical type
- `alias`: The alias to set

`duckdb_create_list_type` Creates a LIST type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_list_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The child type of the list

Return Value The logical type.

`duckdb_create_array_type` Creates an ARRAY type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_array_type(
  duckdb_logical_type type,
  idx_t array_size
);
```

Parameters

- `type`: The child type of the array.
- `array_size`: The number of elements in the array.

Return Value The logical type.

`duckdb_create_map_type` Creates a MAP type from its key type and value type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_map_type(
  duckdb_logical_type key_type,
  duckdb_logical_type value_type
);
```

Parameters

- `key_type`: The map's key type.
- `value_type`: The map's value type.

Return Value The logical type.

`duckdb_create_union_type` Creates a UNION type from the passed arrays. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_union_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of union member types.
- `member_names`: The union member names.
- `member_count`: The number of union members.

Return Value The logical type.

`duckdb_create_struct_type` Creates a STRUCT type based on the member types and names. The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_struct_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of types of the struct members.
- `member_names`: The array of names of the struct members.
- `member_count`: The number of members of the struct.

Return Value The logical type.

`duckdb_create_enum_type` Creates an ENUM type from the passed member name array. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_enum_type(
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_names`: The array of names that the enum should consist of.
- `member_count`: The number of elements that were specified in the array.

Return Value The logical type.

`duckdb_create_decimal_type` Creates a DECIMAL type with the specified width and scale. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_decimal_type(
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `width`: The width of the decimal type
- `scale`: The scale of the decimal type

Return Value The logical type.

`duckdb_get_type_id` Retrieves the enum `duckdb_type` of a `duckdb_logical_type`.

Syntax

```
            duckdb_type duckdb_get_type_id(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type.

Return Value The `duckdb_type` id.

`duckdb_decimal_width` Retrieves the width of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_width(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The width of the decimal type

`duckdb_decimal_scale` Retrieves the scale of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_scale(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value

The scale of the decimal type

`duckdb_decimal_internal_type`

Retrieves the internal storage type of a decimal type.

Syntax

```
            duckdb_type duckdb_decimal_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value

The internal type of the decimal type

`duckdb_enum_internal_type`

Retrieves the internal storage type of an enum type.

Syntax

```
            duckdb_type duckdb_enum_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value

The internal type of the enum type

`duckdb_enum_dictionary_size`

Retrieves the dictionary size of the enum type.

Syntax

```
            uint32_t duckdb_enum_dictionary_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value

The dictionary size of the enum type

`duckdb_enum_dictionary_value`

Retrieves the dictionary value at the specified position from the enum.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_enum_dictionary_value(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The index in the dictionary

Return Value The string value of the enum type. Must be freed with `duckdb_free`.

`duckdb_list_type_child_type` Retrieves the child type of the given LIST type. Also accepts MAP types. The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_list_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type, either LIST or MAP.

Return Value The child type of the LIST or MAP type.

`duckdb_array_type_child_type` Retrieves the child type of the given ARRAY type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_array_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type. Must be ARRAY.

Return Value The child type of the ARRAY type.

`duckdb_array_type_array_size` Retrieves the array size of the given array type.

Syntax

```
            idx_t duckdb_array_type_array_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The fixed number of elements the values of this array type can store.

`duckdb_map_type_key_type`

Retrieves the key type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_key_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The key type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_value_type`

Retrieves the value type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_value_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The value type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_struct_type_child_count`

Returns the number of children of a struct type.

Syntax

```
            idx_t duckdb_struct_type_child_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The number of children of a struct type.

`duckdb_struct_type_child_name` Retrieves the name of the struct child.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_struct_type_child_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the struct type. Must be freed with `duckdb_free`.

`duckdb_struct_type_child_type` Retrieves the child type of the given struct type at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_struct_type_child_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the struct type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_union_type_member_count` Returns the number of members that the union type has.

Syntax

```
            idx_t duckdb_union_type_member_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type (union) object

Return Value The number of members of a union type.

`duckdb_union_type_member_name` Retrieves the name of the union member.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_union_type_member_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the union member. Must be freed with `duckdb_free`.

`duckdb_union_type_member_type`

Retrieves the child type of the given union member at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_union_type_member_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the union member. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_destroy_logical_type`

Destroys the logical type and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_logical_type(
  duckdb_logical_type *type
);
```

Parameters

- `type`: The logical type to destroy.

`duckdb_register_logical_type`

Registers a custom type within the given connection. The type must have an alias

Syntax

```
            duckdb_state duckdb_register_logical_type(
  duckdb_connection con,
  duckdb_logical_type type,
  duckdb_create_type_info info
);
```

Parameters

- `con`: The connection to use
- `type`: The custom type to register

Return Value

Whether or not the registration was successful.

`duckdb_create_data_chunk`

Creates an empty data chunk with the specified column types. The result must be destroyed with `duckdb_destroy_data_chunk`.

Syntax

```
            duckdb_data_chunk duckdb_create_data_chunk(
  duckdb_logical_type *types,
  idx_t column_count
);
```

Parameters

- `types`: An array of column types. Column types can not contain ANY and INVALID types.
- `column_count`: The number of columns.

Return Value

The data chunk.

`duckdb_destroy_data_chunk`

Destroys the data chunk and de-allocates all memory allocated for that chunk.

Syntax

```
            void duckdb_destroy_data_chunk(
  duckdb_data_chunk *chunk
);
```

Parameters

- `chunk`: The data chunk to destroy.

`duckdb_data_chunk_reset`

Resets a data chunk, clearing the validity masks and setting the cardinality of the data chunk to 0. After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            void duckdb_data_chunk_reset(
  duckdb_data_chunk chunk
);
```

Parameters

- `chunk`: The data chunk to reset.

`duckdb_data_chunk_get_column_count`

Retrieves the number of columns in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_column_count(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of columns in the data chunk

`duckdb_data_chunk_get_vector` Retrieves the vector at the specified column index in the data chunk.

The pointer to the vector is valid for as long as the chunk is alive. It does NOT need to be destroyed.

Syntax

```
            duckdb_vector duckdb_data_chunk_get_vector(
  duckdb_data_chunk chunk,
  idx_t col_idx
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The vector

`duckdb_data_chunk_get_size` Retrieves the current number of tuples in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_size(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of tuples in the data chunk

`duckdb_data_chunk_set_size` Sets the current number of tuples in a data chunk.

Syntax

```
            void duckdb_data_chunk_set_size(
  duckdb_data_chunk chunk,
  idx_t size
);
```

Parameters

- `chunk`: The data chunk to set the size in
- `size`: The number of tuples in the data chunk

`duckdb_create_vector` Creates a flat vector. Must be destroyed with `duckdb_destroy_vector`.

Syntax

```
            duckdb_vector duckdb_create_vector(
  duckdb_logical_type type,
  idx_t capacity
);
```

Parameters

- `type`: The logical type of the vector.
- `capacity`: The capacity of the vector.

Return Value The vector.

`duckdb_destroy_vector` Destroys the vector and de-allocates its memory.

Syntax

```
            void duckdb_destroy_vector(
  duckdb_vector *vector
);
```

Parameters

- `vector`: A pointer to the vector.

`duckdb_vector_get_column_type` Retrieves the column type of the specified vector.

The result must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_vector_get_column_type(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector get the data from

Return Value The type of the vector

`duckdb_vector_get_data` Retrieves the data pointer of the vector.

The data pointer can be used to read or write values from the vector. How to read or write values depends on the type of the vector.

Syntax

```
            void *duckdb_vector_get_data(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The data pointer

`duckdb_vector_get_validity` Retrieves the validity mask pointer of the specified vector.

If all values are valid, this function MIGHT return NULL!

The validity mask is a bitset that signifies null-ness within the data chunk. It is a series of uint64_t values, where each uint64_t value contains validity for 64 tuples. The bit is set to 1 if the value is valid (i.e., not NULL) or 0 if the value is invalid (i.e., NULL).

Validity of a specific value can be obtained like this:

idx_t entry_idx = row_idx / 64; idx_t idx_in_entry = row_idx % 64; bool is_valid = validity_mask[entry_idx] & (1 « idx_in_entry);

Alternatively, the (slower) duckdb_validity_row_is_valid function can be used.

Syntax

```
            uint64_t *duckdb_vector_get_validity(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The pointer to the validity mask, or NULL if no validity mask is present

`duckdb_vector_ensure_validity_writable` Ensures the validity mask is writable by allocating it.

After this function is called, `duckdb_vector_get_validity` will ALWAYS return non-NULL. This allows NULL values to be written to the vector, regardless of whether a validity mask was present before.

Syntax

```
            void duckdb_vector_ensure_validity_writable(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to alter

`duckdb_vector_assign_string_element` Assigns a string element in the vector at the specified location.

Syntax

```
            void duckdb_vector_assign_string_element(
  duckdb_vector vector,
  idx_t index,
  const char *str
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The null-terminated string

`duckdb_vector_assign_string_element_len` Assigns a string element in the vector at the specified location. You may also use this function to assign BLOBs.

Syntax

```
            void duckdb_vector_assign_string_element_len(
  duckdb_vector vector,
  idx_t index,
  const char *str,
  idx_t str_len
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The string
- `str_len`: The length of the string (in bytes)

`duckdb_list_vector_get_child` Retrieves the child vector of a list vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_list_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_list_vector_get_size` Returns the size of the child vector of the list.

Syntax

```
            idx_t duckdb_list_vector_get_size(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The size of the child list

`duckdb_list_vector_set_size` Sets the total size of the underlying child-vector of a list vector.

Syntax

```
            duckdb_state duckdb_list_vector_set_size(
  duckdb_vector vector,
  idx_t size
);
```

Parameters

- `vector`: The list vector.
- `size`: The size of the child list.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_list_vector_reserve` Sets the total capacity of the underlying child-vector of a list.

After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            duckdb_state duckdb_list_vector_reserve(
  duckdb_vector vector,
  idx_t required_capacity
);
```

Parameters

- `vector`: The list vector.
- `required_capacity`: the total capacity to reserve.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_struct_vector_get_child` Retrieves the child vector of a struct vector. The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_struct_vector_get_child(
  duckdb_vector vector,
  idx_t index
);
```

Parameters

- `vector`: The vector
- `index`: The child index

Return Value The child vector

`duckdb_array_vector_get_child` Retrieves the child vector of an array vector. The resulting vector is valid as long as the parent vector is valid. The resulting vector has the size of the parent vector multiplied by the array size.

Syntax

```
            duckdb_vector duckdb_array_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_slice_vector` Slice a vector with a selection vector. The length of the selection vector must be less than or equal to the length of the vector. Turns the vector into a dictionary vector.

Syntax

```
            void duckdb_slice_vector(
  duckdb_vector vector,
  duckdb_selection_vector sel,
  idx_t len
);
```

Parameters

- `vector`: The vector to slice.
- `sel`: The selection vector.
- `len`: The length of the selection vector.

`duckdb_vector_copy_sel` Copy the src vector to the dst with a selection vector that identifies which indices to copy.

Syntax

```
            void duckdb_vector_copy_sel(
  duckdb_vector src,
  duckdb_vector dst,
  duckdb_selection_vector sel,
  idx_t src_count,
  idx_t src_offset,
  idx_t dst_offset
);
```

Parameters

- `src`: The vector to copy from.
- `dst`: The vector to copy to.
- `sel`: The selection vector. The length of the selection vector should not be more than the length of the src vector
- `src_count`: The number of entries from selection vector to copy. Think of this as the effective length of the selection vector starting from index 0
- `src_offset`: The offset in the selection vector to copy from (important: actual number of items copied = src_count - src_offset).
- `dst_offset`: The offset in the dst vector to start copying to.

`duckdb_vector_reference_value` Copies the value from `value` to `vector`.

Syntax

```
            void duckdb_vector_reference_value(
  duckdb_vector vector,
  duckdb_value value
);
```

Parameters

- `vector`: The receiving vector.
- `value`: The value to copy into the vector.

`duckdb_vector_reference_vector` Changes `to_vector` to reference ‘from_vector. After, the vectors share ownership of the data.

Syntax

```
            void duckdb_vector_reference_vector(
  duckdb_vector to_vector,
  duckdb_vector from_vector
);
```

Parameters

- `to_vector`: The receiving vector.
- `from_vector`: The vector to reference.

`duckdb_validity_row_is_valid` Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

Syntax

```
            bool duckdb_validity_row_is_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`
- `row`: The row index

Return Value true if the row is valid, false otherwise

`duckdb_validity_set_row_validity` In a validity mask, sets a specific row to either valid or invalid.

Note that `duckdb_vector_ensure_validity_writable` should be called before calling `duckdb_vector_get_validity`, to ensure that there is a validity mask to write to.

Syntax

```
            void duckdb_validity_set_row_validity(
  uint64_t *validity,
  idx_t row,
  bool valid
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`.
- `row`: The row index
- `valid`: Whether or not to set the row to valid, or invalid

`duckdb_validity_set_row_invalid` In a validity mask, sets a specific row to invalid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to false.

Syntax

```
            void duckdb_validity_set_row_invalid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

`duckdb_validity_set_row_valid` In a validity mask, sets a specific row to valid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```
            void duckdb_validity_set_row_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

`duckdb_create_scalar_function` Creates a new empty scalar function.

The return value must be destroyed with `duckdb_destroy_scalar_function`.

Return Value The scalar function object.

Syntax

```
            duckdb_scalar_function duckdb_create_scalar_function(

);
```

`duckdb_destroy_scalar_function` Destroys the given scalar function object.

Syntax

```
            void duckdb_destroy_scalar_function(
  duckdb_scalar_function *scalar_function
);
```

Parameters

- `scalar_function`: The scalar function to destroy

`duckdb_scalar_function_set_name` Sets the name of the given scalar function.

Syntax

```
            void duckdb_scalar_function_set_name(
  duckdb_scalar_function scalar_function,
  const char *name
);
```

Parameters

- `scalar_function`: The scalar function
- `name`: The name of the scalar function

`duckdb_scalar_function_set_varargs` Sets the parameters of the given scalar function to varargs. Does not require adding parameters with duckdb_scalar_function_add_parameter.

Syntax

```
            void duckdb_scalar_function_set_varargs(
  duckdb_scalar_function scalar_function,
  duckdb_logical_type type
);
```

Parameters

- `scalar_function`: The scalar function.
- `type`: The type of the arguments.

Return Value The parameter type. Cannot contain INVALID.

`duckdb_scalar_function_set_special_handling` Sets the scalar function's null-handling behavior to special.

Syntax

```
            void duckdb_scalar_function_set_special_handling(
  duckdb_scalar_function scalar_function
);
```

Parameters

- `scalar_function`: The scalar function.

`duckdb_scalar_function_set_volatile` Sets the Function Stability of the scalar function to VOLATILE, indicating the function should be re-run for every row. This limits optimization that can be performed for the function.

Syntax

```
            void duckdb_scalar_function_set_volatile(
  duckdb_scalar_function scalar_function
);
```

Parameters

- `scalar_function`: The scalar function.

`duckdb_scalar_function_add_parameter` Adds a parameter to the scalar function.

Syntax

```
            void duckdb_scalar_function_add_parameter(
  duckdb_scalar_function scalar_function,
  duckdb_logical_type type
);
```

Parameters

- `scalar_function`: The scalar function.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_scalar_function_set_return_type` Sets the return type of the scalar function.

Syntax

```
            void duckdb_scalar_function_set_return_type(
  duckdb_scalar_function scalar_function,
  duckdb_logical_type type
);
```

Parameters

- `scalar_function`: The scalar function
- `type`: Cannot contain INVALID or ANY.

`duckdb_scalar_function_set_extra_info` Assigns extra information to the scalar function that can be fetched during binding, etc.

Syntax

```
           void duckdb_scalar_function_set_extra_info(
  duckdb_scalar_function scalar_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `scalar_function`: The scalar function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_scalar_function_set_bind` Sets the (optional) bind function of the scalar function.

Syntax

```
           void duckdb_scalar_function_set_bind(
  duckdb_scalar_function scalar_function,
  duckdb_scalar_function_bind_t bind
);
```

Parameters

- `scalar_function`: The scalar function.
- `bind`: The bind function.

`duckdb_scalar_function_set_bind_data` Sets the user-provided bind data in the bind object of the scalar function. The bind data object can be retrieved again during execution. In most case, you also need to set the copy-callback of your bind data via duckdb_scalar_function_set_bind_data_copy.

Syntax

```
           void duckdb_scalar_function_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The bind info of the scalar function.
- `bind_data`: The bind data object.
- `destroy`: The callback to destroy the bind data (if any).

`duckdb_scalar_function_set_bind_data_copy` Sets the copy-callback for the user-provided bind data in the bind object of the scalar function.

Syntax

```
           void duckdb_scalar_function_set_bind_data_copy(
  duckdb_bind_info info,
  duckdb_copy_callback_t copy
);
```

Parameters

- `info`: The bind info of the scalar function.
- `copy`: The callback to copy the bind data (if any).

`duckdb_scalar_function_bind_set_error` Report that an error has occurred while calling bind on a scalar function.

Syntax

```
            void duckdb_scalar_function_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`: The bind info object.
- `error`: The error message.

`duckdb_scalar_function_set_function` Sets the main function of the scalar function.

Syntax

```
            void duckdb_scalar_function_set_function(
  duckdb_scalar_function scalar_function,
  duckdb_scalar_function_t function
);
```

Parameters

- `scalar_function`: The scalar function
- `function`: The function

`duckdb_register_scalar_function` Register the scalar function object within the given connection.

The function requires at least a name, a function and a return type.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_scalar_function(
  duckdb_connection con,
  duckdb_scalar_function scalar_function
);
```

Parameters

- `con`: The connection to register it in.
- `scalar_function`: The function pointer

Return Value Whether or not the registration was successful.

`duckdb_scalar_function_get_extra_info` Retrieves the extra info of the function as set in `duckdb_scalar_function_set_extra_info`.

Syntax

```
            void *duckdb_scalar_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object.

Return Value

The extra info.

`duckdb_scalar_function_bind_get_extra_info`

Retrieves the extra info of the function as set in the bind info.

Syntax

```
            void *duckdb_scalar_function_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object.

Return Value

The extra info.

`duckdb_scalar_function_get_bind_data`

Gets the scalar function's bind data set by `duckdb_scalar_function_set_bind_data`. Note that the bind data is read-only.

Syntax

```
            void *duckdb_scalar_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The function info.

Return Value

The bind data object.

`duckdb_scalar_function_get_client_context`

Retrieves the client context of the bind info of a scalar function.

Syntax

```
            void duckdb_scalar_function_get_client_context(
  duckdb_bind_info info,
  duckdb_client_context *out_context
);
```

Parameters

- `info`: The bind info object of the scalar function.
- `out_context`: The client context of the bind info. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_scalar_function_set_error` Report that an error has occurred while executing the scalar function.

Syntax

```
            void duckdb_scalar_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object.
- `error`: The error message

`duckdb_create_scalar_function_set` Creates a new empty scalar function set.

The return value must be destroyed with `duckdb_destroy_scalar_function_set`.

Return Value The scalar function set object.

Syntax

```
            duckdb_scalar_function_set duckdb_create_scalar_function_set(
  const char *name
);
```

`duckdb_destroy_scalar_function_set` Destroys the given scalar function set object.

Syntax

```
            void duckdb_destroy_scalar_function_set(
  duckdb_scalar_function_set *scalar_function_set
);
```

`duckdb_add_scalar_function_to_set` Adds the scalar function as a new overload to the scalar function set.

Returns DuckDBError if the function could not be added, for example if the overload already exists.

Syntax

```
            duckdb_state duckdb_add_scalar_function_to_set(
  duckdb_scalar_function_set set,
  duckdb_scalar_function function
);
```

Parameters

- `set`: The scalar function set
- `function`: The function to add

`duckdb_register_scalar_function_set` Register the scalar function set within the given connection.

The set requires at least a single valid overload.

If the set is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_scalar_function_set(
  duckdb_connection con,
  duckdb_scalar_function_set set
);
```

Parameters

- `con`: The connection to register it in.
- `set`: The function set to register

Return Value Whether or not the registration was successful.

`duckdb_scalar_function_bind_get_argument_count` Returns the number of input arguments of the scalar function.

Syntax

```
            idx_t duckdb_scalar_function_bind_get_argument_count(
  duckdb_bind_info info
);
```

Parameters

- `info`: The bind info.

Return Value The number of input arguments.

`duckdb_scalar_function_bind_get_argument` Returns the input argument at index of the scalar function.

Syntax

```
            duckdb_expression duckdb_scalar_function_bind_get_argument(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`: The bind info.
- `index`: The argument index.

Return Value The input argument at index. Must be destroyed with `duckdb_destroy_expression`.

`duckdb_create_selection_vector` Creates a new selection vector of size `size`. Must be destroyed with `duckdb_destroy_selection_vector`.

Syntax

```
            duckdb_selection_vector duckdb_create_selection_vector(
  idx_t size
);
```

Parameters

- `size`: The size of the selection vector.

Return Value The selection vector.

`duckdb_destroy_selection_vector` Destroys the selection vector and de-allocates its memory.

Syntax

```
            void duckdb_destroy_selection_vector(
  duckdb_selection_vector sel
);
```

Parameters

- `sel`: The selection vector.

`duckdb_selection_vector_get_data_ptr` Access the data pointer of a selection vector.

Syntax

```
            sel_t *duckdb_selection_vector_get_data_ptr(
  duckdb_selection_vector sel
);
```

Parameters

- `sel`: The selection vector.

Return Value The data pointer.

`duckdb_create_aggregate_function` Creates a new empty aggregate function.

The return value should be destroyed with `duckdb_destroy_aggregate_function`.

Return Value The aggregate function object.

Syntax

```
            duckdb_aggregate_function duckdb_create_aggregate_function(

);
```

`duckdb_destroy_aggregate_function` Destroys the given aggregate function object.

Syntax

```
            void duckdb_destroy_aggregate_function(
  duckdb_aggregate_function *aggregate_function
);
```

`duckdb_aggregate_function_set_name` Sets the name of the given aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_name(
  duckdb_aggregate_function aggregate_function,
  const char *name
);
```

Parameters

- `aggregate_function`: The aggregate function
- `name`: The name of the aggregate function

`duckdb_aggregate_function_add_parameter` Adds a parameter to the aggregate function.

Syntax

```
            void duckdb_aggregate_function_add_parameter(
  duckdb_aggregate_function aggregate_function,
  duckdb_logical_type type
);
```

Parameters

- `aggregate_function`: The aggregate function.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_aggregate_function_set_return_type` Sets the return type of the aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_return_type(
  duckdb_aggregate_function aggregate_function,
  duckdb_logical_type type
);
```

Parameters

- `aggregate_function`: The aggregate function.
- `type`: The return type. Cannot contain INVALID or ANY.

`duckdb_aggregate_function_set_functions` Sets the main functions of the aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_functions(
  duckdb_aggregate_function aggregate_function,
  duckdb_aggregate_state_size state_size,
  duckdb_aggregate_init_t state_init,
  duckdb_aggregate_update_t update,
  duckdb_aggregate_combine_t combine,
  duckdb_aggregate_finalize_t finalize
);
```

Parameters

- `aggregate_function`: The aggregate function
- `state_size`: state size
- `state_init`: state init function
- `update`: update states
- `combine`: combine states
- `finalize`: finalize states

`duckdb_aggregate_function_set_destructor` Sets the state destructor callback of the aggregate function (optional)

Syntax

```
            void duckdb_aggregate_function_set_destructor(
  duckdb_aggregate_function aggregate_function,
  duckdb_aggregate_destroy_t destroy
);
```

Parameters

- `aggregate_function`: The aggregate function
- `destroy`: state destroy callback

`duckdb_register_aggregate_function` Register the aggregate function object within the given connection.

The function requires at least a name, functions and a return type.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_aggregate_function(
  duckdb_connection con,
  duckdb_aggregate_function aggregate_function
);
```

Parameters

- `con`: The connection to register it in.

Return Value Whether or not the registration was successful.

`duckdb_aggregate_function_set_special_handling` Sets the NULL handling of the aggregate function to SPECIAL_HANDLING.

Syntax

```
            void duckdb_aggregate_function_set_special_handling(
  duckdb_aggregate_function aggregate_function
);
```

Parameters

- `aggregate_function`: The aggregate function

`duckdb_aggregate_function_set_extra_info` Assigns extra information to the scalar function that can be fetched during binding, etc.

Syntax

```
            void duckdb_aggregate_function_set_extra_info(
  duckdb_aggregate_function aggregate_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `aggregate_function`: The aggregate function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_aggregate_function_get_extra_info` Retrieves the extra info of the function as set in `duckdb_aggregate_function_set_extra_info`.

Syntax

```
            void *duckdb_aggregate_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_aggregate_function_set_error` Report that an error has occurred while executing the aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_create_aggregate_function_set` Creates a new empty aggregate function set.

The return value should be destroyed with `duckdb_destroy_aggregate_function_set`.

Return Value The aggregate function set object.

Syntax

```
            duckdb_aggregate_function_set duckdb_create_aggregate_function_set(
  const char *name
);
```

`duckdb_destroy_aggregate_function_set` Destroys the given aggregate function set object.

Syntax

```
            void duckdb_destroy_aggregate_function_set(
  duckdb_aggregate_function_set *aggregate_function_set
);
```

`duckdb_add_aggregate_function_to_set` Adds the aggregate function as a new overload to the aggregate function set.

Returns DuckDBError if the function could not be added, for example if the overload already exists.

Syntax

```
            duckdb_state duckdb_add_aggregate_function_to_set(
  duckdb_aggregate_function_set set,
  duckdb_aggregate_function function
);
```

Parameters

- `set`: The aggregate function set
- `function`: The function to add

`duckdb_register_aggregate_function_set` Register the aggregate function set within the given connection.

The set requires at least a single valid overload.

If the set is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_aggregate_function_set(
  duckdb_connection con,
  duckdb_aggregate_function_set set
);
```

Parameters

- `con`: The connection to register it in.
- `set`: The function set to register

Return Value Whether or not the registration was successful.

`duckdb_create_table_function` Creates a new empty table function.

The return value should be destroyed with `duckdb_destroy_table_function`.

Return Value The table function object.

Syntax

```
duckdb_table_function duckdb_create_table_function(

);
```

`duckdb_destroy_table_function` Destroys the given table function object.

Syntax

```
void duckdb_destroy_table_function(
  duckdb_table_function *table_function
);
```

Parameters

- `table_function`: The table function to destroy

`duckdb_table_function_set_name` Sets the name of the given table function.

Syntax

```
void duckdb_table_function_set_name(
  duckdb_table_function table_function,
  const char *name
);
```

Parameters

- `table_function`: The table function
- `name`: The name of the table function

`duckdb_table_function_add_parameter` Adds a parameter to the table function.

Syntax

```
void duckdb_table_function_add_parameter(
  duckdb_table_function table_function,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_table_function_add_named_parameter` Adds a named parameter to the table function.

Syntax

```
            void duckdb_table_function_add_named_parameter(
  duckdb_table_function table_function,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `name`: The parameter name.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_table_function_set_extra_info` Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```
            void duckdb_table_function_set_extra_info(
  duckdb_table_function table_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `table_function`: The table function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_table_function_set_bind` Sets the bind function of the table function.

Syntax

```
            void duckdb_table_function_set_bind(
  duckdb_table_function table_function,
  duckdb_table_function_bind_t bind
);
```

Parameters

- `table_function`: The table function
- `bind`: The bind function

`duckdb_table_function_set_init` Sets the init function of the table function.

Syntax

```
            void duckdb_table_function_set_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_local_init` Sets the thread-local init function of the table function.

Syntax

```
            void duckdb_table_function_set_local_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_function` Sets the main function of the table function.

Syntax

```
            void duckdb_table_function_set_function(
  duckdb_table_function table_function,
  duckdb_table_function_t function
);
```

Parameters

- `table_function`: The table function
- `function`: The function

`duckdb_table_function_supports_projection_pushdown` Sets whether or not the given table function supports projection pushdown.

If this is set to true, the system will provide a list of all required columns in the `init` stage through the `duckdb_init_get_column_count` and `duckdb_init_get_column_index` functions. If this is set to false (the default), the system will expect all columns to be projected.

Syntax

```
            void duckdb_table_function_supports_projection_pushdown(
  duckdb_table_function table_function,
  bool pushdown
);
```

Parameters

- `table_function`: The table function
- `pushdown`: True if the table function supports projection pushdown, false otherwise.

`duckdb_register_table_function` Register the table function object within the given connection.

The function requires at least a name, a bind function, an init function and a main function.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_table_function(
  duckdb_connection con,
  duckdb_table_function function
);
```

Parameters

- `con`: The connection to register it in.
- `function`: The function pointer

Return Value Whether or not the registration was successful.

`duckdb_bind_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_table_function_get_client_context` Retrieves the client context of the bind info of a table function.

Syntax

```
            void duckdb_table_function_get_client_context(
  duckdb_bind_info info,
  duckdb_client_context *out_context
);
```

Parameters

- `info`: The bind info object of the table function.
- `out_context`: The client context of the bind info. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_bind_add_result_column` Adds a result column to the output of the table function.

Syntax

```
            void duckdb_bind_add_result_column(
  duckdb_bind_info info,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `info`: The table function's bind info.
- `name`: The column name.
- `type`: The logical column type.

`duckdb_bind_get_parameter_count` Retrieves the number of regular (non-named) parameters to the function.

Syntax

```
            idx_t duckdb_bind_get_parameter_count(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The number of parameters

`duckdb_bind_get_parameter` Retrieves the parameter at the given index.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_parameter(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`: The info object
- `index`: The index of the parameter to get

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_get_named_parameter` Retrieves a named parameter with the given name.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_named_parameter(
  duckdb_bind_info info,
  const char *name
);
```

Parameters

- `info`: The info object
- `name`: The name of the parameter

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_set_bind_data` Sets the user-provided bind data in the bind object of the table function. This object can be retrieved again during execution.

Syntax

```
            void duckdb_bind_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The bind info of the table function.
- `bind_data`: The bind data object.
- `destroy`: The callback to destroy the bind data (if any).

`duckdb_bind_set_cardinality` Sets the cardinality estimate for the table function, used for optimization.

Syntax

```
            void duckdb_bind_set_cardinality(
  duckdb_bind_info info,
  idx_t cardinality,
  bool is_exact
);
```

Parameters

- `info`: The bind data object.
- `is_exact`: Whether or not the cardinality estimate is exact, or an approximation

`duckdb_bind_set_error` Report that an error has occurred while calling bind on a table function.

Syntax

```
            void duckdb_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_init_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_init_get_extra_info(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_init_get_bind_data` Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_init_get_bind_data(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The bind data object

`duckdb_init_set_init_data` Sets the user-provided init data in the init object. This object can be retrieved again during execution.

Syntax

```
            void duckdb_init_set_init_data(
  duckdb_init_info info,
  void *init_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The info object
- `init_data`: The init data object.
- `destroy`: The callback that will be called to destroy the init data (if any)

`duckdb_init_get_column_count` Returns the number of projected columns.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_count(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The number of projected columns.

`duckdb_init_get_column_index` Returns the column index of the projected column at the specified position.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_index(
  duckdb_init_info info,
  idx_t column_index
);
```

Parameters

- `info`: The info object
- `column_index`: The index at which to get the projected column index, from 0..duckdb_init_get_column_count(info)

Return Value The column index of the projected column.

`duckdb_init_set_max_threads` Sets how many threads can process this table function in parallel (default: 1)

Syntax

```
            void duckdb_init_set_max_threads(
  duckdb_init_info info,
  idx_t max_threads
);
```

Parameters

- `info`: The info object
- `max_threads`: The maximum amount of threads that can process this table function

`duckdb_init_set_error` Report that an error has occurred while calling init.

Syntax

```
            void duckdb_init_set_error(
  duckdb_init_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_function_get_extra_info` Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value

The extra info

`duckdb_function_get_bind_data` Gets the table function's bind data set by `duckdb_bind_set_bind_data`.

Note that the bind data is read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The function info object.

Return Value

The bind data object.

`duckdb_function_get_init_data` Gets the init data set by `duckdb_init_set_init_data` during the init.

Syntax

```
            void *duckdb_function_get_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value

The init data object

`duckdb_function_get_local_init_data` Gets the thread-local init data set by `duckdb_init_set_init_data` during the local_init.

Syntax

```
            void *duckdb_function_get_local_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The init data object

`duckdb_function_set_error`

Report that an error has occurred while executing the function.

Syntax

```
            void duckdb_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_add_replacement_scan`

Add a replacement scan definition to the specified database.

Syntax

```
            void duckdb_add_replacement_scan(
  duckdb_database db,
  duckdb_replacement_callback_t replacement,
  void *extra_data,
  duckdb_delete_callback_t delete_callback
);
```

Parameters

- db: The database object to add the replacement scan to
- `replacement`: The replacement scan callback
- `extra_data`: Extra data that is passed back into the specified callback
- `delete_callback`: The delete callback to call on the extra data, if any

`duckdb_replacement_scan_set_function_name`

Sets the replacement function name. If this function is called in the replacement callback, the replacement scan is performed. If it is not called, the replacement callback is not performed.

Syntax

```
            void duckdb_replacement_scan_set_function_name(
  duckdb_replacement_scan_info info,
  const char *function_name
);
```

Parameters

- `info`: The info object
- `function_name`: The function name to substitute.

`duckdb_replacement_scan_add_parameter` Adds a parameter to the replacement scan function.

Syntax

```
           void duckdb_replacement_scan_add_parameter(
  duckdb_replacement_scan_info info,
  duckdb_value parameter
);
```

Parameters

- `info`: The info object
- `parameter`: The parameter to add.

`duckdb_replacement_scan_set_error` Report that an error has occurred while executing the replacement scan.

Syntax

```
           void duckdb_replacement_scan_set_error(
  duckdb_replacement_scan_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_get_profiling_info` Returns the root node of the profiling information. Returns nullptr, if profiling is not enabled.

Syntax

```
           duckdb_profiling_info duckdb_get_profiling_info(
  duckdb_connection connection
);
```

Parameters

- `connection`: A connection object.

Return Value A profiling information object.

`duckdb_profiling_info_get_value` Returns the value of the metric of the current profiling info node. Returns nullptr, if the metric does not exist or is not enabled. Currently, the value holds a string, and you can retrieve the string by calling the corresponding function: char *duckdb_get_varchar(duckdb_value value).

Syntax

```
            duckdb_value duckdb_profiling_info_get_value(
  duckdb_profiling_info info,
  const char *key
);
```

Parameters

- `info`: A profiling information object.
- `key`: The name of the requested metric.

Return Value The value of the metric. Must be freed with `duckdb_destroy_value`

`duckdb_profiling_info_get_metrics` Returns the key-value metric map of this profiling node as a MAP duckdb_value. The individual elements are accessible via the duckdb_value MAP functions.

Syntax

```
            duckdb_value duckdb_profiling_info_get_metrics(
  duckdb_profiling_info info
);
```

Parameters

- `info`: A profiling information object.

Return Value The key-value metric map as a MAP duckdb_value.

`duckdb_profiling_info_get_child_count` Returns the number of children in the current profiling info node.

Syntax

```
            idx_t duckdb_profiling_info_get_child_count(
  duckdb_profiling_info info
);
```

Parameters

- `info`: A profiling information object.

Return Value The number of children in the current node.

`duckdb_profiling_info_get_child` Returns the child node at the specified index.

Syntax

```
            duckdb_profiling_info duckdb_profiling_info_get_child(
  duckdb_profiling_info info,
  idx_t index
);
```

Parameters

- `info`: A profiling information object.
- `index`: The index of the child node.

Return Value The child node at the specified index.

`duckdb_appender_create` Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
           duckdb_state duckdb_appender_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_create_ext` Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
           duckdb_state duckdb_appender_create_ext(
  duckdb_connection connection,
  const char *catalog,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `catalog`: The catalog of the table to append to, or `nullptr` for the default catalog.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_create_query` Creates an appender object that executes the given query with any data appended to it.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create_query(
  duckdb_connection connection,
  const char *query,
  idx_t column_count,
  duckdb_logical_type *types,
  const char *table_name,
  const char **column_names,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `query`: The query to execute, can be an INSERT, DELETE, UPDATE or MERGE INTO statement.
- `column_count`: The number of columns to append.
- `types`: The types of the columns to append.
- `table_name`: (optionally) the table name used to refer to the appended data, defaults to "appended_data".
- `column_names`: (optionally) the list of column names, defaults to "col1", "col2", ...
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_column_count` Returns the number of columns that belong to the appender. If there is no active column list, then this equals the table's physical columns.

Syntax

```
            idx_t duckdb_appender_column_count(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the column count from.

Return Value The number of columns in the data chunks.

`duckdb_appender_column_type` Returns the type of the column at the specified index. This is either a type in the active column list, or the same type as a column in the receiving table.

Note: The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_appender_column_type(
  duckdb_appender appender,
  idx_t col_idx
);
```

Parameters

- `appender`: The appender to get the column type from.
- `col_idx`: The index of the column to get the type of.

Return Value The `duckdb_logical_type` of the column.

`duckdb_appender_error`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release. Use duckdb_appender_error_data instead.

Returns the error message associated with the appender. If the appender has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_appender_destroy` is called.

Syntax

```
            const char *duckdb_appender_error(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_appender_error_data` Returns the error data associated with the appender. Must be destroyed with duckdb_destroy_error_data.

Syntax

```
            duckdb_error_data duckdb_appender_error_data(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the error data from.

Return Value The error data.

`duckdb_appender_flush` Flush the appender to the table, forcing the cache of the appender to be cleared. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. It is not possible to append more values. Call duckdb_appender_error_data to obtain the error data followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_flush(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_close` Closes the appender by flushing all intermediate states and closing it for further appends. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. Call duckdb_appender_error_data to obtain the error data followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
duckdb_state duckdb_appender_close(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush and close.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_destroy` Closes the appender by flushing all intermediate states to the table and destroying it. By destroying it, this function de-allocates all memory associated with the appender. If flushing the data triggers a constraint violation, then all data is invalidated, and this function returns DuckDBError. Due to the destruction of the appender, it is no longer possible to obtain the specific error message with duckdb_appender_error. Therefore, call duckdb_appender_close before destroying the appender, if you need insights into the specific error.

Syntax

```
duckdb_state duckdb_appender_destroy(
  duckdb_appender *appender
);
```

Parameters

- `appender`: The appender to flush, close and destroy.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_add_column` Appends a column to the active column list of the appender. Immediately flushes all previous data.

The active column list specifies all columns that are expected when flushing the data. Any non-active columns are filled with their default values, or NULL.

Syntax

```
duckdb_state duckdb_appender_add_column(
  duckdb_appender appender,
  const char *name
);
```

Parameters

- `appender`: The appender to add the column to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_clear_columns` Removes all columns from the active column list of the appender, resetting the appender to treat all columns as active. Immediately flushes all previous data.

Syntax

```
            duckdb_state duckdb_appender_clear_columns(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to clear the columns from.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_begin_row` A nop function, provided for backwards compatibility reasons. Does nothing. Only `duckdb_appender_end_row` is required.

Syntax

```
            duckdb_state duckdb_appender_begin_row(
  duckdb_appender appender
);
```

`duckdb_appender_end_row` Finish the current row of appends. After end_row is called, the next row can be appended.

Syntax

```
            duckdb_state duckdb_appender_end_row(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_default` Append a DEFAULT value (NULL if DEFAULT not available for column) to the appender.

Syntax

```
            duckdb_state duckdb_append_default(
  duckdb_appender appender
);
```

`duckdb_append_default_to_chunk` Append a DEFAULT value, at the specified row and column, (NULL if DEFAULT not available for column) to the chunk created from the specified appender. The default value of the column must be a constant value. Non-deterministic expressions like nextval('seq') or random() are not supported.

Syntax

```
            duckdb_state duckdb_append_default_to_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk,
  idx_t col,
  idx_t row
);
```

Parameters

- `appender`: The appender to get the default value from.
- `chunk`: The data chunk to append the default value to.
- `col`: The chunk column index to append the default value to.
- `row`: The chunk row index to append the default value to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_bool` Append a bool value to the appender.

Syntax

```
            duckdb_state duckdb_append_bool(
  duckdb_appender appender,
  bool value
);
```

`duckdb_append_int8` Append an int8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int8(
  duckdb_appender appender,
  int8_t value
);
```

`duckdb_append_int16` Append an int16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int16(
  duckdb_appender appender,
  int16_t value
);
```

`duckdb_append_int32` Append an int32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int32(
  duckdb_appender appender,
  int32_t value
);
```

`duckdb_append_int64` Append an int64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int64(
  duckdb_appender appender,
  int64_t value
);
```

`duckdb_append_hugeint` Append a duckdb_hugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_hugeint(
  duckdb_appender appender,
  duckdb_hugeint value
);
```

`duckdb_append_uint8` Append a uint8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint8(
  duckdb_appender appender,
  uint8_t value
);
```

`duckdb_append_uint16` Append a uint16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint16(
  duckdb_appender appender,
  uint16_t value
);
```

`duckdb_append_uint32` Append a uint32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint32(
  duckdb_appender appender,
  uint32_t value
);
```

`duckdb_append_uint64` Append a uint64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint64(
  duckdb_appender appender,
  uint64_t value
);
```

`duckdb_append_uhugeint` Append a duckdb_uhugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_uhugeint(
  duckdb_appender appender,
  duckdb_uhugeint value
);
```

`duckdb_append_float` Append a float value to the appender.

Syntax

```
            duckdb_state duckdb_append_float(
  duckdb_appender appender,
  float value
);
```

`duckdb_append_double` Append a double value to the appender.

Syntax

```
            duckdb_state duckdb_append_double(
  duckdb_appender appender,
  double value
);
```

`duckdb_append_date` Append a duckdb_date value to the appender.

Syntax

```
            duckdb_state duckdb_append_date(
  duckdb_appender appender,
  duckdb_date value
);
```

`duckdb_append_time` Append a duckdb_time value to the appender.

Syntax

```
            duckdb_state duckdb_append_time(
  duckdb_appender appender,
  duckdb_time value
);
```

`duckdb_append_timestamp` Append a duckdb_timestamp value to the appender.

Syntax

```
            duckdb_state duckdb_append_timestamp(
  duckdb_appender appender,
  duckdb_timestamp value
);
```

`duckdb_append_interval` Append a duckdb_interval value to the appender.

Syntax

```
            duckdb_state duckdb_append_interval(
  duckdb_appender appender,
  duckdb_interval value
);
```

`duckdb_append_varchar` Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar(
  duckdb_appender appender,
  const char *val
);
```

`duckdb_append_varchar_length` Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar_length(
  duckdb_appender appender,
  const char *val,
  idx_t length
);
```

`duckdb_append_blob` Append a blob value to the appender.

Syntax

```
            duckdb_state duckdb_append_blob(
  duckdb_appender appender,
  const void *data,
  idx_t length
);
```

`duckdb_append_null` Append a NULL value to the appender (of any type).

Syntax

```
            duckdb_state duckdb_append_null(
  duckdb_appender appender
);
```

`duckdb_append_value` Append a duckdb_value to the appender.

Syntax

```
            duckdb_state duckdb_append_value(
  duckdb_appender appender,
  duckdb_value value
);
```

`duckdb_append_data_chunk` Appends a pre-filled data chunk to the specified appender. Attempts casting, if the data chunk types do not match the active appender types.

Syntax

```
            duckdb_state duckdb_append_data_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk
);
```

Parameters

- `appender`: The appender to append to.
- `chunk`: The data chunk to append.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_create` Creates a table description object. Note that `duckdb_table_description_destroy` should always be called on the resulting table_description, even if the function returns `DuckDBError`.

Syntax

```
            duckdb_state duckdb_table_description_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_table_description *out
);
```

Parameters

- `connection`: The connection context.
- `schema`: The schema of the table, or `nullptr` for the default schema.
- `table`: The table name.
- `out`: The resulting table description object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_create_ext` Creates a table description object. Note that `duckdb_table_description_destroy` must be called on the resulting table_description, even if the function returns `DuckDBError`.

Syntax

```
            duckdb_state duckdb_table_description_create_ext(
  duckdb_connection connection,
  const char *catalog,
  const char *schema,
  const char *table,
  duckdb_table_description *out
);
```

Parameters

- `connection`: The connection context.
- `catalog`: The catalog (database) name of the table, or `nullptr` for the default catalog.
- `schema`: The schema of the table, or `nullptr` for the default schema.
- `table`: The table name.
- `out`: The resulting table description object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_destroy` Destroy the TableDescription object.

Syntax

```
            void duckdb_table_description_destroy(
  duckdb_table_description *table_description
);
```

Parameters

- `table_description`: The table_description to destroy.

`duckdb_table_description_error` Returns the error message associated with the given table_description. If the table_description has no error message, this returns `nullptr` instead. The error message should not be freed. It will be de-allocated when `duckdb_table_description_destroy` is called.

Syntax

```
            const char *duckdb_table_description_error(
  duckdb_table_description table_description
);
```

Parameters

- `table_description`: The table_description to get the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_column_has_default` Check if the column at 'index' index of the table has a DEFAULT expression.

Syntax

```
           duckdb_state duckdb_column_has_default(
  duckdb_table_description table_description,
  idx_t index,
  bool *out
);
```

Parameters

- `table_description`: The table_description to query.
- `index`: The index of the column to query.
- `out`: The out-parameter used to store the result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_get_column_name` Obtain the column name at 'index'. The out result must be destroyed with `duckdb_free`.

Syntax

```
           char *duckdb_table_description_get_column_name(
  duckdb_table_description table_description,
  idx_t index
);
```

Parameters

- `table_description`: The table_description to query.
- `index`: The index of the column to query.

Return Value The column name.

`duckdb_to_arrow_schema` Transforms a DuckDB Schema into an Arrow Schema

Syntax

```
           duckdb_error_data duckdb_to_arrow_schema(
  duckdb_arrow_options arrow_options,
  duckdb_logical_type *types,
  const char **names,
  idx_t column_count,
  struct ArrowSchema *out_schema
);
```

Parameters

- `arrow_options`: The Arrow settings used to produce arrow.
- `types`: The DuckDB logical types for each column in the schema.
- `names`: The names for each column in the schema.
- `column_count`: The number of columns that exist in the schema.
- `out_schema`: The resulting arrow schema. Must be destroyed with `out_schema->release(out_schema)`.

Return Value The error data. Must be destroyed with `duckdb_destroy_error_data`.

`duckdb_data_chunk_to_arrow` Transforms a DuckDB data chunk into an Arrow array.

Syntax

```
            duckdb_error_data duckdb_data_chunk_to_arrow(
  duckdb_arrow_options arrow_options,
  duckdb_data_chunk chunk,
  struct ArrowArray *out_arrow_array
);
```

Parameters

- `arrow_options`: The Arrow settings used to produce arrow.
- `chunk`: The DuckDB data chunk to convert.
- `out_arrow_array`: The output Arrow structure that will hold the converted data. Must be released with `out_arrow_array->release(out_arrow_array)`

Return Value The error data. Must be destroyed with `duckdb_destroy_error_data`.

`duckdb_schema_from_arrow` Transforms an Arrow Schema into a DuckDB Schema.

Syntax

```
            duckdb_error_data duckdb_schema_from_arrow(
  duckdb_connection connection,
  struct ArrowSchema *schema,
  duckdb_arrow_converted_schema *out_types
);
```

Parameters

- `connection`: The connection to get the transformation settings from.
- `schema`: The input Arrow schema. Must be released with `schema->release(schema)`.
- `out_types`: The Arrow converted schema with extra information about the arrow types. Must be destroyed with `duckdb_destroy_arrow_converted_schema`.

Return Value The error data. Must be destroyed with `duckdb_destroy_error_data`.

`duckdb_data_chunk_from_arrow` Transforms an Arrow array into a DuckDB data chunk. The data chunk will retain ownership of the underlying Arrow data.

Syntax

```
            duckdb_error_data duckdb_data_chunk_from_arrow(
  duckdb_connection connection,
  struct ArrowArray *arrow_array,
  duckdb_arrow_converted_schema converted_schema,
  duckdb_data_chunk *out_chunk
);
```

Parameters

- `connection`: The connection to get the transformation settings from.
- `arrow_array`: The input Arrow array. Data ownership is passed on to DuckDB's DataChunk, the underlying object does not need to be released and won't have ownership of the data.
- `converted_schema`: The Arrow converted schema with extra information about the arrow types.
- `out_chunk`: The resulting DuckDB data chunk. Must be destroyed by duckdb_destroy_data_chunk.

Return Value The error data. Must be destroyed with `duckdb_destroy_error_data`.

`duckdb_destroy_arrow_converted_schema` Destroys the arrow converted schema and de-allocates all memory allocated for that arrow converted schema.

Syntax

```
            void duckdb_destroy_arrow_converted_schema(
  duckdb_arrow_converted_schema *arrow_converted_schema
);
```

Parameters

- `arrow_converted_schema`: The arrow converted schema to destroy.

`duckdb_query_arrow`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes a SQL query within a connection and stores the full (materialized) result in an arrow structure. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_query_arrow_error`.

Note that after running `duckdb_query_arrow`, `duckdb_destroy_arrow` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
            duckdb_state duckdb_query_arrow(
  duckdb_connection connection,
  const char *query,
  duckdb_arrow *out_result
);
```

Parameters

- `connection`: The connection to perform the query in.
- `query`: The SQL query to run.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_query_arrow_schema`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetch the internal arrow schema from the arrow result. Remember to call release on the respective ArrowSchema object.

Syntax

```
           duckdb_state duckdb_query_arrow_schema(
  duckdb_arrow result,
  duckdb_arrow_schema *out_schema
);
```

Parameters

- `result`: The result to fetch the schema from.
- `out_schema`: The output schema.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_prepared_arrow_schema`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetch the internal arrow schema from the prepared statement. Remember to call release on the respective ArrowSchema object.

Syntax

```
           duckdb_state duckdb_prepared_arrow_schema(
  duckdb_prepared_statement prepared,
  duckdb_arrow_schema *out_schema
);
```

Parameters

- `prepared`: The prepared statement to fetch the schema from.
- `out_schema`: The output schema.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_result_arrow_array`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Convert a data chunk into an arrow struct array. Remember to call release on the respective ArrowArray object.

Syntax

```
           void duckdb_result_arrow_array(
  duckdb_result result,
  duckdb_data_chunk chunk,
  duckdb_arrow_array *out_array
);
```

Parameters

- `result`: The result object the data chunk have been fetched from.
- `chunk`: The data chunk to convert.
- `out_array`: The output array.

`duckdb_query_arrow_array`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetch an internal arrow struct array from the arrow result. Remember to call release on the respective ArrowArray object.

This function can be called multiple time to get next chunks, which will free the previous out_array. So consume the out_array before calling this function again.

Syntax

```
            duckdb_state duckdb_query_arrow_array(
  duckdb_arrow result,
  duckdb_arrow_array *out_array
);
```

Parameters

- `result`: The result to fetch the array from.
- `out_array`: The output array.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_column_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of columns present in the arrow result object.

Syntax

```
            idx_t duckdb_arrow_column_count(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object.

Return Value The number of columns present in the result object.

`duckdb_arrow_row_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows present in the arrow result object.

Syntax

```
            idx_t duckdb_arrow_row_count(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows present in the result object.

`duckdb_arrow_rows_changed`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows changed by the query stored in the arrow result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
            idx_t duckdb_arrow_rows_changed(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows changed.

`duckdb_query_arrow_error`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the error message contained within the result. The error is only set if `duckdb_query_arrow` returns `DuckDBError`.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_arrow` is called.

Syntax

```
            const char *duckdb_query_arrow_error(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error of the result.

`duckdb_destroy_arrow`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Closes the result and de-allocates all memory allocated for the arrow result.

Syntax

```
            void duckdb_destroy_arrow(
  duckdb_arrow *result
);
```

Parameters

- `result`: The result to destroy.

duckdb_destroy_arrow_stream

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Releases the arrow array stream and de-allocates its memory.

Syntax

```
            void duckdb_destroy_arrow_stream(
  duckdb_arrow_stream *stream_p
);
```

Parameters

- `stream_p`: The arrow array stream to destroy.

duckdb_execute_prepared_arrow

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns an arrow query result. Note that after running `duckdb_execute_prepared_arrow`, `duckdb_destroy_arrow` must be called on the result object.

Syntax

```
            duckdb_state duckdb_execute_prepared_arrow(
  duckdb_prepared_statement prepared_statement,
  duckdb_arrow *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_arrow_scan

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Scans the Arrow stream and creates a view with the given name.

Syntax

```
            duckdb_state duckdb_arrow_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_stream arrow
);
```

Parameters

- `connection`: The connection on which to execute the scan.
- `table_name`: Name of the temporary view to create.
- `arrow`: Arrow stream wrapper.

Return Value DuckDBSuccess on success or DuckDBError on failure.

duckdb_arrow_array_scan

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Scans the Arrow array and creates a view with the given name. Note that after running duckdb_arrow_array_scan, duckdb_destroy_arrow_stream must be called on the out stream.

Syntax

```
            duckdb_state duckdb_arrow_array_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_schema arrow_schema,
  duckdb_arrow_array arrow_array,
  duckdb_arrow_stream *out_stream
);
```

Parameters

- connection: The connection on which to execute the scan.
- table_name: Name of the temporary view to create.
- arrow_schema: Arrow schema wrapper.
- arrow_array: Arrow array wrapper.
- out_stream: Output array stream that wraps around the passed schema, for releasing/deleting once done.

Return Value DuckDBSuccess on success or DuckDBError on failure.

duckdb_execute_tasks

Execute DuckDB tasks on this thread.

Will return after max_tasks have been executed, or if there are no more tasks present.

Syntax

```
            void duckdb_execute_tasks(
  duckdb_database database,
  idx_t max_tasks
);
```

Parameters

- database: The database object to execute tasks for
- max_tasks: The maximum amount of tasks to execute

duckdb_create_task_state

Creates a task state that can be used with duckdb_execute_tasks_state to execute tasks until duckdb_finish_execution is called on the state.

duckdb_destroy_state must be called on the result.

Syntax

```
            duckdb_task_state duckdb_create_task_state(
  duckdb_database database
);
```

Parameters

- `database`: The database object to create the task state for

Return Value The task state that can be used with duckdb_execute_tasks_state.

`duckdb_execute_tasks_state` Execute DuckDB tasks on this thread.

The thread will keep on executing tasks forever, until duckdb_finish_execution is called on the state. Multiple threads can share the same duckdb_task_state.

Syntax

```
            void duckdb_execute_tasks_state(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state of the executor

`duckdb_execute_n_tasks_state` Execute DuckDB tasks on this thread.

The thread will keep on executing tasks until either duckdb_finish_execution is called on the state, max_tasks tasks have been executed or there are no more tasks to be executed.

Multiple threads can share the same duckdb_task_state.

Syntax

```
            idx_t duckdb_execute_n_tasks_state(
  duckdb_task_state state,
  idx_t max_tasks
);
```

Parameters

- `state`: The task state of the executor
- `max_tasks`: The maximum amount of tasks to execute

Return Value The amount of tasks that have actually been executed

`duckdb_finish_execution` Finish execution on a specific task.

Syntax

```
            void duckdb_finish_execution(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state to finish execution

`duckdb_task_state_is_finished` Check if the provided duckdb_task_state has finished execution

Syntax

```
            bool duckdb_task_state_is_finished(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state to inspect

Return Value Whether or not duckdb_finish_execution has been called on the task state

`duckdb_destroy_task_state` Destroys the task state returned from duckdb_create_task_state.

Note that this should not be called while there is an active duckdb_execute_tasks_state running on the task state.

Syntax

```
            void duckdb_destroy_task_state(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state to clean up

`duckdb_execution_is_finished` Returns true if the execution of the current query is finished.

Syntax

```
            bool duckdb_execution_is_finished(
  duckdb_connection con
);
```

Parameters

- `con`: The connection on which to check

`duckdb_stream_fetch_chunk`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetches a data chunk from the (streaming) duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function can only be used on duckdb_results created with 'duckdb_pending_prepared_streaming'

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions or the materialized result functions).

It is not known beforehand how many chunks will be returned by this result.

Syntax

```
            duckdb_data_chunk duckdb_stream_fetch_chunk(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the data chunk from.

Return Value The resulting data chunk. Returns `NULL` if the result has an error.

`duckdb_fetch_chunk` Fetches a data chunk from a duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

It is not known beforehand how many chunks will be returned by this result.

Syntax

```
            duckdb_data_chunk duckdb_fetch_chunk(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the data chunk from.

Return Value The resulting data chunk. Returns `NULL` if the result has an error.

`duckdb_create_cast_function` Creates a new cast function object.

Return Value The cast function object.

Syntax

```
            duckdb_cast_function duckdb_create_cast_function(

);
```

`duckdb_cast_function_set_source_type` Sets the source type of the cast function.

Syntax

```
            void duckdb_cast_function_set_source_type(
  duckdb_cast_function cast_function,
  duckdb_logical_type source_type
);
```

Parameters

- `cast_function`: The cast function object.
- `source_type`: The source type to set.

`duckdb_cast_function_set_target_type` Sets the target type of the cast function.

Syntax

```
            void duckdb_cast_function_set_target_type(
  duckdb_cast_function cast_function,
  duckdb_logical_type target_type
);
```

Parameters

- `cast_function`: The cast function object.
- `target_type`: The target type to set.

`duckdb_cast_function_set_implicit_cast_cost` Sets the "cost" of implicitly casting the source type to the target type using this function.

Syntax

```
            void duckdb_cast_function_set_implicit_cast_cost(
  duckdb_cast_function cast_function,
  int64_t cost
);
```

Parameters

- `cast_function`: The cast function object.
- `cost`: The cost to set.

`duckdb_cast_function_set_function` Sets the actual cast function to use.

Syntax

```
            void duckdb_cast_function_set_function(
  duckdb_cast_function cast_function,
  duckdb_cast_function_t function
);
```

Parameters

- `cast_function`: The cast function object.
- `function`: The function to set.

`duckdb_cast_function_set_extra_info` Assigns extra information to the cast function that can be fetched during execution, etc.

Syntax

```
            void duckdb_cast_function_set_extra_info(
  duckdb_cast_function cast_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_cast_function_get_extra_info` Retrieves the extra info of the function as set in `duckdb_cast_function_set_extra_info`.

Syntax

```
            void *duckdb_cast_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object.

Return Value The extra info.

`duckdb_cast_function_get_cast_mode` Get the cast execution mode from the given function info.

Syntax

```
            duckdb_cast_mode duckdb_cast_function_get_cast_mode(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object.

Return Value The cast mode.

`duckdb_cast_function_set_error` Report that an error has occurred while executing the cast function.

Syntax

```
            void duckdb_cast_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object.
- `error`: The error message.

`duckdb_cast_function_set_row_error` Report that an error has occurred while executing the cast function, setting the corresponding output row to NULL.

Syntax

```
            void duckdb_cast_function_set_row_error(
  duckdb_function_info info,
  const char *error,
  idx_t row,
  duckdb_vector output
);
```

Parameters

- `info`: The info object.
- `error`: The error message.
- `row`: The index of the row within the output vector to set to NULL.
- `output`: The output vector.

`duckdb_register_cast_function` Registers a cast function within the given connection.

Syntax

```
            duckdb_state duckdb_register_cast_function(
  duckdb_connection con,
  duckdb_cast_function cast_function
);
```

Parameters

- `con`: The connection to use.
- `cast_function`: The cast function to register.

Return Value Whether or not the registration was successful.

`duckdb_destroy_cast_function` Destroys the cast function object.

Syntax

```
            void duckdb_destroy_cast_function(
  duckdb_cast_function *cast_function
);
```

Parameters

- `cast_function`: The cast function object.

`duckdb_destroy_expression` Destroys the expression and de-allocates its memory.

Syntax

```
            void duckdb_destroy_expression(
  duckdb_expression *expr
);
```

Parameters

- `expr`: A pointer to the expression.

`duckdb_expression_return_type` Returns the return type of an expression.

Syntax

```
            duckdb_logical_type duckdb_expression_return_type(
  duckdb_expression expr
);
```

Parameters

- `expr`: The expression.

Return Value The return type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_expression_is_foldable` Returns whether the expression is foldable into a value or not.

Syntax

```
            bool duckdb_expression_is_foldable(
  duckdb_expression expr
);
```

Parameters

- `expr`: The expression.

Return Value True, if the expression is foldable, else false.

`duckdb_expression_fold` Folds an expression creating a folded value.

Syntax

```
            duckdb_error_data duckdb_expression_fold(
  duckdb_client_context context,
  duckdb_expression expr,
  duckdb_value *out_value
);
```

Parameters

- `context`: The client context.
- `expr`: The expression. Must be foldable.
- `out_value`: The folded value, if folding was successful. Must be destroyed with `duckdb_destroy_value`.

Return Value The error data. Must be destroyed with `duckdb_destroy_error_data`.

C++ API

> Installation To use the DuckDB C++ API, download the `libduckdb` archive for your platform.
>
> The latest stable version of the DuckDB C++ API is 2.0.0.

> **Warning.** DuckDB's C++ API is internal. It is not guaranteed to be stable and can change without notice. If you would like to build an application on DuckDB, we recommend using the C API.

Installation

The DuckDB C++ API can be installed as part of the `libduckdb` packages. Please see the installation page.

Basic API Usage

DuckDB implements a custom C++ API. This is built around the abstractions of a database instance (`DuckDB` class), multiple `Connections` to the database instance and `QueryResult` instances as the result of queries. The header file for the C++ API is `duckdb.hpp`.

Startup & Shutdown

To use DuckDB, you must first initialize a `DuckDB` instance using its constructor. `DuckDB()` takes as parameter the database file to read and write from. The special value `nullptr` can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process). The second parameter to the `DuckDB` constructor is an optional `DBConfig` object. In `DBConfig`, you can set various database parameters, for example the read/write mode or memory limits. The `DuckDB` constructor may throw exceptions, for example if the database file is not usable.

With the `DuckDB` instance, you can create one or many `Connection` instances using the `Connection()` constructor. While connections should be thread-safe, they will be locked during querying. It is therefore recommended that each thread uses its own connection if you are in a multithreaded environment.

```
DuckDB db(nullptr);
Connection con(db);
```

Querying

Connections expose the `Query()` method to send a SQL query string to DuckDB from C++. `Query()` runs the query to completion and returns a `QueryResult` that retains the result in memory, so it can be read repeatedly. To submit a query without blocking, use `Submit()`.

```
// create a table
con.Query("CREATE TABLE integers (i INTEGER, j INTEGER)");

// insert three rows into the table
con.Query("INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL)");

auto result = con.Query("SELECT * FROM integers");
if (result->HasError()) {
    cerr << result->GetError() << endl;
} else {
    cout << result->ToString() << endl;
}
```

`Query` will not throw exceptions under normal circumstances. Instead, invalid queries or other issues cause the `HasError()` method of the returned `QueryResult` to return `true`. In this case, the error message is available via `GetError()`. The methods `GetErrorType()` and `GetErrorObject()` are also available for any `QueryResult` instance which may aid in more explicit error handling.

```
auto result = con.Query("INSERT INTO integers VALUES (1, 2)");
if (result->HasError()) {
    auto errorType = result->GetErrorType();
    switch (errorType) {
    case duckdb::ExceptionType::CONSTRAINT: {
        // Example handling
        auto errorObject = result->GetErrorObject();
        errorObject.ConvertErrorToJSON();
        std::cout << errorObject.Message() << std::endl;
        break;
    }
    // More handling
    }
} else {
    // Normal code
}
```

If successful, other fields are set: the type of statement that was just executed (e.g., `StatementType::INSERT_STATEMENT`) is returned by `GetStatementType()`. The high-level ("Logical type"/"SQL type") types of the result set columns are returned by `GetTypes()`. The names of the result columns are returned by `GetNames()` as a vector of strings. In case multiple result sets are returned, for example because the result set contained multiple statements, the result set can be chained using the `next` field.

Because `Query()` retains the full result, the result set can be traversed directly by index. Use `ColumnCount()` and `RowCount()` for the dimensions, and `Collection().GetValue(column, row)` to retrieve an individual value as a `duckdb::Value`:

```
auto result = con.Query("SELECT * FROM integers");
if (result->HasError()) {
    cerr << result->GetError() << endl;
} else {
    // Print the column names as a header row
    for (idx_t col_idx = 0; col_idx < result->ColumnCount(); col_idx++) {
        cout << result->GetNames()[col_idx];
        cout << (col_idx + 1 < result->ColumnCount() ? "\t" : "\n");
    }
    // Iterate over every row and column of the materialized result
    for (idx_t row_idx = 0; row_idx < result->RowCount(); row_idx++) {
        for (idx_t col_idx = 0; col_idx < result->ColumnCount(); col_idx++) {
            // GetValue returns a duckdb::Value; ToString renders it (NULL values become "NULL")
            cout << result->Collection().GetValue(col_idx, row_idx).ToString();
            cout << (col_idx + 1 < result->ColumnCount() ? "\t" : "\n");
        }
    }
}
```

DuckDB also supports prepared statements in the C++ API with the `Prepare()` method. This returns an instance of `PreparedStatement`. This instance can be used to execute the prepared statement with parameters. Below is an example:

```
std::unique_ptr<PreparedStatement> prepare = con.Prepare("SELECT count(*) FROM a WHERE i = $1");
std::unique_ptr<QueryResult> result = prepare->Execute(12);
```

> **Warning.** Do **not** use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

UDF API

> **Deprecated.** The `Connection` methods `CreateScalarFunction()`, `CreateVectorizedFunction()` and `CreateAggregateFunction()` were removed in DuckDB 2.0. To define user-defined functions, use the scalar function interface of the C API or write an extension.

CLI

Command Line Client

> Installation To use the DuckDB CLI client, visit the CLI installation page.
>
> The latest stable version of the DuckDB command line client is 2.0.0.

Installation

The DuckDB CLI (Command Line Interface) is a single, dependency-free executable. It is precompiled for Windows, Mac and Linux for both the stable version and for nightly builds produced by GitHub Actions. Please see the installation page under the CLI tab for download links.

The DuckDB CLI is based on the SQLite command line shell, so CLI-client-specific functionality is similar to what is described in the SQLite documentation (although DuckDB's SQL syntax follows PostgreSQL conventions with a few exceptions).

> DuckDB has a tldr page, which summarizes the most common uses of the CLI client. If you have tldr installed, you can display it by running `tldr duckdb`.

Getting Started

Once the CLI executable has been downloaded, decompress it and save it to any directory. Navigate to that directory in a terminal and enter the command `duckdb` to run the executable. If in a PowerShell or POSIX shell environment, use the command `./duckdb` instead.

Usage

The typical usage of the `duckdb` command is the following:

```
duckdb <OPTIONS> <FILENAME>
```

Options

The `<OPTIONS>`{:.language-sql .highlight} part encodes arguments for the CLI client. Common options include:

- `-csv`: sets the output mode to CSV
- `-json`: sets the output mode to JSON
- `-readonly`: open the database in read-only mode (see concurrency in DuckDB)

For a full list of options, see the command line arguments page.

In-Memory vs. Persistent Database

When no `<FILENAME>`{:.language-sql .highlight} argument is provided, the DuckDB CLI will open a temporary in-memory database. You will see DuckDB's version number, the information on the connection and a prompt starting with a D.

```
duckdb
```

```
DuckDB v2.0.0 (Cyanoptera)
Enter ".help" for usage hints.
«memory» D
```

To open or create a persistent database, simply include a path as a command line argument:

```
duckdb my_database.duckdb
```

Running SQL Statements in the CLI

Once the CLI has been opened, enter a SQL statement followed by a semicolon, then hit enter and it will be executed. Results will be displayed in a table in the terminal. If a semicolon is omitted, hitting enter will allow for multi-line SQL statements to be entered.

```
SELECT 'quack' AS my_column;
```

my_column
quack

The CLI supports all of DuckDB's rich SQL syntax including `SELECT`, `CREATE` and `ALTER` statements.

Editor Features

The CLI supports autocompletion, and has sophisticated editor features and syntax highlighting on macOS, Linux and Windows.

Exiting the CLI

To exit the CLI, press `Ctrl+D` if your platform supports it. Otherwise, press `Ctrl+C` or use the `.exit` command. If the session is connected to a remote database via the CONNECT statement, the first `Ctrl+D` disconnects and returns to the local database; press it again to exit. A CLI started with `-connect` exits on the first `Ctrl+D`. If you used a persistent database, DuckDB will automatically checkpoint (save the latest edits to disk) and close. This will remove the `.wal` file (the write-ahead log) and consolidate all of your data into the single-file database.

Dot Commands

In addition to SQL syntax, special dot commands may be entered into the CLI client. To use one of these commands, begin the line with a period (`.`) immediately followed by the name of the command you wish to execute. Additional arguments to the command are entered, space separated, after the command. If an argument must contain a space, either single or double quotes may be used to wrap that parameter. Dot commands must be entered on a single line, and no whitespace may occur before the period. No semicolon is required at the end of the line.

Frequently-used configurations can be stored in the file `~/.duckdbrc`, which will be loaded when starting the CLI client. See the Configuring the CLI section below for further information on these options.

> **Tip.** To prevent the DuckDB CLI client from reading the `~/.duckdbrc` file, start it with the `-no-init` argument:
>
> ```
> duckdb -no-init
> ```

Below, we summarize a few important dot commands. To see all available commands, see the dot commands page or use the `.help` command.

Opening Database Files In addition to connecting to a database when opening the CLI, a new database connection can be made by using the `.open` command. If no additional parameters are supplied, a new in-memory database connection is created. This database will not be persisted when the CLI connection is closed.

```
.open
```

The `.open` command optionally accepts several options, but the final parameter can be used to indicate a path to a persistent database (or where one should be created). The special string `:memory:` can also be used to open a temporary in-memory database.

```
.open persistent.duckdb
```

> **Warning.** `.open` closes the current database. To keep the current database, while adding a new database, use the ATTACH statement.

One important option accepted by `.open` is the `--readonly` flag. This disallows any editing of the database. To open in read only mode, the database must already exist. This also means that a new in-memory database can't be opened in read only mode since in-memory databases are created upon connection.

```
.open --readonly preexisting.duckdb
```

The `--sql` option allows setting the database path using a SQL expression:

```
.open --sql "getenv('MY_DB_PATH')"
```

Output Formats The `.mode` dot command may be used to change the appearance of the tables returned in the terminal output. These include the default `duckbox` mode, `csv` and `json` mode for ingestion by other tools, `markdown` and `latex` for documents and `insert` mode for generating SQL statements.

Writing Results to a File By default, the DuckDB CLI sends results to the terminal's standard output. However, this can be modified using either the `.output` or `.once` commands. For details, see the documentation for the output dot command.

Displaying Function Documentation The `.manual` command shows the signatures, description and examples of a SQL function:

```
.manual regexp_extract
```

For details, see the documentation for the manual dot command.

Reading SQL from a File The DuckDB CLI can read both SQL commands and dot commands from an external file instead of the terminal using the `.read` command. This allows for a number of commands to be run in sequence and allows command sequences to be saved and reused.

The `.read` command requires only one argument: the path to the file containing the SQL and/or commands to execute. After running the commands in the file, control will revert back to the terminal. Output from the execution of that file is governed by the same `.output` and `.once` commands that have been discussed previously. This allows the output to be displayed back to the terminal, as in the first example below, or out to another file, as in the second example.

In this example, the file `select_example.sql` is located in the same directory as duckdb.exe and contains the following SQL statement:

```
SELECT *
FROM generate_series(5);
```

To execute it from the CLI, the `.read` command is used.

```
.read select_example.sql
```

The output below is returned to the terminal by default. The formatting of the table can be adjusted using the `.output` or `.once` commands.

```
| generate_series |
|----------------:|
| 0               |
| 1               |
| 2               |
| 3               |
| 4               |
| 5               |
```

Multiple commands, including both SQL and dot commands, can also be run in a single `.read` command. In this example, the file `write_markdown_to_file.sql` is located in the same directory as duckdb.exe and contains the following commands:

```
.mode markdown
.output series.md
SELECT *
FROM generate_series(5);
```

To execute it from the CLI, the `.read` command is used as before.

```
.read write_markdown_to_file.sql
```

In this case, no output is returned to the terminal. Instead, the file `series.md` is created (or replaced if it already existed) with the markdown-formatted results shown here:

```
| generate_series |
|----------------:|
| 0               |
| 1               |
| 2               |
| 3               |
| 4               |
| 5               |
```

Configuring the CLI

Several dot commands can be used to configure the CLI. On startup, the CLI reads and executes all commands in the file `~/.duckdbrc`, including dot commands and SQL statements. This allows you to store the configuration state of the CLI. You may also point to a different initialization file using the `-init` flag.

> **Warning.** The CLI reads and writes several files in the user's home directory, including `~/.duckdbrc`, `~/.duckdb_history`, and the extension directory `~/.duckdb/extensions`. If the home directory is missing or not writable, startup can fail with an error such as `IO Error: Failed to create directory`. In sandboxed or restricted environments, set the `HOME` environment variable to a writable location to control where these files are created. See Files Created by DuckDB for the full list.

Setting a Custom Prompt

As an example, a file in the same directory as the DuckDB CLI named `prompt.sql` will change the DuckDB prompt to be a duck head and run a SQL statement. Note that the duck head is built with Unicode characters and does not work in all terminal environments (e.g., in Windows, unless running with WSL and using the Windows Terminal).

```
.prompt "{color:yellow1}{sql:select current_database()} 🦆 "
```

Or a simpler version without colours:

```
.prompt "{sql:select current_database()} 🦆 "
```

To invoke that file on initialization, use this command:

```
duckdb -init prompt.sql
```

This outputs:

```
-- Loading resources from prompt.sql
DuckDB v2.0.0 (Cyanoptera)
Enter ".help" for usage hints.
«memory» D
```

Non-Interactive Usage

To read/process a file and exit immediately, redirect the file contents in to `duckdb`:

```
duckdb < select_example.sql
```

To execute a command with SQL text passed in directly from the command line, call `duckdb` with two arguments: the database location (or `:memory:`), and a string with the SQL statement to execute.

```
duckdb :memory: "SELECT 42 AS the_answer"
```

Loading Extensions

To load extensions, use DuckDB's SQL `INSTALL` and `LOAD` commands as you would other SQL statements.

```
INSTALL fts;
LOAD fts;
```

For details, see the Extension docs.

Reading from stdin and Writing to stdout

When in a Unix environment, it can be useful to pipe data between multiple commands. DuckDB is able to read data from stdin as well as write to stdout using the file location of stdin (`/dev/stdin`) and stdout (`/dev/stdout`) within SQL commands, as pipes act very similarly to file handles.

This command will create an example CSV:

```
COPY (SELECT 42 AS woot UNION ALL SELECT 43 AS woot) TO 'test.csv' (HEADER);
```

First, read a file and pipe it to the `duckdb` CLI executable. As arguments to the DuckDB CLI, pass in the location of the database to open, in this case, an in-memory database, and a SQL command that utilizes `/dev/stdin` as a file location.

```
cat test.csv | duckdb -c "SELECT * FROM read_csv('/dev/stdin')"
```

woot
42
43

To write back to stdout, the copy command can be used with the `/dev/stdout` file location.

```
cat test.csv | \
    duckdb -c "COPY (SELECT * FROM read_csv('/dev/stdin')) TO '/dev/stdout' WITH (FORMAT csv, HEADER)"
```

```
woot
42
43
```

Reading Environment Variables

The `getenv` function can read environment variables.

Examples

To retrieve the home directory's path from the HOME environment variable, use:

```
SELECT getenv('HOME') AS home;
```

home
/Users/user_name

The output of the `getenv` function can be used to set configuration options. For example, to set the `NULL` order based on the environment variable `DEFAULT_NULL_ORDER`, use:

```
SET default_null_order = getenv('DEFAULT_NULL_ORDER');
```

Restrictions for Reading Environment Variables

The `getenv` function can only be run when the `enable_external_access` option is set to `true` (the default setting). It is only available in the CLI client and is not supported in other DuckDB clients.

Prepared Statements

The DuckDB CLI supports executing prepared statements in addition to regular `SELECT` statements. To create and execute a prepared statement in the CLI client, use the `PREPARE` clause and the `EXECUTE` statement.

Query Completion ETA

DuckDB's CLI now provides intelligent time-to-completion estimates for running queries and displays total execution time upon completion.

When executing queries in the DuckDB CLI, the progress bar displays an estimated time remaining until completion. This feature employs advanced statistical modeling (Kalman filtering) to deliver more accurate predictions than simple linear extrapolation.

How It Works

DuckDB calculates the estimated time to completion through the following process:

1. Progress Monitoring: DuckDB's internal progress API reports the estimated completion percentage for the running query
2. Statistical Filtering: A Kalman filter smooths noisy progress measurements and accounts for execution variability
3. Continuous Refinement: The system continuously updates predicted completion time as new progress data becomes available, improving accuracy throughout execution

The Kalman filter adapts to changing execution conditions such as memory pressure, I/O bottlenecks, or network delays. This adaptive approach means estimated completion times may not always decrease linearly—estimates can increase when query execution becomes less predictable.

Factors Affecting The Accuracy of Query Completion ETA

Completion time estimates may be less reliable under these conditions:

System resource constraints:

- Memory pressure causing disk swapping

- High CPU load from competing processes
- Disk I/O bottlenecks

Query execution characteristics:

- Variable execution phases (initial setup versus main processing)
- Network-dependent operations with inconsistent latency
- Queries with unpredictable branching logic
- Operations on remote data sources
- External function calls
- Highly skewed data distributions

Command Line Arguments

The table below summarizes DuckDB's command line options. To list all command line options, use the command:

```
duckdb -help
```

For a list of dot commands available in the CLI shell, see the Dot Commands page.

Argument	Description
`-agent`	Render output for an AI coding agent. See the Agent Mode page
`-ascii`	Set output mode to `ascii`
`-bail`	Stop after hitting an error
`-batch`	Force batch I/O
`-box`	Set output mode to `box`
`-column`	Set output mode to `column`
`-cmd COMMAND`	Run `COMMAND` before reading `stdin`
`-connect [TYPE[:SECRET]]`	Connect to a remote database. The default type is `quack`. A named secret can be provided after the type
`-c COMMAND`	Run `COMMAND` and exit
`-csv`	Set output mode to `csv`
`-dark-mode`	Use dark mode colors for syntax highlighting
`-echo`	Print commands before execution
`-f FILENAME`	Run the script in `FILENAME` and exit. Note that the `~/.duckdbrc` is read and executed first (if it exists)

Argument	Description
`-format`	Format SQL read from `stdin`, writing the result to `stdout`
`-format-file FILENAME`	Format the SQL in `FILENAME`, writing the result to `stdout`
`-init FILENAME`	Run the script in `FILENAME` upon startup (instead of `~/.duckdbrc`)
`-header`	Turn headers on
`-h`	Show the help message
`-help`	Show the help message
`-html`	Set output mode to HTML
`-interactive`	Force interactive I/O
`-json`	Set output mode to `json`
`-jsonlines`	Set output mode to `jsonlines`
`-light-mode`	Use light mode colors for syntax highlighting
`-line`	Set output mode to `line`
`-list`	Set output mode to `list`
`-manual FUNCTION`	Show the manual page for a SQL function and exit
`-markdown`	Set output mode to `markdown`
`-newline SEP`	Set output row separator. Default: `\n`
`-no-agent`	Never render output for an AI coding agent. See the Agent Mode page
`-no-init`	Skip processing the init file (`~/.duckdbrc` or the file passed to `-init`)
`-noheader`	Turn headers off
`-no-stdin`	Exit after processing options instead of reading stdin

Argument	Description
`-nullvalue TEXT`	Set text string for `NULL` values. Default: `NULL`
`-quote`	Set output mode to `quote`
`-readonly`	Open the database read-only. This option also supports attaching to remote databases via HTTPS
`-s COMMAND`	Run `COMMAND` and exit
`-safe`	Enable safe mode
`-separator SEP`	Set output column separator to `SEP`. Default: `|`
`-serve [quack[:SECRET]]`	Serve the opened database with Quack. A named secret can be provided after the type
`-storage_version VER`	Database storage compatibility version to use. To use the latest storage version, pass `-storage_version latest`
`-table`	Set output mode to `table`
`-ui`	Loads and starts the DuckDB UI. If the UI is not yet installed, it installs the `ui` extension
`-unredacted`	Allow printing unredacted secrets. Sets the `allow_unredacted_secrets` option to `true`
`-unsigned`	Allow loading of unsigned extensions. This option is intended to be used for developing extensions. Consult the Securing DuckDB page for guidelines on how to set up DuckDB in a secure manner
`-version`	Show DuckDB version

Serving and Connecting to Remote Databases

> New The `-serve` and `-connect` arguments were introduced in DuckDB 2.0.

The `-serve` argument starts a Quack server for the database opened by the CLI. Without a database path or additional configuration, it serves an in-memory database on `localhost:9494`:

```
duckdb -serve
```

To serve a persistent database, provide its path before `-serve`:

```
duckdb analytics.duckdb -serve
```

The `-connect` argument connects the CLI to a remote database. Without arguments, it connects to the Quack server on `localhost:9494` using the default `quack` secret. If this secret does not exist yet, `-serve` creates it as a persistent secret, so running `duckdb -serve` and `duckdb -connect` on the same machine works without further setup:

```
duckdb -connect
```

Both arguments accept a connection type and an optional named secret in the form `TYPE:SECRET`. Only the `quack` type can be served. For example, after creating a persistent Quack secret named `analytics`, serve or connect with:

```
duckdb analytics.duckdb -serve quack:analytics
duckdb -connect quack:analytics
```

The `-connect` argument cannot be combined with a local database path because the CLI acts as a client for the remote database.

The SQL commands executed by `-serve` and `-connect` can be customized using the `.serve_command` and `.connect_command` dot commands, e.g., in the `~/.duckdbrc` file. By default, `-serve` calls `quack_serve` and `-connect` runs a `CONNECT` statement.

Passing a Sequence of Arguments

Note that the CLI arguments are processed in order, similarly to the behavior of the SQLite CLI. For example:

```
duckdb -csv -c 'SELECT 42 AS hello' -json -c 'SELECT 84 AS world'
```

Returns the following:

```
hello
42
[{"world":84}]
```

> The `-init` script (or the default `~/.duckdbrc` when `-init` is not given) is an exception: it always runs at startup, before any `-cmd`, `-c`, or `-s` command, regardless of where it appears on the command line. You therefore cannot use a `-cmd` argument to inject configuration that takes effect before the init script runs. To set configuration that the init script depends on, place it at the top of the init script itself.

Dot Commands

Dot commands are available in the DuckDB CLI client. To use one of these commands, begin the line with a period (`.`) immediately followed by the name of the command you wish to execute. Additional arguments to the command are entered, space separated, after the command. If an argument must contain a space, either single or double quotes may be used to wrap that parameter. Dot commands must be entered on a single line, and no whitespace may occur before the period. No semicolon is required at the end of the line. To see available commands, use the `.help` command.

List of Dot Commands

Command	Description
`.about`{:.language-sql .highlight}	Prints version, description and license information about DuckDB.
`.auto_format <on/off>`{:.language-sql .highlight}	Automatically format SQL before execution. Default: `off`
`.bail <on/off>`{:.language-sql .highlight}	Stop after hitting an error. Default: `off`
`.binary <on/off>`{:.language-sql .highlight}	Turn binary output on or `off`. Default: `off`
`.cd <DIRECTORY>`{:.language-sql .highlight}	Change the working directory to `DIRECTORY`
`.changes <on/off>`{:.language-sql .highlight}	Show number of rows changed by SQL
`.columns`{:.language-sql .highlight}	Column-wise rendering of query results

Command	Description
`.connect_command <COMMAND>`{:.language-sql .highlight}	Set the command executed by the `-connect` argument
`.databases`{:.language-sql .highlight}	List names and files of attached databases
`.decimal_sep <SEP>`{:.language-sql .highlight}	Set the decimal separator used when rendering numbers. Only for duckbox mode
`.dump <TABLE>`{:.language-sql .highlight}	Render database content as SQL. `TABLE` is a `LIKE` pattern for the tables to dump
`.display_colors <bold/underline>`{:.language-sql .highlight}	Display all terminal colors and their names. See the 8-bit colors section
`.echo <on/off>`{:.language-sql .highlight}	Turn command echo on or `off`
`.edit`{:.language-sql .highlight}	Open an external text editor to edit a query. `\e` can be used as an alias. See the external editor mode section
`.excel`{:.language-sql .highlight}	Display the output of the next command in a spreadsheet. Equivalent to `.once -x`
`.exit <CODE>`{:.language-sql .highlight}	Exit this program with return-code `CODE`
`.headers <on/off>`{:.language-sql .highlight}	Turn display of headers on or `off`. Does not apply to duckbox mode
`.help <-all> <PATTERN>`{:.language-sql .highlight}	Show help text for `PATTERN`. Use `.help shortcuts` to display keyboard shortcuts
`.highlight <on/off>`{:.language-sql .highlight}	Toggle syntax highlighting in the shell on / `off`. See the query syntax highlighting section for more details
`.highlight_colors <COMPONENT> <COLOR>`{:.language-sql .highlight}	Configure the color of each query and result component. See the Syntax Highlighting page for details
`.highlight_errors <on/off>`{:.language-sql .highlight}	Turn highlighting of error messages on or `off`
`.highlight_mode <mixed/dark/light>`{:.language-sql .highlight}	Toggle the highlight mode. See the dark/light mode section for details
`.highlight_results <on/off>`{:.language-sql .highlight}	Toggle highlighting in result tables on / `off` (duckbox only). See the result syntax highlighting section for more details
`.history <N>`{:.language-sql .highlight}	Show the command history with syntax highlighting. If `N` is given, only the last `N` entries are shown. See the command history section
`.import <FILE> <TABLE>`{:.language-sql .highlight}	Import data from `FILE` into `TABLE`. Supports `--csv`, `--json`, `--parquet` options
`.indexes <TABLE>`{:.language-sql .highlight}	Show names of indexes
`.large_number_rendering <all/footer/off>`{:.language-sql .highlight}	Toggle readable rendering of large numbers (duckbox only, default: `footer`)

Command	Description
`.last`{:.language-sql .highlight}	Render the last result without truncating. Useful for navigating with the pager
`.log <FILE/off>`{:.language-sql .highlight}	Turn logging on or `off`. `FILE` can be `stderr` / `stdout`
`.manual <FUNCTION>`{:.language-sql .highlight}	Show the manual page (signatures, descriptions and examples) for a SQL function. See the manual section
`.maxbytes <COUNT>`{:.language-sql .highlight}	Sets the maximum number of bytes of rows for display (`0` = all). Only for agent mode
`.maxcellwidth <COUNT>`{:.language-sql .highlight}	Sets the maximum number of characters per cell (`0` = all). Only for agent mode
`.maxrows <COUNT>`{:.language-sql .highlight}	Sets the maximum number of rows for display (default: `40`, `-1` = all). For duckbox mode and agent mode
`.maxwidth <COUNT>`{:.language-sql .highlight}	Sets the maximum width in characters. `0` defaults to terminal width. Only for duckbox mode
`.mode <MODE> <TABLE>`{:.language-sql .highlight}	Set output mode
`.multiline`{:.language-sql .highlight}	Set multi-line mode (default)
`.nullvalue <STRING>`{:.language-sql .highlight}	Use `STRING` in place of `NULL` values. Default: `NULL`
`.once <OPTIONS> <FILE>`{:.language-sql .highlight}	Output for the next SQL command only to `FILE`
`.open <OPTIONS> <FILE>`{:.language-sql .highlight}	Close existing database and reopen `FILE`. Options: `--new`, `--nofollow`, `--readonly`, `--sql`
`.output <FILE>`{:.language-sql .highlight}	Send output to `FILE` or `stdout` if `FILE` is omitted
`.pager <OPTIONS>`{:.language-sql .highlight}	Control pager usage for output. See the paging section for details
`.print <STRING...>`{:.language-sql .highlight}	Print literal `STRING`
`.progress_bar <COMPONENT>`{:.language-sql .highlight}	Set the progress bar component styles
`.prompt <OPTIONS> <CONTINUE>`{:.language-sql .highlight}	Replace the standard prompts
`.quit`{:.language-sql .highlight}	Exit this program
`.read <FILE>`{:.language-sql .highlight}	Read input from `FILE`
`.read_line_version <linenoise/fallback>`{:.language-sql .highlight}	Set the library used for processing interactive input

Command	Description
`.render_completion <on/off>`{:.language-sql .highlight}	Toggle displaying of completion prompts in the shell on / `off`
`.render_errors <on/off>`{:.language-sql .highlight}	Toggle rendering of errors (such as mismatched brackets and unclosed quotes) in the shell on / `off`. See the Syntax Highlighting page
`.rows`{:.language-sql .highlight}	Row-wise rendering of query results (default)
`.safe_mode`{:.language-sql .highlight}	Activates safe mode
`.schema <OPTIONS> <PATTERN>`{:.language-sql .highlight}	Show the CREATE statements matching PATTERN. Pretty-prints and highlights by default; use `--no-indent` (alias `--no-format`) for the stored single-line form
`.separator <COL> <ROW>`{:.language-sql .highlight}	Change the column and row separators
`.serve_command <COMMAND>`{:.language-sql .highlight}	Set the command executed by the `-serve` argument
`.shell <CMD> <ARGS...>`{:.language-sql .highlight}	Run CMD with ARGS... in a system shell
`.show`{:.language-sql .highlight}	Show the current values for various settings
`.singleline`{:.language-sql .highlight}	Set single-line mode
`.startup_text <none/version/all>`{:.language-sql .highlight}	Controls the start-up text displayed when launching the CLI. Set this as the first line in `~/.duckdbrc`
`.system <CMD> <ARGS...>`{:.language-sql .highlight}	Run CMD with ARGS... in a system shell
`.tables <TABLE>`{:.language-sql .highlight}	List tables matching `LIKE` pattern TABLE with column names, types and row counts, grouped by database and schema
`.thousand_sep <SEP>`{:.language-sql .highlight}	Set the thousand separator used when rendering numbers. Only for duckbox mode
`.timer <on/off> <DIGITS>`{:.language-sql .highlight}	Turn SQL timer on or `off`, with DIGITS decimals for the real time (default: 3). SQL statements separated by `;` but *not* separated via newline are measured together
`.ui_command <COMMAND>`{:.language-sql .highlight}	Set the command executed by the `-ui` argument. CALL is prepended to COMMAND. Default: `CALL start_ui()`
`.version`{:.language-sql .highlight}	Show the DuckDB version
`.web`{:.language-sql .highlight}	Render the query graph of the previous query and open it in a web browser. Equivalent to `EXPLAIN ANALYZE (FORMAT WEB)`. Disabled in safe mode
`.width <NUM1> <NUM2> ...`{:.language-sql .highlight}	Set minimum column widths for columnar output

Using the `.help` Command

The `.help` text may be filtered by passing in a text string as the first argument.

```
.help m
```

```
.manual FUNCTION       Show the manual page for a SQL function
  Displays the signatures, descriptions and examples of all overloads of FUNCTION.
  FUNCTION may be qualified: [database.][schema.]function.
.maxbytes COUNT        Sets the maximum number of bytes of rows for display (0 = all). Only for markdown
mode in -agent mode.
.maxcellwidth COUNT    Sets the maximum number of characters per cell (0 = all). Only for markdown mode in
-agent mode.
.maxrows COUNT         Sets the maximum number of rows for display (default: 40, -1 = all). Only for duckbox
mode and for markdown mode in -agent mode.
.maxwidth COUNT        Sets the maximum width in characters. 0 defaults to terminal width. Only for duckbox
mode.
.mode MODE ?TABLE?     Set output mode
  MODE is one of:
    ascii              Columns/rows delimited by 0x1F and 0x1E
    box                Tables using unicode box-drawing characters
    csv                Comma-separated values
    column             Output in columns. (See .width)
    duckbox            Tables with extensive features
    html               HTML <table> code
    insert             SQL insert statements for TABLE
    json               Results in a JSON array
    jsonlines          Results in a NDJSON
    latex              LaTeX tabular environment code
    line               One value per line
    list               Values delimited by "|"
    markdown           Markdown table format
    quote              Escape answers as for SQL
    table              ASCII-art table
    tabs               Tab-separated values
    tcl                TCL list elements
    trash              No output
.multiline             Sets the render mode to multi-line
```

`.output`: Writing Results to a File

By default, the DuckDB CLI sends results to the terminal's standard output. However, this can be modified using either the `.output` or `.once` commands. Pass in the desired output file location as a parameter. The `.once` command will only output the next set of results and then revert to standard out, but `.output` will redirect all subsequent output to that file location. Note that each result will overwrite the entire file at that destination. To revert back to standard output, enter `.output` with no file parameter.

In this example, the output format is changed to `markdown`, the destination is identified as a Markdown file, and then DuckDB will write the output of the SQL statement to that file. Output is then reverted to standard output using `.output` with no parameter.

```
.mode markdown
.output my_results.md
SELECT 'taking flight' AS output_column;
.output
SELECT 'back to the terminal' AS displayed_column;
```

The file `my_results.md` will then contain:

```
| output_column |
| ------------- |
| taking flight |
```

The terminal will then display:

```
| displayed_column     |
| -------------------- |
| back to the terminal |
```

A common output format is CSV, or comma separated values. DuckDB supports SQL syntax to export data as CSV or Parquet, but the CLI-specific commands may be used to write a CSV instead if desired.

```
.mode csv
.once my_output_file.csv
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

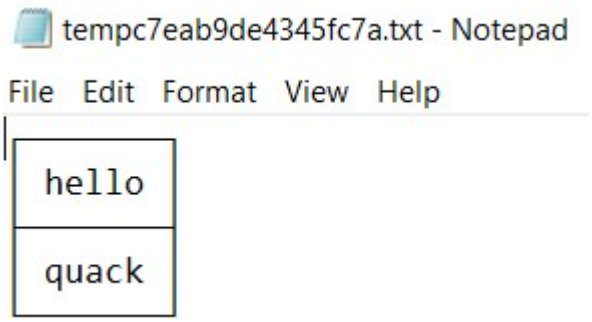
The file `my_output_file.csv` will then contain:

```
col_1,col_2
1,2
10,20
```

By passing special options (flags) to the `.once` command, query results can also be sent to a temporary file and automatically opened in the user's default program. Use either the `-e` flag for a text file (opened in the default text editor), or the `-x` flag for a CSV file (opened in the default spreadsheet editor). This is useful for more detailed inspection of query results, especially if there is a relatively large result set. The `.excel` command is equivalent to `.once -x`.

```
.once -e
SELECT 'quack' AS hello;
```

The results then open in the default text file editor of the system, for example:



> **Tip.** macOS users can copy the results to their clipboards using `pbcopy` by using `.once` to output to `pbcopy` via a pipe: `.once |pbcopy`
>
> Combining this with the `.headers off` and `.mode lines` options can be particularly effective.

Querying the Database Schema

All DuckDB clients support querying the database schema with SQL, but the CLI has additional dot commands that can make it easier to understand the contents of a database. The `.tables` command will return a list of tables in the database. It has an optional argument that will filter the results according to a `LIKE` pattern.

```
CREATE TABLE swimmers AS SELECT 'duck' AS animal;
CREATE TABLE fliers AS SELECT 'duck' AS animal;
CREATE TABLE walkers AS SELECT 'duck' AS animal;
.tables
```

```
──────────────────── memory ────────────────────
──────────────────── main ─────────────────────
┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│   swimmers   │ │   walkers    │ │    fliers    │
└──────────────┘ └──────────────┘ └──────────────┘
```

```
┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐
│ animal varchar  │ │ animal varchar  │ │ animal varchar  │
│                 │ │                 │ │                 │
│     1 rows      │ │     1 rows      │ │     1 rows      │
└─────────────────┘ └─────────────────┘ └─────────────────┘
```

For example, to filter to only tables that contain an `l`, use the `LIKE` pattern `%l%`.

```
.tables %l%
```

```
──────────── memory ────────────
──────────── main ─────────────
┌─────────────────┐ ┌─────────────────┐
│    walkers      │ │     fliers      │
│                 │ │                 │
│ animal varchar  │ │ animal varchar  │
│                 │ │                 │
│     1 rows      │ │     1 rows      │
└─────────────────┘ └─────────────────┘
```

The `.schema` command will show all of the SQL statements used to define the schema of the database.

> New As of DuckDB 2.0, `.schema` pretty-prints and highlights its output by default. Use `--no-indent` (alias `--no-format`) to restore the earlier single-line form.

By default the statements are pretty-printed with indentation and, on an interactive terminal, syntax highlighting:

```
.schema
```

```
CREATE TABLE fliers(
    animal VARCHAR
);
CREATE TABLE swimmers(
    animal VARCHAR
);
CREATE TABLE walkers(
    animal VARCHAR
);
```

To print the statements as they are stored, one per line and without formatting, use `--no-indent` (or its alias `--no-format`):

```
.schema --no-indent
```

```
CREATE TABLE fliers(animal VARCHAR);
CREATE TABLE swimmers(animal VARCHAR);
CREATE TABLE walkers(animal VARCHAR);
```

The `--indent` option (alias `--format`) forces pretty-printing, which is the default.

Dumping Database Content as SQL

The `.dump` command renders the database content as SQL statements, including both schema definitions and data. This is useful for creating backups or migrating data.

```
.dump
```

An optional `TABLE` argument filters the output using a `LIKE` pattern. Multiple patterns can be provided as additional arguments.

```
.dump %swim%
```

The `--newlines` option allows unescaped newline characters in the output:

```
.dump --newlines
```

Command History

> New As of DuckDB 2.0, the `.history` dot command and the `shell_history()` table function are available in the command line client.

The `.history` command shows the previously executed statements with syntax highlighting:

```
.history
```

To show only the last <N> entries, pass a count:

```
.history 10
```

The same information is available as a table through the `shell_history()` table function, which returns one row per history entry with a numeric `id` column and the statement text in a `sql` column:

```
FROM shell_history();
```

The history reflects the interactive shell session, so it is only populated in interactive use.

Displaying Function Documentation

> New The `.manual` dot command (and the `-manual` command line argument) was introduced in DuckDB 2.0.

The `.manual` command displays a manual page for a SQL function, including its signatures (overloads), description and examples. This information is retrieved from the duckdb_functions() table function.

```
.manual map
```

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────┐
│ system.main                                map                                scalar function    │
│                                                                                                  │
│ SIGNATURE                                                                                        │
│                                                                                                  │
│       map() -> MAP("NULL", "NULL")                                                               │
│                                                                                                  │
│       map(keys K[], values V[]) -> MAP(K, V)                                                     │
│                                                                                                  │
│ DESCRIPTION                                                                                      │
│                                                                                                  │
│       Creates a map from a set of keys and values                                                │
│                                                                                                  │
│ EXAMPLES                                                                                         │
│                                                                                                  │
│       map(['key1', 'key2'], ['val1', 'val2'])                                                    │
└──────────────────────────────────────────────────────────────────────────────────────────────────┘
```

If the overloads of a function have different descriptions or examples, the signatures are numbered and the descriptions and examples refer to these numbers. If multiple functions match the name (e.g., `range` is both a scalar function and a table function), a separate manual page is shown for each of them. Long output is shown using the pager if it is enabled.

The function name may be qualified with a schema and a database name, i.e., `<database>.<schema>.<function>`{:.language-sql .highlight}. Each part of the name is matched case-insensitively as a LIKE pattern, so wildcards can be used:

```
.manual list_%
```

If no function matches the name, the CLI suggests similarly named functions.

The manual page can also be displayed directly from the command line using the `-manual` argument:

```
duckdb -manual regexp_extract
```

Progress Bar

The DuckDB CLI client's progress bar supports customization through components.

The `.progress_bar` command supports `--add` and `--clear` parameters for adding and removing components.

For details on specific usage, see the examples below.

Configuring the Progress Bar Display

To check if the progress bar is enabled:

```
SELECT * FROM duckdb_settings() WHERE name = 'enable_progress_bar';
```

To check the current minimum amount of time (in milliseconds) a query needs to take before displaying a progress bar:

```
SELECT * FROM duckdb_settings() WHERE name = 'progress_bar_time';
```

To set the minimum amount of time that the progress bar displays to 100 milliseconds:

```
SET progress_bar_time = 100;
```

To set that progress bar component to a red text that displays the current time on the progress bar:

```
.progress_bar --add "{align:right}{min_size:20}{color:red}Time: {sql:select (current_
time::varchar).split('.')[1]}{color:reset} "
```



> `.progress_bar --add` commands are additive, issuing multiple `--add` calls will stack additional components on the progress bar.

To set that progress bar component to a blue text that displays the file cache RAM usage on the progress bar:

```
.progress_bar --add "{align:right}{min_size:20}{color:blue}External Cache Usage: {sql:select format_
bytes(memory_usage_bytes) from duckdb_memory() where tag='EXTERNAL_FILE_CACHE'}{color:reset}"
```



To reset all existing progress bar components:

```
.progress_bar --clear
```

Syntax Highlighters

The DuckDB CLI client has a syntax highlighter for the SQL queries and another for the duckbox-formatted result tables.

Configuring the Query Syntax Highlighter

By default the shell includes support for syntax highlighting. The CLI's syntax highlighter can be configured using the following commands.

To turn off the highlighter:

```
.highlight off
```

To turn on the highlighter:

```
.highlight on
```

To configure the color used to highlight a component of the query, use the `.highlight_colors` command. For example, to highlight keywords in red and constants in yellow:

```
.highlight_colors keyword red
.highlight_colors numeric_constant yellow
.highlight_colors string_constant yellow
```

See the Syntax Highlighting page for the list of components and colors.

Configuring the Result Syntax Highlighter

By default, the result highlighting makes a few small modifications:

- Bold column names.
- `NULL` values are greyed out.
- Layout elements are grayed out.

The highlighting of each of the components can be customized using the `.highlight_colors` command. For example:

```
.highlight_colors layout red
.highlight_colors column_type yellow
.highlight_colors column_name yellow bold_underline
.highlight_colors numeric_value cyan underline
.highlight_colors temporal_value red bold
.highlight_colors string_value green bold
.highlight_colors footer gray
```

The result highlighting can be disabled using `.highlight_results off`.

Shorthands

DuckDB's CLI allows using shorthands for dot commands. Once a sequence of characters can be unambiguously completed to a dot command or an argument, the CLI (silently) autocompletes them. For example:

```
.mo ma
```

Is equivalent to:

```
.mode markdown
```

> **Tip.** Avoid using shorthands in SQL scripts to improve readability and ensure that the scripts are future-proof.

Importing Data

The `.import` command imports data from a file into a DuckDB table. It uses DuckDB's reader functions (`read_csv`, `read_json`, `read_parquet`) and supports automatic schema detection. If the target table does not exist, it is automatically created.

The file format can be specified explicitly using `--csv`, `--json`, or `--parquet`. If no format is specified, the format is inferred from the file extension.

```
.import data.csv my_table
```

Additional parameters can be passed to the underlying reader function using `--<parameter> <value>`{:.language-sql .highlight} syntax:

```
.import data.csv my_table --delimiter "|" --header false
```

To import a JSON file:

```
.import data.json my_table --json
```

To import a Parquet file:

```
.import data.parquet my_table
```

Output Formats

The `.mode` dot command may be used to change the appearance of the tables returned in the terminal output. In addition to customizing the appearance, these modes have additional benefits. This can be useful for presenting DuckDB output elsewhere by redirecting the terminal output to a file. Using the `insert` mode will build a series of SQL statements that can be used to insert the data at a later point. The `markdown` mode is particularly useful for building documentation and the `latex` mode is useful for writing academic papers.

> **Warning.** Unicode handling in Windows Terminal
>
> When long results are displayed in Windows Terminal the more system utility is used by default to provide the scrolling through the results. This utility has incomplete support of Unicode, depending on the output data, in some cases it can display Unicode characters in garbled form.
>
> We suggest using the third-party less utility instead, that is installed by default along with the Git for Windows installation. It can be enabled the following way:
>
> ```
> .pager '"C:\Program Files\Git\usr\bin\less.exe" -R'
> ```

List of Output Formats

Mode	Description
`ascii`	Columns/rows delimited by 0x1F and 0x1E
`box`	Tables using unicode box-drawing characters
`csv`	Comma-separated values
`column`	Output in columns (See `.width`)
`duckbox`	Tables with extensive features (default)
`html`	HTML `<table>` code
`insert <TABLE>`{:.language-sql .highlight}	SQL insert statements for `<TABLE>`{:.language-sql .highlight}
`json`	Results in a JSON array
`jsonlines`	Results in a NDJSON

Mode	Description
`latex`	LaTeX tabular environment code
`line`	One value per line
`list`	Values delimited by |
`markdown`	Markdown table format
`quote`	Escape answers as for SQL
`table`	ASCII-art table
`tabs`	Tab-separated values
`tcl`	TCL list elements
`trash`	No output

Changing the Output Format

Use the vanilla `.mode` dot command to query the appearance currently in use.

```
.mode
```

```
current output mode: duckbox
```

Use the `.mode` dot command with an argument to set the output format.

```
.mode markdown
SELECT 'quacking intensifies' AS incoming_ducks;
```

```
|    incoming_ducks    |
|----------------------|
| quacking intensifies |
```

The output appearance can also be adjusted with the `.separator` command. If using an export mode that relies on a separator (`csv` or `tabs` for example), the separator will be reset when the mode is changed. For example, `.mode csv` will set the separator to a comma (`,`). Using `.separator "|"` will then convert the output to be pipe-separated.

```
.mode csv
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

```
col_1,col_2
1,2
10,20
```

```
.separator "|"
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

```
col_1|col_2
1|2
10|20
```

Paging

The CLI supports paging for large result sets using the `.pager` command. When enabled, results that exceed the terminal size are displayed in a pager (such as `less`) for easier navigation.

The pager has three modes:

- `automatic` (default) – The pager is triggered when the result exceeds the row or column threshold.
- `on` – The pager is always used for output.
- `off` – The pager is disabled.

```
.pager on
```

```
.pager off
```

```
.pager automatic
```

In automatic mode, the thresholds for triggering the pager can be configured:

```
.pager set_row_threshold 50
.pager set_column_threshold 5
```

A custom pager command can be set by passing it as an argument:

```
.pager less -RS
```

The default pager command can also be configured via the DUCKDB_PAGER or PAGER environment variables.

duckbox Mode

By default, DuckDB renders query results in `duckbox` mode, which is a feature-rich ASCII-art style output format.

The duckbox mode supports the `large_number_rendering` option, which allows human-readable rendering of large numbers. It has three levels:

- `off` – All numbers are printed using regular formatting.
- `footer` (default) – Large numbers are augmented with the human-readable format. Only applies to single-row results.
- `all` - All large numbers are replaced with the human-readable format.

See the following examples:

```
.large_number_rendering off
SELECT pi() * 1_000_000_000 AS x;
```

```
┌────────────────────┐
│          x         │
│       double       │
├────────────────────┤
│ 3141592653.589793  │
└────────────────────┘
```

```
.large_number_rendering footer
SELECT pi() * 1_000_000_000 AS x;
```

```
┌────────────────────┐
│          x         │
│       double       │
├────────────────────┤
│ 3141592653.589793  │
│  (3.14 billion)    │
└────────────────────┘
```

```
.large_number_rendering all
SELECT pi() * 1_000_000_000 AS x;
```

```
┌──────────────┐
│      x       │
│    double    │
├──────────────┤
│ 3.14 billion │
└──────────────┘
```

Agent Mode

> New Agent mode was introduced in DuckDB 2.0.

Agent mode renders the CLI's output for consumption by an AI coding agent rather than a human at a terminal. It produces compact Markdown tables with explicit result limits, machine-readable errors, and progress information on `stderr`, so that an agent can run queries and reliably parse the results.

For the motivation behind agent mode and a benchmark of its effect, see the "Agent Mode in the DuckDB CLI" blog post.

Activating Agent Mode

Agent mode is activated automatically when all of the following hold:

- A known AI agent environment variable (such as `AI_AGENT`) is set.
- The standard output is not a terminal (for example, it is piped to another process).
- No output mode flag (such as `-csv` or `-json`) was passed.

You can also control agent mode explicitly:

- The `-agent` and `-no-agent` command line arguments force the mode on or off.
- The `DUCKDB_AGENT_MODE` environment variable forces the mode on (`1`) or off (`0`).

The command line flags take precedence over the environment variable. To hide the startup line that announces agent mode, set `.startup_text none` as the first line in `~/.duckdbrc`.

For example, to force agent mode on when running a single command:

```
duckdb -agent -c "SELECT 42 AS answer;"
```

To force it off:

```
duckdb -no-agent -c "SELECT 42 AS answer;"
```

Output Format

In agent mode, DuckDB:

- Renders results as compact Markdown tables.
- Prints a footer with the row count and, when the whole result was read, an order-independent hash of the result.
- Emits errors as JSON on `stderr`.
- Renders compact query plans and cost estimates.
- Prints an estimate line and progress on `stderr` for a query that reads more than 1,000,000 rows.

By default, a result of up to 1,000 rows and 10,000 bytes is rendered whole; a larger result is rendered as its first and last rows together with the total row count. Cells are truncated at 500 characters.

These limits can be adjusted with dot commands:

Dot command	Description
`.maxrows <count>`	Maximum number of rows to render whole (`-1` for all)
`.maxbytes <count>`	Maximum number of bytes of rows to render whole (`0` for all)
`.maxcellwidth <count>`	Maximum number of characters per cell (`0` for all)

Tips for Agents

The `.help agent` command prints a summary of agent mode. A few things that help when driving DuckDB programmatically:

- `SET max_execution_time = <milliseconds>`{:.language-sql .highlight} bounds how long a query may run.
- `DESCRIBE <query>` returns the result columns without running the query.
- `SUMMARIZE <table>` returns summary statistics for a table.
- `.tables` lists the available tables.
- `duckdb_functions()` returns function descriptions and examples.

Editing

> The linenoise-based CLI editor is available for macOS, Linux and Windows.

DuckDB's CLI uses a line-editing library based on linenoise, which has shortcuts that are based on Emacs mode of readline. Below is a list of available commands. You can also view these shortcuts from within the CLI using `.help shortcuts`.

Moving

Key	Action
`Left`	Move back a character
`Right`	Move forward a character
Up	Move up a line. When on the first line, move to previous history entry
Down	Move down a line. When on last line, move to next history entry
Home	Move to beginning of buffer
End	Move to end of buffer
`Ctrl+Left`	Move back a word
`Ctrl+Right`	Move forward a word
`Ctrl+A`	Move to beginning of line
`Ctrl+B`	Move back a character
`Ctrl+E`	Move to end of line
`Ctrl+F`	Move forward a character
`Alt+Left`	Move back a word
`Alt+Right`	Move forward a word

History

Key	Action
`Ctrl+P`	Move to previous history entry
`Ctrl+N`	Move to next history entry
`Ctrl+R`	Reverse search the history
`Ctrl+S`	Forward search the history

Key	Action
Ctrl+Up	Move to first history entry
Ctrl+Down	Move to last history entry
Alt+N	Search the history
Alt+P	Search the history

Changing Text

Key	Action
Backspace	Delete previous character
Delete	Delete next character
Ctrl+D	Delete next character. When buffer is empty, end editing
Ctrl+H	Delete previous character
Ctrl+K	Delete everything after the cursor
Ctrl+T	Swap character under cursor with previous character
Ctrl+U	Delete all text
Ctrl+W	Delete previous word
Alt+C	Convert next word to titlecase
Alt+D	Delete next word
Alt+L	Convert next word to lowercase
Alt+R	Delete all text
Alt+T	Swap current word with previous word
Alt+U	Convert next word to uppercase
Alt+Backspace	Delete previous word
Alt+\	Delete spaces around cursor

Completing

Key	Action
Tab	Autocomplete. When autocompleting, cycle to next entry
Shift+Tab	When autocompleting, cycle to previous entry
Esc+Esc	When autocompleting, revert autocompletion

Miscellaneous

Key	Action
Enter	Execute query. If query is not complete, insert a newline at the end of the buffer

Key	Action
`Ctrl+J`	Execute query. If query is not complete, insert a newline at the end of the buffer
`Ctrl+C`	Cancel editing of current query or interrupt the currently running query
`Ctrl+G`	Execute query. If query is not complete, terminate it with `;` and execute it
`Ctrl+L`	Clear screen
`Ctrl+O`	Cancel editing of current query
`Ctrl+X`	Insert a newline after the cursor
`Ctrl+Q`, then click	Move cursor to mouse click position
`Ctrl+Z`	Suspend CLI and return to shell, use `fg` to re-open

External Editor Mode

Use `.edit` or `\e` to open a query in an external text editor.

- When entered alone, it opens the previous command for editing.
- When used inside a multi-line command, it opens the current command in the editor.

The editor is taken from the first set environment variable among `DUCKDB_EDITOR`, `EDITOR` or `VISUAL` (in that order). If none are set, `vi` is used.

Using Read-Line

If you prefer, you can use `rlwrap` to use read-line directly with the shell. Then, use `Shift+Enter` to insert a newline and `Enter` to execute the query:

```
rlwrap --substitute-prompt="D " duckdb -batch
```

Friendly CLI

Along with our Friendly SQL, we provide **friendly CLI** features.

Dark/Light Mode

The CLI automatically detects whether the terminal is using a dark or light background and adjusts syntax highlighting colors accordingly. The mode can also be set manually using the `.highlight_mode` command:

```
.highlight_mode dark
```

```
.highlight_mode light
```

To use a mix of colors suitable for both dark and light backgrounds:

```
.highlight_mode mixed
```

The mode can also be selected when launching the CLI using the `-dark-mode` and `-light-mode` command line arguments:

```
duckdb -dark-mode
```

```
duckdb -light-mode
```

8-Bit Colors

Since DuckDB 1.5, the CLI supports 8-bit colors corresponding to Xterm system colors:

```
.display_colors
```

```
darkred1 red darkred2 red3 red4 red1 brightred indianred1 ...
```

Dynamic Prompt

The default prompt is the following:

```
{max_length:50}{highlight_element:prompt_connect}{setting:connect_name_prefix}{highlight_
element:prompt}{setting:current_database_and_schema}{color:reset} D
```

While the session is connected to a remote database with `CONNECT`, `{setting:connect_name_prefix}` shows the remote database's name (or its URI, e.g., `quack:localhost`, for `CONNECT '<uri>'` and `-connect`); otherwise it is empty.

Return the Result of the Last Query Using _

You can use the _ (underscore) table to query the result of the last query:

```
SELECT 42 AS x;
```

```
┌───────┐
│   x   │
│ int32 │
├───────┤
│    42 │
└───────┘
```

```
FROM _;
```

```
┌───────┐
│   x   │
│ int32 │
├───────┤
│    42 │
└───────┘
```

If the last query did not return a result (e.g., because it performed an update operation), the CLI throws an error:

```
Binder Error:
Failed to query last result "_": no result available
```

Safe Mode

The DuckDB CLI client supports "safe mode". Safe mode can be enabled from the command line using the `-safe` argument:

```
duckdb -safe
```

Alternatively, it can be activated during a session using the `.safe_mode` dot command.

In safe mode, the CLI is prevented from accessing external files other than the database file that it was initially connected to and prevented from interacting with the host file system.

This has the following effects:

- The following dot commands are disabled:
  - `.cd`

  - `.excel`
  - `.import`
  - `.log`
  - `.once`
  - `.open`
  - `.output`
  - `.read`
  - `.sh`
  - `.system`

- Auto-complete no longer scans the file system for files to suggest as auto-complete targets.
- The `getenv` function is disabled.
- The `enable_external_access` option is set to `false`. This implies that:
  - `ATTACH` cannot attach to a database in a file.
  - `COPY` cannot read to or write from files.
  - Functions such as `read_csv`, `read_parquet`, `read_json`, etc. cannot read from an external source.

Once safe mode is activated, it cannot be deactivated in the same DuckDB CLI session.

For more information on running DuckDB in secure environments, see the "Securing DuckDB" page.

Autocomplete

The shell offers context-aware autocomplete of SQL queries through the `autocomplete` extension. autocomplete is triggered by pressing `Tab`.

Multiple autocomplete suggestions can be present. You can cycle forwards through the suggestions by repeatedly pressing `Tab`, or `Shift+Tab` to cycle backwards. autocompletion can be reverted by pressing ESC twice.

The shell autocompletes four different groups:

- Keywords
- Table names and table functions
- Column names and scalar functions
- File names

The shell looks at the position in the SQL statement to determine which of these autocompletions to trigger. For example:

```
SELECT s
```

```
student_id
```

```
SELECT student_id F
```

```
FROM
```

```
SELECT student_id FROM g
```

```
grades
```

```
SELECT student_id FROM 'd
```

```
'data/
```

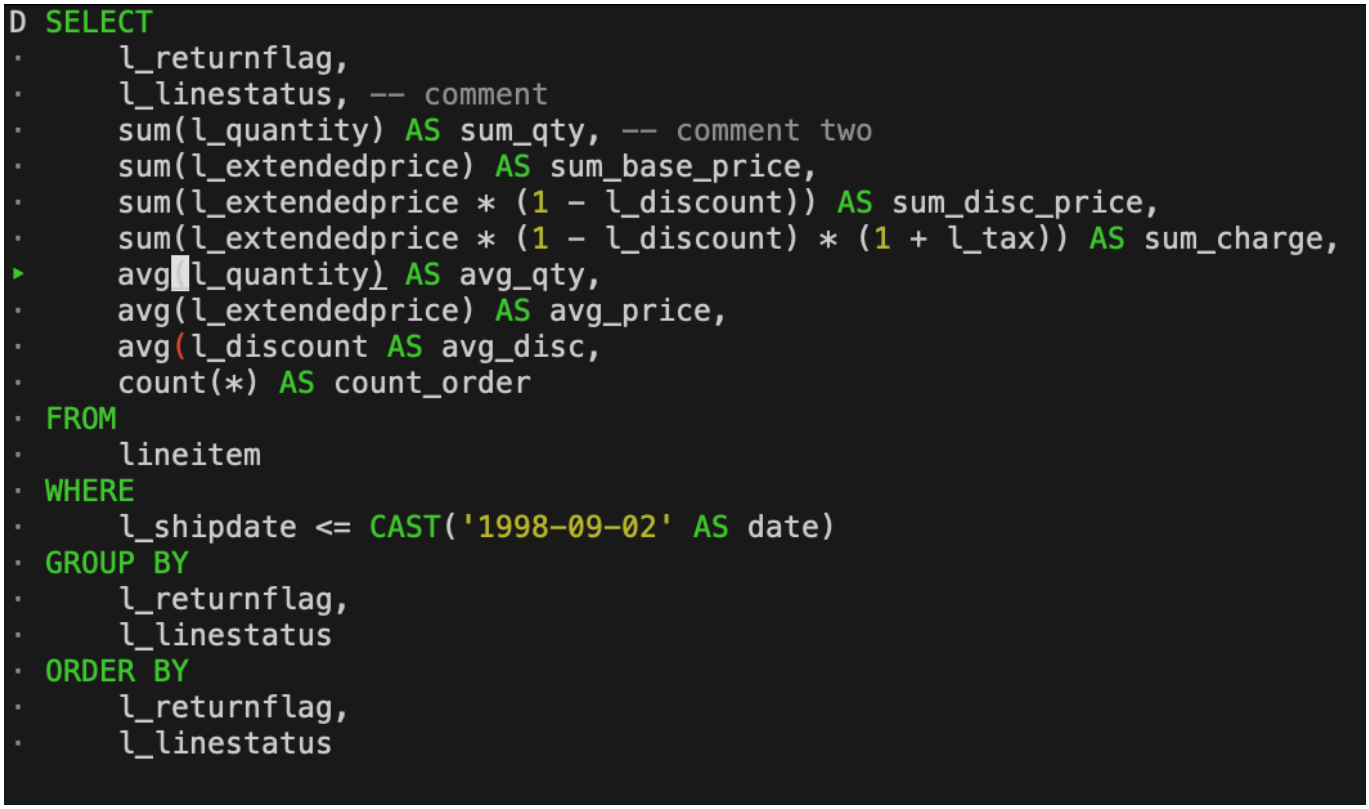
```
SELECT student_id FROM 'data/
```

```
'data/grades.csv
```

Syntax Highlighting

> Syntax highlighting in the CLI is available for macOS, Linux and Windows.

SQL queries that are written in the shell are automatically highlighted using syntax highlighting.



There are several components of a query that are highlighted in different colors. The colors can be configured using the `.highlight_colors` dot command: Syntax highlighting can also be disabled entirely using the `.highlight off` command.

Below is a list of query components that can be configured.

Type	Component	Default color
Keywords	`keyword`	`green`
Numeric constants	`numeric_constant`	`yellow`
String constants	`string_constant`	`yellow`
Comments	`comment`	`gray`
Errors	`error`	`red`
Continuation	`continuation`	`gray`
Continuation (Selected)	`continuation_selected`	`green`

Colors are specified by name. The following basic colors are supported by all terminals:

`black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `brightgray`, `gray`, `brightred`, `brightgreen`, `brightyellow`, `brightblue`, `brightmagenta`, `brightcyan`, `white`

In addition, 8-bit colors can be used. Run `.display_colors` to list all available color names, see 8-Bit Colors.

The optional intensity can be `standard`, `bold`, `underline` or `bold_underline`.

For example, here is an alternative set of syntax highlighting colors:

```
.highlight_colors keyword brightred bold
.highlight_colors numeric_constant white
.highlight_colors string_constant white
.highlight_colors comment cyan
.highlight_colors error yellow
.highlight_colors continuation blue
.highlight_colors continuation_selected brightblue
```

If you wish to start up the CLI with a different set of colors every time, you can place these commands in the `~/.duckdbrc` file that is loaded on start-up of the CLI.

Error Highlighting

The shell has support for highlighting certain errors. In particular, mismatched brackets and unclosed quotes are highlighted in red (or another color if specified). This highlighting is automatically disabled for large queries. In addition, it can be disabled manually using the `.render_errors off` command.

Known Issues

Incorrect Memory Values on Old Linux Distributions and WSL 2

On Windows Subsystem for Linux 2 (WSL2), when querying the `max_memory` or `memory_limit` from the `duckdb_settings`, the values may be inaccurate on certain Ubuntu versions (e.g., 20.04 and 24.04). The issue also occurs on older distributions such as Red Hat Enterprise Linux 8 (RHEL 8):

Example:

```
FROM duckdb_settings() WHERE name LIKE '%mem%';
```

The output contains values larger than 1000 PiB:

name varchar	value varchar	description varchar	input_type varchar	scope varchar
max_memory	1638.3 PiB	The maximum memory of the system (e.g. 1GB)	VARCHAR	GLOBAL
memory_limit	1638.3 PiB	The maximum memory of the system (e.g. 1GB)	VARCHAR	GLOBAL

Go

Go Client

> The latest stable release of the DuckDB Go client bundles DuckDB {{ latest_version }}. Its own version tag encodes that DuckDB version; see Versioning below.

The DuckDB Go client, `duckdb-go`, is a SQL driver that conforms to Go's built-in `database/sql` interface, so DuckDB is used through the same API as any other Go SQL database. On top of `database/sql`, the client adds DuckDB-specific interfaces for the Appender, Apache Arrow, user-defined functions, and profiling. This page focuses on installation. The other pages in this section cover connecting and each feature in detail.

For general instructions on the `database/sql` interface, see the official documentation and the Go database access tutorial.

Installation

The client is a Go module. Add it to a project with `go get`, using the `/v2` major-version suffix:

```
go get github.com/duckdb/duckdb-go/v2
```

To copy the module's dependencies into the project's `vendor` directory, including the pre-built DuckDB libraries from `duckdb-go-bindings`, run `go mod vendor`.

DuckDB is written in C++, so the client uses cgo and requires a C compiler to build. By default it statically links a pre-built DuckDB library into the binary, so no separate DuckDB installation is needed. Pre-built libraries ship for macOS (amd64, arm64), Linux (amd64, arm64), and Windows (amd64). Other platforms, custom builds, and dynamic linking are covered in Troubleshoot.

Importing

To register the driver, import the package for its side effects with the blank identifier alongside `database/sql`:

```
import (
    "database/sql"

    _ "github.com/duckdb/duckdb-go/v2"
)
```

The import registers a driver named `duckdb` with `database/sql`. Code that calls the client's own types and functions directly, such as the Appender or user-defined functions, imports the package under its `duckdb` name instead of with the blank identifier:

```
import "github.com/duckdb/duckdb-go/v2"
```

Versioning

Starting with DuckDB 1.5.0, the `duckdb-go` version encodes the DuckDB version it bundles in its second semantic-versioning component. That component is the DuckDB `major`, `minor`, and `patch` numbers concatenated, with `minor` and `patch` each zero-padded to two digits: DuckDB 1.5.0 maps to `duckdb-go` v2.10500.x, and DuckDB {{ latest_version }} maps to {{ latest_package }}. The README has the full mapping table for earlier releases.

The DuckDB 1.4 Andium LTS release line is published as its own tags, which still follow the client's older versioning scheme. Select one in `go.mod` the same way as any other version; see the releases page for the available tags.

> This project moved from `github.com/marcboeker/go-duckdb` to `github.com/duckdb/duckdb-go` starting with v2.5.0. All versions prior to v2.5.0 use the old import paths. See Migrating from `marcboeker/go-duckdb` below.

Basic API Usage

Open a database with `sql.Open()`, passing the driver name `duckdb` and a data source name (DSN). An empty DSN, or the DSN `:memory:`, opens an in-memory database, and a file path opens (or creates) a persistent database. From there, `Exec`, `Query`, and `QueryRow` run statements exactly as they do for any `database/sql` driver:

```
package main

import (
    "database/sql"
    "errors"
    "fmt"
    "log"

    _ "github.com/duckdb/duckdb-go/v2"
)

func main() {
    db, err := sql.Open("duckdb", "")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    _, err = db.Exec(` CREATE TABLE people (id INTEGER, name VARCHAR)`)
    if err != nil {
        log.Fatal(err)
    }
    _, err = db.Exec(` INSERT INTO people VALUES (42, 'John')`)
    if err != nil {
        log.Fatal(err)
    }

    var (
        id   int
        name string
    )
    row := db.QueryRow(` SELECT id, name FROM people`)
    err = row.Scan(&id, &name)
    if errors.Is(err, sql.ErrNoRows) {
        log.Println("no rows")
    } else if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("id: %d, name: %s\n", id, name)
}
```

This example follows the client's `simple` example. Run Queries covers sending queries, binding parameters, and reading results in full.

Data Source Names

The DSN passed to `sql.Open()` is the database path followed by optional DuckDB configuration options as URL-style query parameters:

```
// In-memory database: an empty DSN and ":memory:" are equivalent.
db, err := sql.Open("duckdb", "")

// Persistent database, created if it does not exist.
```

```
db, err := sql.Open("duckdb", "/path/to/foo.db")

// Persistent database with configuration options.
db, err := sql.Open("duckdb", "/path/to/foo.db?access_mode=read_only&threads=4")
```

To run initialization steps, such as SET statements, before the first query, open the database with `sql.OpenDB()` and a `Connector`. See Connect for connectors, configuration, and the connection lifetime.

Build Tags

Some client features are gated behind Go build tags, passed to `go build` with the `-tags` flag, for example `go build -tags=duckdb_arrow`. The available tags:

Build tag	Enables
`duckdb_arrow`	The Apache Arrow interface. It is a heavy dependency, so it is opt-in.
`duckdb_use_lib`	Dynamically link against a DuckDB library on the system instead of statically linking the bundled one. See Troubleshoot.
`duckdb_use_static_lib`	Statically link against a custom DuckDB static library instead of the bundled one. See Troubleshoot.

Bundled Extensions

Every pre-built library statically links DuckDB's default extensions: ICU, JSON, Parquet, and Autocomplete. Automatic extension loading is also enabled, so other core extensions install and load on first use.

Migrating from `marcboeker/go-duckdb`

The project moved from `github.com/marcboeker/go-duckdb` to `github.com/duckdb/duckdb-go` with v2.5.0. To migrate a project, update the dependency and rewrite the import paths with `gofmt`:

```
# Update the dependency.
go get github.com/duckdb/duckdb-go/v2@v2.20000.0

# Rewrite the import paths.
gofmt -w -r '"github.com/marcboeker/go-duckdb/v2" -> "github.com/duckdb/duckdb-go/v2"' .

# If the mapping or arrowmapping submodules are used, also run:
gofmt -w -r '"github.com/marcboeker/go-duckdb/mapping" -> "github.com/duckdb/duckdb-go/v2/mapping"' .
gofmt -w -r '"github.com/marcboeker/go-duckdb/arrowmapping" ->
"github.com/duckdb/duckdb-go/v2/arrowmapping"' .

# Clean up.
go mod tidy
```

Moving to v2 also introduced a few breaking changes, including opt-in Arrow support and stricter JSON scanning. See Troubleshoot and the README for the complete list.

Further Reading

- Connect — opening in-memory and file-backed databases, DSN configuration, connectors, and the connection lifetime.
- Run Queries — `Exec`, `Query`, prepared statements, parameter binding, transactions, and scanning results into Go values.
- Import Data — bulk loading with the Appender and reading directly from Parquet, CSV, and JSON files.

- Handle Results — the Apache Arrow interface for columnar result exchange.
- Write User Defined Functions — scalar and table user-defined functions, and replacement scans.
- Profile and Monitor — query profiling and logging.
- Troubleshoot — linking, cgo, Windows setup, and other build and runtime issues.
- Clients Overview — the other client APIs DuckDB provides alongside Go.

Acknowledgements

We would like to thank Marc Boeker for the initial implementation of the DuckDB Go client and for his continued work on it as part of this joint effort with the DuckDB team.

Connect

Overview

The Go client works through the standard `database/sql` types: a `*sql.DB` is a pool of connections to a DuckDB database. This page covers opening in-memory and file-backed databases, passing DuckDB configuration options, running initialization steps with a `Connector`, the client's thread-safety and connection-lifetime rules, and closing a database cleanly.

Opening a Database

`sql.Open()` takes the driver name `duckdb` and a data source name (DSN). An empty DSN, or the DSN `:memory:`, opens an in-memory database; a file path opens a persistent database, creating the file if it does not exist:

```
// In-memory database: nothing is persisted to disk.
// An empty DSN and ":memory:" are equivalent.
db, err := sql.Open("duckdb", "")
if err != nil {
    log.Fatal(err)
}
defer db.Close()

// File-backed database: created if it does not exist.
db, err := sql.Open("duckdb", "/path/to/foo.db")
```

`sql.Open()` does not necessarily open a connection: it validates its arguments and returns a `*sql.DB` that opens connections lazily. Call `db.Ping()` to verify that the database is reachable.

Data Source Names

DuckDB configuration options are appended to the DSN as URL-style `name=value` query parameters, separated by `&`:

```
db, err := sql.Open("duckdb", "/path/to/foo.db?access_mode=read_only&threads=4")
```

The `access_mode=read_only` option opens a database file so that several processes can read it at once. The full list of options is on the Configuration page. Options can also be changed after connecting with a SET statement.

Running Initialization Steps with a Connector

To run setup statements before any query, or to configure DuckDB in code rather than in the DSN, create a `Connector` with `duckdb.NewConnector()` and open the database with `sql.OpenDB()`. The connector's second argument is a callback that runs against each new connection, which is where boot queries belong:

```
connector, err := duckdb.NewConnector("/path/to/foo.db?threads=4", func(execer driver.ExecerContext) error
{
    bootQueries := []string{
        "SET schema = 'main'",
        "SET search_path = 'main'",
    }
    for _, query := range bootQueries {
        if _, err := execer.ExecContext(context.Background(), query, nil); err != nil {
            return err
        }
    }
    return nil
})
if err != nil {
    log.Fatal(err)
}
defer connector.Close()

db := sql.OpenDB(connector)
defer db.Close()
```

Pass `nil` as the callback when no initialization is needed. A `Connector` is also the entry point for the Appender and Arrow interfaces, which operate on an individual connection obtained from it, and for replacement scans and log storage, which are registered on the `Connector` itself.

Connection-Local Operations

Some features are scoped to a single connection rather than the pool: profiling, registering user-defined functions, and the Appender all act on one connection. Check a connection out of the pool with `db.Conn()`, which returns a `*sql.Conn` that is not shared with other goroutines until it is closed:

```
conn, err := db.Conn(context.Background())
if err != nil {
    log.Fatal(err)
}
defer conn.Close()
```

The client also exposes two helpers that take a `*sql.Conn`: `duckdb.GetTableNames()` returns the table names referenced by a query, optionally fully qualified, and `duckdb.ConnId()` returns the underlying DuckDB connection id.

Thread Safety and Concurrency

A `*sql.DB` is safe for concurrent use by multiple goroutines: it manages its own pool of connections and hands each in-flight query a connection of its own. A single `*sql.Conn` checked out with `db.Conn()`, and the DuckDB Appender and Arrow handles built on one, are not safe for concurrent use. DuckDB runs queries on its own pool of native threads, sized by the `threads` option and shared across the pool. See the DuckDB concurrency documentation for how connections and threads interact.

Connection Lifetime

Temporary objects such as temporary tables are scoped to a connection. When code closes a connection, `database/sql` may return it to the pool as an idle connection rather than actually closing it, so a temporary table can outlive the code that created it. To make closing a connection tear it down, disable idle connections:

```
db.SetMaxIdleConns(0)
```

Closing a Database

DuckDB runs in the same process as the Go application, so all of its memory is held by that process. Closing the `database/sql` objects is what frees DuckDB's memory and, for a persistent database, flushes the write-ahead log to disk. Always close every object that has a `Close()` method, which `defer` makes convenient:

```
db, _ := sql.Open("duckdb", "")
defer db.Close()

conn, _ := db.Conn(context.Background())
defer conn.Close()

rows, _ := conn.QueryContext(context.Background(), "SELECT 42")
defer rows.Close() // or drain the rows until Next() returns false

connector, _ := duckdb.NewConnector("", nil)
defer connector.Close() // if the connector is passed to sql.OpenDB
```

Failing to close a persistent database can leave changes in the write-ahead log that are not synchronized to persistent storage. For long-running applications, closing results, connections, and databases promptly is essential to keep memory in check.

Further Reading

- Run Queries — using the database and connections opened here to send queries and read results.
- Configuration — the full list of DuckDB settings that can be passed in the DSN or the connector callback.
- Concurrency — how DuckDB handles multiple connections and threads.
- Troubleshoot — cgo, linking, and other build problems encountered when opening a database.

Import Data

Overview

The Go client offers two ways to load data into DuckDB in bulk: the high-performance Appender for inserting rows generated in Go, and reading directly from data files such as Parquet, CSV, and JSON. Both are described below. For inserting a handful of rows, an ordinary `INSERT` statement is fine; the options here are for volume.

Appender

The Appender is the fastest way to insert rows generated in Go, much faster than issuing `INSERT` statements. It operates on a raw DuckDB connection, so it is created from a `Connector` rather than a `*sql.DB`. Obtain a connection from the connector, then create the appender for a target table with `duckdb.NewAppenderFromConn()`, passing the schema (empty for the default) and table name:

```
connector, err := duckdb.NewConnector("test.db", nil)
if err != nil {
    log.Fatal(err)
}
defer connector.Close()

conn, err := connector.Connect(context.Background())
if err != nil {
    log.Fatal(err)
}
defer conn.Close()

// The table must already exist.
appender, err := duckdb.NewAppenderFromConn(conn, "", "users")
if err != nil {
```

```
    log.Fatal(err)
}
defer appender.Close()

// Each argument is one column value, in table column order.
if err := appender.AppendRow("Fred", int32(34)); err != nil {
    log.Fatal(err)
}
```

Each argument to `AppendRow()` is a column value in table-column order, and its Go type must match the column's DuckDB type; the example passes `int32(34)` for an `INTEGER` column. Call `AppendRow()` once per row.

> **Warning.** The appender buffers rows and flushes them to DuckDB in chunks, so a constraint violation such as a `NOT NULL` or foreign-key failure may surface only when the buffer is flushed rather than from the `AppendRow()` call that added the offending row. Call `Flush()` and check its error to force pending rows in before relying on them, and always check the error from `Close()`, which performs a final flush.

The following follows the client's `appender` example, which creates the table through a `*sql.DB` opened on the same connector, appends a row, then reads it back:

```
package main

import (
    "context"
    "database/sql"
    "log"

    "github.com/duckdb/duckdb-go/v2"
)

func main() {
    c, err := duckdb.NewConnector("", nil)
    if err != nil {
        log.Fatal(err)
    }
    defer c.Close()

    con, err := c.Connect(context.Background())
    if err != nil {
        log.Fatal(err)
    }
    defer con.Close()

    db := sql.OpenDB(c)
    defer db.Close()
    if _, err := db.Exec(` CREATE TABLE users (name VARCHAR, age INTEGER)`); err != nil {
        log.Fatal(err)
    }

    a, err := duckdb.NewAppenderFromConn(con, "", "users")
    if err != nil {
        log.Fatal(err)
    }

    if err := a.AppendRow("Fred", int32(34)); err != nil {
        log.Fatal(err)
    }

    // Close flushes any buffered rows before closing the appender.
    if err := a.Close(); err != nil {
        log.Fatal(err)
    }

    var (
```

```
        name string
        age  int
    )
    row := db.QueryRowContext(context.Background(), `SELECT name, age FROM users`)
    if err := row.Scan(&name, &age); err != nil {
        log.Fatal(err)
    }
    log.Printf("User: name=%s, age=%d", name, age)
}
```

Targeting Other Schemas and Column Subsets

The package provides more constructors for targeting other catalogs and schemas, selecting a subset of columns, or transforming batches as they are appended. Each takes a `driver.Conn` as its first argument:

- `duckdb.NewAppender(driverConn, catalog, schema, table)` appends to all columns of the specified table.
- `duckdb.NewTableAppender(driverConn, query, catalog, schema, table, columnNames)` runs an `INSERT`, `DELETE`, `UPDATE`, or `MERGE INTO` query for each buffered batch. It infers the input types from the named columns in the target table. Pass an empty column-name slice to use all columns.
- `duckdb.NewQueryAppender(driverConn, query, temporaryTable, columnTypes, columnNames)` also runs a query for each batch, but lets the caller specify the temporary input table's name, column types, and column names.

`duckdb.NewAppenderWithColumns(driverConn, catalog, schema, table, columnNames)` can also append a subset of columns, leaving other columns at their default values or `NULL`. It is retained mostly for backward compatibility. Prefer `NewTableAppender` for this case because its query-based appender is more performant.

Reading Data Files

DuckDB can read many file formats directly in SQL, which is often the simplest way to load external data: the Go client just sends the query. Because the Parquet, CSV, and JSON readers are among the bundled extensions, no extra installation is needed. Query a file in place with the matching table function:

```
rows, err := db.QueryContext(ctx, `SELECT * FROM read_parquet('data.parquet')`)
```

The same pattern reads CSV with `read_csv()` and JSON with `read_json()`. To load a file into a table rather than query it in place, wrap the reader in a `CREATE TABLE ... AS SELECT` or `COPY` statement sent with `Exec`:

```
_, err := db.ExecContext(ctx, `CREATE TABLE data AS SELECT * FROM read_parquet('data.parquet')`)
```

Further Reading

- Appender — the engine-level Appender interface that the Go `Appender` wraps.
- Data Import — DuckDB's full set of bulk-loading options.
- Run Queries — sending the `CREATE TABLE` and `COPY` statements the file readers build on.
- Connect — creating the `Connector` and connection the Appender operates on.

Run Queries

Overview

Because the Go client implements `database/sql`, queries run through the standard methods:

- `Exec` and `ExecContext` send statements that do not return rows, such as `INSERT` or DDL, and report the number of affected rows.
- `Query`, `QueryContext`, `QueryRow`, and `QueryRowContext` run statements that return rows and hand back a `*sql.Rows` (or a single `*sql.Row`) to scan into Go values.

The `Context` variants take a `context.Context` as their first argument, which is the recommended form because it lets a query be cancelled or time out. The sections below cover sending statements, binding parameters, prepared statements, transactions, and scanning results. For opening the database these run on, see Connect.

Sending Statements

Use `Exec` (or `ExecContext`) for a statement that does not return rows, such as `INSERT` or `UPDATE`. It returns a `sql.Result`, from which `RowsAffected()` reports how many rows changed:

```
_, err := db.ExecContext(ctx, `CREATE TABLE users (name VARCHAR, age INTEGER)`)
if err != nil {
    log.Fatal(err)
}

res, err := db.ExecContext(ctx, `INSERT INTO users VALUES ('marc', 99)`)
if err != nil {
    log.Fatal(err)
}
n, _ := res.RowsAffected()
log.Printf("inserted %d rows", n)
```

Binding Parameters

Values are bound to a statement's placeholders rather than formatted into the SQL string, which avoids SQL injection and lets DuckDB reuse a plan. DuckDB accepts positional (`?`) and numbered (`$1`, `$2`) placeholders; pass the values as trailing arguments in order:

```
rows, err := db.QueryContext(ctx, `
    SELECT name, age
    FROM users
    WHERE (name = ? OR name = ?) AND age > ?`,
    "macgyver", "marc", 30,
)
```

For named placeholders such as `$name`, wrap each argument with `sql.Named`:

```
row := db.QueryRowContext(ctx,
    "SELECT $age >= 18 AND $name = 'Alice'",
    sql.Named("age", minAge),
    sql.Named("name", name),
)
```

> **Warning.** Do *not* use prepared statements to insert large amounts of data into DuckDB. See Import Data for the Appender, which is far faster for bulk inserts.

Forcing a Parameter Type

The client infers the DuckDB type of a bound value from its Go type. When that inference is not what you want, wrap the value with `duckdb.Typed()` to pin the DuckDB logical type. This matters most for timestamps: a bare `time.Time` binds as `TIMESTAMP_TZ`, so binding against a `TIMESTAMP_NS` column needs the hint:

```
start := time.Date(2024, time.April, 5, 0, 0, 0, 0, time.UTC)
end := time.Date(2024, time.April, 6, 0, 0, 0, 0, time.UTC)

row := db.QueryRow(`
    SELECT count(*)
    FROM (VALUES (TIMESTAMP_NS '2024-04-05 12:00:00.000000001')) events(ts)
    WHERE ts >= ? AND ts < ?`,
    duckdb.Typed(start, duckdb.TYPE_TIMESTAMP_NS),
    duckdb.Typed(end, duckdb.TYPE_TIMESTAMP_NS),
)
```

`duckdb.Typed()` is a scalar binding hint; DuckDB validates the value against the requested type when the parameter is bound. See Troubleshoot for the details of how the client maps `time.Time` to DuckDB's timestamp types.

Prepared Statements

`db.PrepareContext()` compiles a statement once so it can be executed many times, which pays off in a loop. The returned `*sql.Stmt` has its own `Exec` and `Query` methods, and must be closed when no longer needed:

```
stmt, err := db.PrepareContext(ctx, `INSERT INTO users VALUES (?, ?)`)
if err != nil {
    log.Fatal(err)
}
defer stmt.Close()

for _, u := range users {
    if _, err := stmt.ExecContext(ctx, u.name, u.age); err != nil {
        log.Fatal(err)
    }
}
```

Transactions

`db.BeginTx()` starts a transaction and returns a `*sql.Tx` whose `Exec` and `Query` methods run inside it. Commit the transaction with `tx.Commit()`, or roll it back with `tx.Rollback()`:

```
tx, err := db.BeginTx(ctx, nil)
if err != nil {
    log.Fatal(err)
}

if _, err := tx.ExecContext(ctx, `INSERT INTO users VALUES ('gru', 25)`); err != nil {
    tx.Rollback()
    log.Fatal(err)
}

if err := tx.Commit(); err != nil {
    log.Fatal(err)
}
```

Scanning Results

`QueryContext` returns a `*sql.Rows` cursor. Advance it with `Next()`, read the current row's columns into pointers with `Scan()`, and always check `Err()` after the loop and close the cursor. `Scan` matches columns to the destination Go types by position:

```
rows, err := db.QueryContext(ctx, `SELECT name, age FROM users WHERE age > ?`, 30)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()

for rows.Next() {
    var (
        name string
        age  int
    )
    if err := rows.Scan(&name, &age); err != nil {
        log.Fatal(err)
    }
    log.Printf("%s is %d years old", name, age)
}
```

```go
if err := rows.Err(); err != nil {
    log.Fatal(err)
}
```

When the column set is not known ahead of time, `rows.Columns()` returns the column names, and scanning into a slice of `any` reads each column into a `driver.Value`. Remember that a scanned value can be `nil` for a SQL `NULL`.

Reading Nested and Composite Types

DuckDB's nested types, such as `LIST`, `STRUCT`, and `MAP`, and its JSON type, are scanned through the generic `duckdb.Composite[T]` wrapper. Choose the Go type parameter to match the SQL shape, then read the decoded value with `Get()`:

```go
// A JSON (or LIST) value read into a slice.
var arr duckdb.Composite[[]any]
row := db.QueryRow(` SELECT json_array('foo', 'bar')`)
if err := row.Scan(&arr); err != nil {
    log.Fatal(err)
}
log.Printf("first element: %s", arr.Get()[0])

// A JSON object (or STRUCT) value read into a map.
var obj duckdb.Composite[map[string]any]
row = db.QueryRow(` SELECT '{"family": "anatidae", "coolness": 42.42}'::JSON`)
if err := row.Scan(&obj); err != nil {
    log.Fatal(err)
}
log.Printf("family: %s", obj.Get()["family"])
```

This follows the client's `json` example. To read whole result sets as columnar batches instead of row by row, see Handle Results.

Further Reading

- Handle Results — reading results as Apache Arrow record batches instead of row by row.
- Import Data — the Appender, the recommended alternative to prepared statements for bulk inserts.
- Prepared Statements — DuckDB's SQL-level support for the parameterized queries used here.
- Connect — opening the database that these statements run on.

Handle Results

Overview

DuckDB is columnar, and beyond reading a result set row by row through `database/sql`, the Go client can exchange whole results with Apache Arrow as a stream of record batches, and register an Arrow stream as a queryable view. Both are described below.

Enabling the Arrow Interface

The DuckDB Arrow interface is a heavy dependency, so starting with v2 it is opt-in. Enable it with the `duckdb_arrow` build tag:

```
go build -tags=duckdb_arrow
```

Without the tag, the Arrow types and functions described here are not compiled in. The interface builds on `apache/arrow-go`, so its `array` and `arrow` packages provide the record and schema types.

Reading a Result as Arrow

Wrap a raw DuckDB connection with `duckdb.NewArrowFromConn()`, then call `QueryContext()` to run a query and get back an `array.RecordReader`. Iterate it with `Next()`, reading one `arrow.RecordBatch` at a time, and release it when done:

```
connector, err := duckdb.NewConnector("", nil)
if err != nil {
    log.Fatal(err)
}
defer connector.Close()

conn, err := connector.Connect(context.Background())
if err != nil {
    log.Fatal(err)
}
defer conn.Close()

ar, err := duckdb.NewArrowFromConn(conn)
if err != nil {
    log.Fatal(err)
}

reader, err := ar.QueryContext(context.Background(), "SELECT * FROM generate_series(1, 10)")
if err != nil {
    log.Fatal(err)
}
defer reader.Release()

for reader.Next() {
    record := reader.RecordBatch()
    // Process one record batch at a time.
    log.Printf("%d rows", record.NumRows())
}
if err := reader.Err(); err != nil {
    log.Fatal(err)
}
```

`QueryContext()` accepts bound parameters as trailing arguments, the same as `database/sql` queries.

Registering an Arrow Stream as a View

An Arrow stream produced elsewhere in a program, an `array.RecordReader`, can be registered as a DuckDB view and queried in SQL. `RegisterView()` takes the reader and a view name, and returns a `release` function that unregisters the view; call it when the view is no longer needed:

```
release, err := ar.RegisterView(reader, "my_arrow_view")
if err != nil {
    log.Fatal(err)
}
defer release()

result, err := ar.QueryContext(context.Background(),
    "SELECT count(*) FROM my_arrow_view")
```

DuckDB scans the registered view like any other table, so it can be filtered, joined, and aggregated in SQL.

> **Warning.** Arrow connections are not safe for concurrent use, and they do not benefit from `database/sql` connection pooling. Use one Arrow handle per connection, and do not share it across goroutines.

Further Reading

- Run Queries — sending the queries whose results this page reads, and reading them row by row.

- Build Tags — the `duckdb_arrow` tag that enables the Arrow interface, and the other build tags.
- Write User Defined Functions — replacement scans, another way to expose external data to SQL.

Write User Defined Functions

Overview

The Go client can register user-defined functions written in Go: scalar functions that map input rows to a single output value, and table functions that produce a set of rows. It can also register a replacement scan that rewrites an unresolved table name into a query. Scalar and table functions are registered on a connection checked out with `db.Conn()`, while a replacement scan is registered on a `Connector`; in each case the function is then callable from SQL like any built-in. The sections below build and register each.

Scalar Functions

A scalar function is a Go type that implements the `duckdb.ScalarFunc` interface, which has two methods:

- `Config()` returns a `duckdb.ScalarFuncConfig` declaring the input parameter types, the result type, and, for a variadic function, the variadic parameter type. Types are built with `duckdb.NewTypeInfo()` and the composite constructors such as `duckdb.NewListInfo()`.
- `Executor()` returns a `duckdb.ScalarFuncExecutor` holding the callback. The simplest form is `RowExecutor`, a `func(values []driver.Value) (any, error)` called once per row.

Register an instance on a connection with `duckdb.RegisterScalarUDF()`. The following defines `my_length(VARCHAR) -> INTEGER`, adapted from the client's `scalar_udf` example:

```
type varcharLen struct{}

func (*varcharLen) Config() duckdb.ScalarFuncConfig {
    inputType, err := duckdb.NewTypeInfo(duckdb.TYPE_VARCHAR)
    check(err)
    resultType, err := duckdb.NewTypeInfo(duckdb.TYPE_INTEGER)
    check(err)

    return duckdb.ScalarFuncConfig{
        InputTypeInfos: []duckdb.TypeInfo{inputType},
        ResultTypeInfo: resultType,
    }
}

func (*varcharLen) Executor() duckdb.ScalarFuncExecutor {
    return duckdb.ScalarFuncExecutor{
        RowExecutor: func(values []driver.Value) (any, error) {
            return int32(len(values[0].(string))), nil
        },
    }
}

func main() {
    db, err := sql.Open("duckdb", "")
    check(err)
    defer db.Close()

    conn, err := db.Conn(context.Background())
    check(err)
    defer conn.Close()

    var udf *varcharLen
    check(duckdb.RegisterScalarUDF(conn, "my_length", udf))
```

```
    var length int32
    row := db.QueryRow(` SELECT my_length('hello world')`)
    check(row.Scan(&length)) // 11
}
```

Once registered, the function is called from SQL by the name passed to `RegisterScalarUDF()`:

```
SELECT my_length('hello world');
```

Set `VariadicTypeInfo` in the config to accept a variable number of trailing arguments. The field takes a `duckdb.TypeInfo`, so use `duckdb.NewTypeInfo(duckdb.TYPE_ANY)` to accept arguments of any type:

```
anyType, err := duckdb.NewTypeInfo(duckdb.TYPE_ANY)
check(err)

config := duckdb.ScalarFuncConfig{
    VariadicTypeInfo: anyType,
}
```

To register several overloads under one name, for example one taking a `VARCHAR` and another taking a `LIST`, implement each as its own `ScalarFunc` and register them together with `duckdb.RegisterScalarUDFSet()`:

```
check(duckdb.RegisterScalarUDFSet(conn, "my_length", varcharUDF, listUDF))
```

For a callback that processes a whole data chunk at once rather than a row at a time, set `ChunkContextExecutor` in the executor instead of `RowExecutor`; the `scalar_udf` example demonstrates both.

Table Functions

A table function produces rows and is queried in the `FROM` clause. It is described by a `duckdb.RowTableFunction`, whose `Config` declares the argument types and whose `BindArguments` callback is invoked once per call to bind the arguments and return a *table source*. The table source drives execution through its methods:

- `ColumnInfos()` declares the result columns, each a `duckdb.ColumnInfo` with a name and a `TypeInfo`.
- `Init()` runs once before the scan.
- `FillRow()` is called repeatedly, filling one `duckdb.Row` per call with `duckdb.SetRowValue()` and returning `true` while there are more rows, `false` to end the scan.
- `Cardinality()` returns an optional row-count estimate.

Register the function with `duckdb.RegisterTableUDF()`. The following defines `increment(BIGINT)`, which returns one row per integer up to the argument, adapted from the client's `table_udf` example:

```
type incrementTableUDF struct {
    tableSize  int64
    currentRow int64
}

func bindTableUDF(namedArgs map[string]any, args ...any) (duckdb.RowTableSource, error) {
    return &incrementTableUDF{
        currentRow: 0,
        tableSize:  args[0].(int64),
    }, nil
}

func (udf *incrementTableUDF) ColumnInfos() []duckdb.ColumnInfo {
    t, err := duckdb.NewTypeInfo(duckdb.TYPE_BIGINT)
    check(err)
    return []duckdb.ColumnInfo{{Name: "result", T: t}}
}

func (udf *incrementTableUDF) Init() {}
```

```go
func (udf *incrementTableUDF) FillRow(row duckdb.Row) (bool, error) {
    if udf.currentRow+1 > udf.tableSize {
        return false, nil // no more rows: end the scan
    }
    udf.currentRow++
    err := duckdb.SetRowValue(row, 0, udf.currentRow)
    return true, err
}

func (udf *incrementTableUDF) Cardinality() *duckdb.CardinalityInfo {
    return &duckdb.CardinalityInfo{Cardinality: uint(udf.tableSize), Exact: true}
}

func main() {
    db, err := sql.Open("duckdb", "")
    check(err)
    defer db.Close()

    conn, err := db.Conn(context.Background())
    check(err)
    defer conn.Close()

    t, err := duckdb.NewTypeInfo(duckdb.TYPE_BIGINT)
    check(err)
    udf := duckdb.RowTableFunction{
        Config:        duckdb.TableFunctionConfig{Arguments: []duckdb.TypeInfo{t}},
        BindArguments: bindTableUDF,
    }
    check(duckdb.RegisterTableUDF(conn, "increment", udf))

    rows, err := db.QueryContext(context.Background(), `SELECT * FROM increment(100)`)
    check(err)
    defer rows.Close()
}
```

```sql
SELECT * FROM increment(100);
```

For a function whose scan can run on several threads, implement `duckdb.ParallelRowTableSource` and register a `duckdb.ParallelRowTableFunction` instead. Its `Init()` returns a `duckdb.ParallelTableSourceInfo` that sets `MaxThreads`, a `NewLocalState()` method provides per-thread state, and `FillRow()` receives that state so each thread can claim and emit its own range of rows. The client's `table_udf_parallel` example shows the full pattern.

Replacement Scans

A replacement scan intercepts a query that references a table name DuckDB cannot resolve and rewrites it into a table function call. Register one on a `Connector` with `duckdb.RegisterReplacementScan()`, passing a callback of type `duckdb.ReplacementScanCallback`. The callback receives the unresolved table name and returns the replacement function name and its arguments:

```go
connector, err := duckdb.NewConnector("", nil)
check(err)
defer connector.Close()

duckdb.RegisterReplacementScan(connector, func(tableName string) (string, []any, error) {
    // Treat any bare table name as a Parquet file of that name.
    return "read_parquet", []any{tableName + ".parquet"}, nil
})

db := sql.OpenDB(connector)
// SELECT * FROM foo now reads from read_parquet('foo.parquet').
```

Returning an error from the callback surfaces it as a query error. This is the mechanism DuckDB clients use to make external data sources appear as ordinary tables.

Further Reading

- Run Queries — calling registered functions from SQL.
- Handle Results — registering an Arrow stream as a queryable view, a related way to expose external data.
- Connect — checking out the connection that functions are registered on, and the connector for replacement scans.

Profile and Monitor

Overview

The Go client can observe query execution: it can retrieve per-operator profiling metrics after a query runs, and register a log storage that receives DuckDB's log messages. Both are described below.

Profiling

Profiling information is connection-local, so it is read from a `*sql.Conn` checked out with `db.Conn()`. Enable profiling on that connection with the `enable_profiling` setting, run the query to profile, then read the result with `duckdb.GetProfilingInfo()`:

```
db, err := sql.Open("duckdb", "")
check(err)
defer db.Close()

conn, err := db.Conn(context.Background())
check(err)
defer conn.Close()

// Do not print profiling output to stdout after each query.
_, err = conn.ExecContext(context.Background(), `SET enable_profiling = 'no_output'`)
check(err)

// Run the query to profile.
res, err := conn.QueryContext(context.Background(), `SELECT 42`)
check(err)
res.Close()

// Retrieve the metrics collected for that query.
info, err := duckdb.GetProfilingInfo(conn)
check(err)

// Optionally, turn profiling off again.
_, err = conn.ExecContext(context.Background(), `PRAGMA disable_profiling`)
check(err)
```

`GetProfilingInfo()` returns a `duckdb.ProfilingInfo`, a tree that mirrors DuckDB's query plan. Each node has a `Metrics` map of metric name to value and a `Children` slice of child nodes. Metrics are grouped and named in lowercase: the root's children are the root plan operator, whose metrics include `type` and `timing`, followed by one node per metric group, e.g., the `query` group with `total_time`. Walk the tree to read them:

```
for _, child := range info.Children {
    if opType, ok := child.Metrics["type"]; ok {
        log.Printf("operator: %s, timing: %s", opType, child.Metrics["timing"])
    }
}
```

All metrics are collected by default; to restrict them, set the `tracked_metrics` option. See the Profiling page and the metrics list for the available metric names.

Log Storage

DuckDB can forward its log messages to a custom storage registered from Go. Register one on a `Connector` with `duckdb.RegisterLogStorage()`, giving it a name and a `duckdb.LoggerCallbacks` whose `DefaultLoggerCallback` receives each entry's level, type, and message:

```
connector, err := duckdb.NewConnector("", nil)
check(err)
defer connector.Close()

err = duckdb.RegisterLogStorage(connector, "my_log_storage", duckdb.LoggerCallbacks{
    DefaultLoggerCallback: func(level, logType, logMsg string) {
        log.Printf("[%s] %s: %s", level, logType, logMsg)
    },
})
check(err)

// Enable logging, then point it at the registered storage.
db := sql.OpenDB(connector)
_, err = db.Exec(` CALL enable_logging(level = 'info')`)
check(err)
_, err = db.Exec(` SET logging_storage = 'my_log_storage'`)
check(err)
```

Logging is off by default, so enable it with `CALL enable_logging()` in addition to selecting the storage with `SET logging_storage`. Once both are set, DuckDB forwards all log calls to the callback.

Further Reading

- Profiling — DuckDB's query profiling output and the `enable_profiling` and `tracked_metrics` settings.
- Logging — enabling logging with `enable_logging` and querying log entries through `duckdb_logs`.
- Run Queries — running the queries whose metrics this page reads.
- Connect — checking out the connection that profiling operates on, and the connector that log storage is registered on.

Troubleshoot

Overview

This page collects common issues encountered when using the DuckDB Go client, together with their workarounds. Because the client uses cgo to embed DuckDB, most build problems are cgo or linker problems. If you run into a problem that is not covered here, search the client's issue tracker on GitHub.

Classifying DuckDB Errors

The client returns a `*duckdb.Error` for errors reported by DuckDB. Use `errors.As` to handle these errors by category without parsing their messages:

```
_, err := db.Exec(query)
if err != nil {
    var duckdbError *duckdb.Error
    if errors.As(err, &duckdbError) {
        switch duckdbError.Type {
        case duckdb.ErrorTypeCatalog:
            // Handle a missing table, view, or other catalog entry.
        case duckdb.ErrorTypeConstraint:
            // Handle a constraint violation.
        }
```

```
    }
}
```

`duckdb.Error.Type` is a `duckdb.ErrorType`. Other categories include `ErrorTypeBinder`, `ErrorTypeParser`, `ErrorTypeConversion`, `ErrorTypeIO`, `ErrorTypeOutOfRange`, and `ErrorTypeInvalidInput`. The `Msg` field contains the error message.

`undefined: conn` and Other cgo Errors

An `undefined: conn` error while building means the Go compiler has decided cgo is unavailable, so the DuckDB bindings were not compiled. There are two common causes:

- **Build tools are missing.** cgo needs a C compiler and toolchain. On Debian or Ubuntu, install them with:

  ```
  sudo apt-get update && sudo apt-get install build-essential
  ```

- **Cross-compiling disables cgo.** The Go compiler turns cgo off automatically when cross-compiling. Re-enable it and point at the right cross-compiler:

  ```
  CC=<c_cross_compiler> CGO_ENABLED=1 go build
  ```

Windows Setup

On Windows you need a compatible version of `gcc` and the necessary runtime libraries. One way is MSYS2: install it, then open an MSYS2 shell and install the UCRT64 toolchain:

```
pacman -S mingw-w64-ucrt-x86_64-gcc
```

Add `gcc` to the path, for example in PowerShell:

```
$env:PATH = "C:\msys64\ucrt64\bin;$env:PATH"
```

The package then compiles on Windows.

Linking DuckDB

By default the client statically links a pre-built DuckDB library into the binary, which increases the binary size but requires no separate DuckDB installation. Pre-built libraries ship for macOS (amd64, arm64), Linux (amd64, arm64), and Windows (amd64). If none of the pre-built libraries fit, link a custom library instead.

Linking a Custom Static Library

Build DuckDB from source. Starting with DuckDB 2.0, the `bundle-library` Makefile target and the merged `libduckdb_bundle.a` archive no longer exist: the build produces `libduckdb_static.a` plus one static archive per extension (e.g., `libparquet_extension.a`), and statically linked extensions have to be registered at startup. See Statically Linking Extensions for the archives, the required link order and the extension registration.

Then build the Go module against these archives with the `duckdb_use_static_lib` build tag, passing the archives in `CGO_LDFLAGS`, with the extension archives listed before `libduckdb_static.a`.

This is also how a local DuckDB checkout is validated from the Go client, and how the client's own `Makefile` and CI build. It is required to target platforms without a pre-built library, including FreeBSD, for which DuckDB publishes no bundled library.

Linking a Dynamic Library

Alternatively, link dynamically against a `libduckdb` shared library on the system with the `duckdb_use_lib` build tag. Download the shared library (`.so` on Linux, `.dylib` on macOS) from the DuckDB releases page, then build and run with the library on the loader path:

```
# On Linux.
CGO_ENABLED=1 CGO_LDFLAGS="-lduckdb -L/path/to/libs" go build -tags=duckdb_use_lib main.go
LD_LIBRARY_PATH=/path/to/libs ./main

# On macOS.
CGO_ENABLED=1 CGO_LDFLAGS="-lduckdb -L/path/to/libs" go build -tags=duckdb_use_lib main.go
DYLD_LIBRARY_PATH=/path/to/libs ./main
```

`TIMESTAMP` versus `TIMESTAMP_TZ`

In the C API, DuckDB stores both `TIMESTAMP` and `TIMESTAMP_TZ` as the same instant, a number of microseconds since 1970-01-01 UTC, without offset information. When a `time.Time` is passed to the client, it is converted to that instant with `UnixMicro()`, even for `TIMESTAMP_TZ`, and scanning either type back returns an instant, because SQL types do not model a per-value time zone.

A bare `time.Time` binds as `TIMESTAMP_TZ` by default. When a different timestamp type is required, for example binding against a `TIMESTAMP_NS` column, force the type with `duckdb.Typed()`:

```
row := db.QueryRow(query,
    duckdb.Typed(start, duckdb.TYPE_TIMESTAMP_NS),
    duckdb.Typed(end, duckdb.TYPE_TIMESTAMP_NS),
)
```

See Run Queries for the full example.

Scanning JSON Values

Starting with v2, scanning a JSON value directly into a `string` or `[]byte` is no longer supported. Scan into `any` or into the `duckdb.Composite` wrapper instead, as shown in Run Queries. If a plain string or byte result is needed and the JSON structure is not, cast the column to `::VARCHAR` or `::BLOB` in SQL before scanning.

Further Reading

- Go Client — installation, build tags, and the bundled extensions.
- Connect — opening databases and the connection lifetime, whose failures are often build or linking problems.
- Run Queries — parameter binding, including the `duckdb.Typed()` hint and JSON scanning.

Java (JDBC)

Java (JDBC) Client

> Installation To use the DuckDB Java (JDBC) client, visit the Java installation page.
>
> The latest stable version of the DuckDB Java (JDBC) client is {{ latest_version }}.

The DuckDB Java (JDBC) client lets Java applications query DuckDB through the standard JDBC API, extended with DuckDB-specific features for bulk loading, Apache Arrow interchange, user-defined functions, and profiling, and can open a DuckLake catalog directly from the JDBC URL. The driver also supports ahead-of-time compilation with GraalVM Native Image.

This page covers installation; the other pages in this section cover connecting and each feature in detail.

Installation

The DuckDB Java JDBC API can be installed from Maven Central. Please see the installation page for details.

> **Tip.** To try features before they reach a stable release, preview (nightly) builds of the JDBC driver are published as SNAPSHOT versions to DuckDB's snapshot repository at `https://duckdb-staging.duckdb.org/duckdb/duckdb-java/maven`. This is an object storage repository with no browseable web page; it is consumed directly by Maven. Add the repository to your build and depend on a `duckdb_jdbc` SNAPSHOT version. See the preview builds page for a full Maven example.

Basic API Usage

DuckDB's JDBC API implements the main parts of the standard Java Database Connectivity (JDBC) API, version 4.1. Describing JDBC is beyond the scope of this page, see the official documentation for details. Below we focus on the DuckDB-specific parts.

Refer to the externally hosted API Reference for more information about our extensions to the JDBC specification, or the Arrow Methods.

Opening a Connection

In JDBC, database connections are created through the standard `java.sql.DriverManager` class, using the `jdbc:duckdb:` JDBC URL prefix:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

Used on its own, `jdbc:duckdb:` opens an in-memory database. Appending a file name, for example `jdbc:duckdb:/tmp/my_database`, opens a persistent database instead.

To confirm that the driver is on the classpath and a connection can be opened, run a trivial query:

```
try (Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT 42")) {
    rs.next();
    System.out.println(rs.getInt(1)); // prints 42
}
```

Define Connections covers the connection lifecycle in full: the URL forms the driver accepts, DuckDB and driver configuration options, read-only mode, instance caching, threading, and shutdown.

Further Reading

- Define Connections — the JDBC URL forms, configuration options, instance caching, threading, and connection shutdown.
- Connect to a DuckLake — attach or open a DuckLake catalog from JDBC.
- Run Queries — sending queries with `Statement` and `PreparedStatement`, and reading DuckDB's nested types.
- Import Data — bulk-loading data with the Appender and the JDBC batch writer.
- Handle Results — Apache Arrow interchange, result streaming, and chunked results.
- Deploy as Native Image — building standalone executables with GraalVM Native Image.
- Clients Overview — the other client APIs DuckDB provides alongside JDBC.

Connect

Overview

Every DuckDB feature reachable from Java starts with a `Connection`. This page covers the JDBC URL forms the driver accepts, how DuckDB and driver options are passed and which of them win, how connections come to share (or not share) an underlying database instance, how DuckDB's threads relate to that instance, how to open or attach a DuckLake, and how to reach a remote DuckDB server over Quack.

Opening a Connection

In JDBC, database connections are created through the standard `java.sql.DriverManager` class. On modern JVMs the driver registers itself with the `DriverManager` automatically when the JDBC JAR is on the classpath (through Java's service-provider mechanism), so no setup is needed. If that does not happen for some reason, load the `DuckDBDriver` class explicitly to force registration before opening a connection:

```
Class.forName("org.duckdb.DuckDBDriver");
Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

In GraalVM Native Image executables, always use this explicit load: the service-provider discovery is not reliably visible to Native Image's static analysis.

To create a DuckDB connection, call `DriverManager` with the `jdbc:duckdb:` JDBC URL prefix, like so:

```
import java.sql.Connection;
import java.sql.DriverManager;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

To use DuckDB-specific features such as the Appender, cast the object to a `DuckDBConnection`:

```
import java.sql.DriverManager;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
```

Additional connections can be created using the `DriverManager`. The `DuckDBConnection#duplicate()` method opens another connection to the same database instance without re-reading the configuration:

```
DuckDBConnection conn2 = ((DuckDBConnection) conn).duplicate();
```

Its main purpose is to reach a **connection-private in-memory** database instance — the one created by the plain `jdbc:duckdb:` URL — which no other URL can address. On a named in-memory or a file-backed connection, `duplicate()` is effectively the same as opening a new connection on the same URL, because those instances are shared through the instance cache anyway.

Multiple connections to the same instance are allowed, but they must all be opened with the same options. Options are applied when the instance starts, as the first connection is opened, so a later connection that requests different options — a different `access_mode`, for example — is rejected. See Database Instances and Instance Caching.

JDBC URL Syntax

A DuckDB JDBC URL consists of the `jdbc:duckdb:` prefix, an optional database name, and an optional list of options separated by semicolons:

```
jdbc:duckdb:<database>;<key>=<value>;<key>=<value>
```

The database part determines what the connection opens:

JDBC URL	Opens
`jdbc:duckdb:`	A connection-private in-memory database.
`jdbc:duckdb:memory:`	The same as above, written explicitly.
`jdbc:duckdb:memory:<label>`	A named in-memory database that other connections can share.
`jdbc:duckdb:<path>`	A database file on disk, for example `jdbc:duckdb:/tmp/my_database`.
`jdbc:duckdb:ducklake:<metadata_path>`	A DuckLake catalog.

When using the `jdbc:duckdb:` URL alone, a **connection-private in-memory database** is created: the instance belongs to that single connection and cannot be shared, other than through `duplicate()`. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Java program). If you would like to access or create a persistent database, append its file name after the path.

Read-Only Connections

It is possible to open a DuckDB database file in **read-only** mode. This is for example useful if multiple Java processes want to read the same database file at the same time. To open an existing database file in read-only mode, set the connection property `duckdb.read_only` like so:

```
Properties readOnlyProperty = new Properties();
readOnlyProperty.setProperty("duckdb.read_only", "true");
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", readOnlyProperty);
```

Read-only mode can also be set through DuckDB's standard `access_mode` property, for example `access_mode=READ_ONLY`. If both `duckdb.read_only` and `access_mode` are supplied and disagree, the driver throws an error.

Configuration Options

DuckDB Settings

Any DuckDB configuration option can be supplied when a connection is opened: the driver hands every option it does not recognize as its own to DuckDB. Note that many of these settings can also be changed later on with a SET (or SET GLOBAL) statement, or the equivalent PRAGMA.

```
Properties connectionProperties = new Properties();
connectionProperties.setProperty("temp_directory", "/path/to/temp/dir/");
connectionProperties.setProperty("memory_limit", "4GB");
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", connectionProperties);
```

The available options are listed on the Configuration page and can also be queried at runtime with `duckdb_settings()`. The driver exposes the same list through the standard `Driver.getPropertyInfo()` call, which returns one `DriverPropertyInfo` per DuckDB setting alongside the driver's own options.

> **Note.** Options passed at connection time are applied as **global** settings on the database instance. A setting that only has session scope has to be applied with a SET statement after connecting; setting it at connection time fails with `Could not set option "<name>" as a global option`. To find which settings these are, query `duckdb_settings()` and filter on `scope = 'LOCAL'`; `search_path`, `schema`, and the `enable_profiling` / `profiling_mode` family are common examples.

Setting Options in the URL

The same options can be appended to the JDBC URL, separated by semicolons. This is the only way to configure DuckDB in tools that accept a connection string but no `Properties` object:

```
jdbc:duckdb:/tmp/my_database;threads=4;memory_limit=4GB;jdbc_stream_results=true
```

Each entry is a `key=value` pair and surrounding whitespace is trimmed. The driver splits the URL on `;` and each entry on `=`, so a value that itself contains a semicolon or an equals sign has to be passed in a `Properties` object instead.

Option Precedence

When the same option appears in both places, the value in the URL wins:

```
Properties props = new Properties();
props.setProperty("threads", "8");
// The connection is opened with threads = 4.
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database;threads=4", props);
```

Unsupported Options

An option name that is neither a DuckDB setting nor a driver option makes the connection fail. Set `jdbc_ignore_unsupported_options` to `true` to have the driver drop such names silently, which is useful for frameworks and BI tools that add JDBC properties of their own. The driver already discards two options that are known to be passed by mistake: Apache Spark's `path` and LibreOffice Base's `Type`.

JDBC-Specific Options

Alongside DuckDB's own settings, the driver recognizes a number of DuckDB-specific JDBC connection properties:

Property	Description
`duckdb.read_only`	Open the database in read-only mode.
`access_mode`	Standard DuckDB access mode: `READ_ONLY`, `READ_WRITE`, or `AUTOMATIC`.
`custom_user_agent`	Append a custom string to the user agent reported to DuckDB.
`jdbc_stream_results`	Stream result sets instead of materializing them. See Streaming Results.
`jdbc_auto_commit`	Set the default auto-commit mode for new connections.

Property	Description
`jdbc_pin_db`	Keep the database instance alive after its last connection closes. Disabled by default. Enabled by default, together with `jdbc_stream_results`, for `ducklake:` URLs, so that BI tools such as Metabase do not repeatedly close and re-create the instance or accidentally materialize large DuckLake datasets.
`jdbc_instance_cache`	Reuse the process-wide database instance for the same database. Enabled by default. See Database Instances and Instance Caching.
`jdbc_ignore_unsupported_options`	Silently ignore unsupported connection options instead of throwing an error.
`jdbc_jfr_memory_monitor`	Enable JFR memory monitoring for the connection's database instance. See Memory Monitoring with JFR.
`session_init_sql_file`	Run the SQL in the given file before handing over the connection. URL-only, see Running SQL at Connection Startup.
`session_init_sql_file_sha256`	Expected SHA-256 digest of the session init SQL file. URL-only.

Running SQL at Connection Startup

The `session_init_sql_file` option points the driver at a SQL file on the local file system that is executed before the connection is returned to the caller. It is intended for environments where you control the connection string but not the code that opens the connection, for example a BI tool that only offers a JDBC URL field:

```
jdbc:duckdb:/tmp/my_database;session_init_sql_file=/path/to/init.sql
```

The file can be split into a database part and a connection part with a marker comment:

```
-- Runs once, when the database instance is created.
SET memory_limit = '4GB';
CREATE OR REPLACE VIEW recent_events AS
    SELECT *
    FROM events
    WHERE ts > now() - INTERVAL 7 DAYS;

/* DUCKDB_CONNECTION_INIT_BELOW_MARKER */

-- Runs for every connection.
SET search_path = 'analytics';
```

Everything above the marker runs the first time the database is opened in the JVM process, and everything below it runs for every connection. A file without a marker is treated entirely as database initialization. For the unnamed in-memory database the database part runs for every connection, because each connection gets its own instance.

Because the option makes the connection string execute arbitrary SQL, the driver constrains it:

- It is read from the JDBC URL only and is not picked up from a `Properties` object.
- It has to be the first option in the connection string, and `session_init_sql_file_sha256`, if present, has to be the second. Neither may appear more than once.
- The file may not be larger than 1 MB.
- If `session_init_sql_file_sha256` is given and does not match the file's digest, opening the connection fails. Use it to detect a file that was modified after deployment.
- `DuckDBConnection.getSessionInitSQL()` returns the text that was executed, so an application can log or audit it.

> **Warning.** The SQL in the file runs with the full privileges of the connection. Treat the connection string and the file it points at as trusted input.

Database Instances and Instance Caching

The JDBC URL determines whether connections share a single underlying database instance or each get their own. By default, the driver caches the database instance so that connections addressing the same database reuse it.

The behavior depends on the form of the URL:

JDBC URL	Behavior
`jdbc:duckdb:` or `jdbc:duckdb:memory:`	A connection-private, uncached in-memory database. Each connection gets its own separate instance, and its data is not shared with any other connection.
`jdbc:duckdb:memory:<label>`	A named in-memory database. Connections using the same label share one cached instance, keyed by `:memory:<label>`.
`jdbc:duckdb:<path>`	A file-backed database. The instance is cached using the absolute path to the database file as the cache key, so connections that open the same file share one instance. On Windows and macOS the key is case-insensitive.

```java
// Two connections to the same named in-memory database point to the same database instance.
Connection conn1 = DriverManager.getConnection("jdbc:duckdb:memory:shared_db");
Connection conn2 = DriverManager.getConnection("jdbc:duckdb:memory:shared_db");
```

Instance caching is controlled by the `jdbc_instance_cache` connection property, which is enabled by default. Setting it to `false` creates an isolated instance per connection. Disabling the cache for a file-backed database may cause local file lock conflicts if another connection has the same file open, and pinning an uncached instance with `jdbc_pin_db` does not make it reusable.

Because configuration options apply to the instance rather than to the connection, only the connection that creates the instance can set them. A later connection that reaches a cached instance and asks for a different configuration fails with:

```
Connection Error:
Can't open a connection to same database file with a different configuration than existing connections
```

Pass the same options on every connection to a given database, or open the first connection with the intended configuration and create the rest with `duplicate()`.

Threads and Thread Pools

DuckDB runs queries on its own pool of native threads. This pool is independent of the JVM's threads: a single `Statement.executeQuery()` call can occupy every thread in the pool, and blocking JDBC calls from many application threads do not increase DuckDB's parallelism.

The pool belongs to the **database instance**, not to the `Connection`. Every connection to the same instance, whether created through `DriverManager` or `duplicate()`, submits work to the same pool. Two connections that resolve to different instances, as described in Database Instances and Instance Caching, each get a pool of their own.

The size of the pool is set by DuckDB's `threads` option, which defaults to the number of CPU cores. DuckDB reserves `external_threads` of that budget, one by default, for the thread that calls into the library, and starts `threads - external_threads` background threads.

In addition, DuckDB 2.0 runs a second pool of asynchronous threads dedicated to blocking I/O, such as HTTP requests to object storage (see Asynchronous I/O). Its size is set by the `async_threads` option, which defaults to four times the number of CPU threads, capped at 256. On a 16-core machine, that is 15 worker threads plus 64 asynchronous threads, roughly 80 native threads per database instance.

To bound the thread count, set both `threads` and `async_threads` when the instance is created:

```
jdbc:duckdb:/tmp/my_database;threads=4;async_threads=4
```

An already-running instance can be resized with SET statements, which affect every connection to that instance. Note that changing `threads` does not resize the asynchronous pool, so set `async_threads` explicitly as well:

```
SET threads = 4;
SET async_threads = 4;
```

Two consequences are worth calling out for Java applications:

- **Attached databases do not add threads.** ATTACH adds a catalog to the database instance the connection already belongs to. All attached databases share that instance's thread pools, memory limit and temporary directory. Only opening a *new* JDBC connection to a different database name creates a second instance, and with it a second set of pools.
- **Multiple instances multiply the thread count.** An application that opens a dozen private in-memory databases with `jdbc:duckdb:` on a 16-core machine starts more than 900 native threads, each with its own stack, and each instance has its own share of the memory limit.

> **Warning.** Each private (uncached) in-memory database creates its own instance with independent thread pools. An application that opens many connections with the default `jdbc:duckdb:` URL can therefore create a large number of operating system threads. Use a named in-memory database, share one instance through the instance cache, or set both the `threads` and `async_threads` options to bound the thread count, e.g., `jdbc:duckdb:;threads=4;async_threads=4`.

> **Note.** The asynchronous I/O pool is new in DuckDB 2.0. Compared to earlier versions, it raises the default per-instance thread count on a 16-core machine from roughly 15 to roughly 80, which makes bounding the number of instances and setting `threads` and `async_threads` explicitly more important. See the asynchronous I/O blog post for details.

Separately from DuckDB's pools, the driver starts a single daemon thread named `duckdb-query-cancel-scheduler-thread` in the JVM the first time it is loaded. It is shared by the whole process and is used to fire `Statement.setQueryTimeout()` cancellations. Applications that unload the driver, for example in an application server, can stop it with `DuckDBDriver.shutdownQueryCancelScheduler()`.

Connect to a DuckLake

DuckLake stores table data as Parquet files and its metadata in a catalog database, which can be a DuckDB file, a SQLite file, or a PostgreSQL server. From JDBC there are two ways to reach a lake: attach it to an ordinary connection, or open it directly as the connection's default database with the `ducklake:` URL form.

Attach a DuckLake

The ATTACH statement works from any JDBC connection and is the only way to pass DuckLake options such as DATA_PATH. Use it when the lake is created for the first time, or when the application also needs a local database of its own:

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:");
     Statement stmt = conn.createStatement()) {
    stmt.execute("ATTACH 'ducklake:metadata.ducklake' AS lake (DATA_PATH 'lake_files/')");
    stmt.execute("USE lake");

    stmt.execute("CREATE TABLE IF NOT EXISTS events (id INTEGER, payload VARCHAR)");
    stmt.execute("INSERT INTO events VALUES (1, 'hello'), (2, 'world')");

    try (ResultSet rs = stmt.executeQuery("SELECT count(*) FROM events")) {
        rs.next();
        System.out.println(rs.getLong(1));
    }
}
```

The `ducklake` extension is autoloaded on the first `ATTACH`; an explicit `INSTALL ducklake; LOAD ducklake;` is only needed on systems where autoloading is disabled.

Open a DuckLake Directly

Once a lake exists, it can be opened as the connection's default database by placing the DuckLake path after the `jdbc:duckdb:` prefix. Tables are then addressed without a catalog prefix and no `USE` statement is needed:

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:ducklake:metadata.ducklake");
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT id, payload FROM events ORDER BY id")) {
    while (rs.next()) {
        System.out.println(rs.getInt(1) + " " + rs.getString(2));
    }
}
```

The part after `ducklake:` selects the catalog database:

JDBC URL	Catalog database
`jdbc:duckdb:ducklake:<path>.ducklake`	A DuckDB file.
`jdbc:duckdb:ducklake:sqlite:<path>.sqlite`	A SQLite file.
`jdbc:duckdb:ducklake:postgres:<connection_string>`	A PostgreSQL server, for example `postgresql://user:password@host:5432/lake_catalog`.

`ATTACH` options cannot be supplied in the JDBC URL, so a lake opened this way uses the `DATA_PATH` that was recorded when it was created. Everything after the first `;` is still parsed as connection options, which means a PostgreSQL connection string must not itself contain a semicolon.

For `ducklake:` URLs the driver changes two defaults, so that BI tools such as Metabase, which open and close connections frequently, do not restart the database instance on every request or accidentally materialize a large table:

- `jdbc_pin_db` is enabled, keeping the instance alive after the last connection closes. Release it with `DuckDBDriver.releaseDB(url)` when the application shuts down.
- `jdbc_stream_results` is enabled. See Streaming Results for the consequences.

Either default can be overridden by setting the option explicitly, for example `jdbc:duckdb:ducklake:metadata.ducklake;jdbc_pin_db=false`.

Object Storage Credentials

When the data files live on S3, GCS, or another remote file system, the credentials are provided through a secret, created after the connection is opened:

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:");
     Statement stmt = conn.createStatement()) {
    stmt.execute("CREATE SECRET (TYPE s3, KEY_ID '<key>', SECRET '<secret>', REGION 'us-east-1')");
    stmt.execute("ATTACH 'ducklake:postgres:postgresql://user:password@host:5432/lake_catalog' AS lake "
                 + "(DATA_PATH 's3://my-bucket/lake/')");
    stmt.execute("USE lake");
    // ...
}
```

In tools that only offer a JDBC URL field, the same statements can be run from a file with the `session_init_sql_file` option. The lake is opened before the init file runs, which is fine: opening a lake only reads the catalog database, and the data files are first touched

when a table is queried. A temporary secret is visible to every connection on the instance, so it can go in either part of the file; putting it below the marker makes it independent of whether the instance was already initialized:

```
/* DUCKDB_CONNECTION_INIT_BELOW_MARKER */
CREATE OR REPLACE TEMPORARY SECRET s3_lake (
    TYPE s3,
    KEY_ID '<key>',
    SECRET '<secret>',
    REGION 'us-east-1'
);
```

```
jdbc:duckdb:ducklake:postgres:postgresql://user:password@host:5432/lake_catalog;session_init_sql_
file=/etc/duckdb/lake_init.sql
```

> **Warning.** The JDBC URL and the init file both contain credentials. Treat them as secrets in the same way as the database password.

Read-Only Lakes on Object Storage

When both the catalog database and the data files live in object storage, the whole lake can be attached read-only without any local files, a setup often called a "frozen" lake. This works for a DuckDB or SQLite catalog file served over `s3://`, `gcs://`, or `https://`. The catalog file is given as the ATTACH target with `TYPE ducklake`, and `READ_ONLY` is required because the remote file cannot be written:

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:");
     Statement stmt = conn.createStatement()) {
    stmt.execute("CREATE SECRET (TYPE s3, KEY_ID '<key>', SECRET '<secret>', REGION 'us-east-1')");
    stmt.execute("ATTACH 's3://my-bucket/lake/catalog.duckdb' AS lake (TYPE ducklake, READ_ONLY)");
    stmt.execute("USE lake");

    try (ResultSet rs = stmt.executeQuery("SELECT id, payload FROM events ORDER BY id")) {
        while (rs.next()) {
            System.out.println(rs.getInt(1) + " " + rs.getString(2));
        }
    }
}
```

A SQLite catalog is attached the same way, with the file name ending in `.sqlite`. A publicly hosted lake served over `https://` needs no secret at all, so its catalog can be attached directly, for example `ATTACH 'https://<host>/catalog.duckdb' AS lake (TYPE ducklake, READ_ONLY)`. The `DATA_PATH` recorded in the catalog must point at a location the reader can also reach, such as the same bucket or public host.

Once attached or opened, DuckLake tables behave like local tables, so the whole JDBC API applies to them, including the Appender. Note that DuckLake, like other lakehouse formats, does not support indexes, primary keys, or `UNIQUE` and `CHECK` constraints; see the DuckLake documentation for the full list of differences.

Connecting to a Quack Server

The Quack remote protocol turns a DuckDB instance into a server that other DuckDB instances can query over HTTP. The JDBC driver has no dedicated URL scheme for it: open a local, usually in-memory, DuckDB connection in the normal way and attach the remote server as a catalog.

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:");
     Statement stmt = conn.createStatement()) {
    stmt.execute("ATTACH 'quack:<hostname>' AS remote_db (TOKEN '<MY_QUACK_TOKEN>')");

    try (ResultSet rs = stmt.executeQuery("SELECT count(*) FROM remote_db.events")) {
        rs.next();
        System.out.println(rs.getLong(1));
    }
}
```

The token can also be stored in a secret, which keeps it out of the ATTACH statement:

```
stmt.execute("CREATE SECRET (TYPE quack, TOKEN '<MY_QUACK_TOKEN>')");
stmt.execute("ATTACH 'quack:<hostname>' AS remote_db");
```

Once attached, remote tables behave like local ones, so the whole JDBC API applies to them. Run `CONNECT remote_db` to route the whole session to the server, so that unqualified table names resolve against it and queries execute remotely. The local instance still holds its own thread pool and memory limit, and the connection is subject to the same instance rules as any other JDBC connection, so a client that only forwards queries to a server can be opened with a small `threads` value.

> For server setup, authentication and other ways to query a remote server, see the Quack documentation.

Closing Connections

DuckDB shuts down a database when its last open connection is closed. Closing the final connection checkpoints the database, which merges the write-ahead log (the `.wal` file) into the database file and then removes it. See Files Created by DuckDB for details on these files.

For the JDBC driver, "exiting normally" means that every `Connection` to the database has been closed with `Connection.close()` before the Java program ends. There is no separate shutdown method to call: closing all connections is sufficient. Use a try-with-resources block so that connections are closed even when an exception is thrown:

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database")) {
    // work with the connection
}
```

If the process is terminated without closing its connections, the write-ahead log is left in place. It is replayed the next time the database file is opened, so no committed data is lost, but the file is only compacted once the database is checkpointed on a clean shutdown. To force a checkpoint without closing the connection, run the `CHECKPOINT` statement.

Repeatedly opening and closing the last connection to a database means starting and shutting down the instance each time, including its thread pool. Applications that do this in a loop can keep the instance alive between connections with the `jdbc_pin_db` option and release it later with `DuckDBDriver.releaseDB(url)`.

Further Reading

- Run Queries — using the `Connection` to send queries and read result sets.
- Handle Results — the `jdbc_stream_results` option and other result-handling choices set at connection time.
- Configuration — the full list of DuckDB settings that can be passed as connection options.
- Files Created by DuckDB — the database, WAL, and temporary files that connection shutdown checkpoints and cleans up.
- Troubleshoot — workarounds for common connection and driver problems.

Import Data

Overview

The JDBC driver offers two ways to load data into DuckDB in bulk: the high-performance Appender and a JDBC batch writer. This page describes both and when to use each.

Appender

The Appender is available in the DuckDB JDBC driver via the `org.duckdb.DuckDBAppender` class. An appender is created from a `DuckDBConnection` with the `createAppender()` method, passing the table name and, optionally, the schema and catalog it is applied to. The Appender is flushed when the `close()` method is called.

Example:

```java
import java.sql.DriverManager;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
try (var stmt = conn.createStatement()) {
    stmt.execute("CREATE TABLE tbl (x INTEGER, y DOUBLE, s VARCHAR)");

    // using try-with-resources to automatically close the appender at the end of the scope
    try (var appender = conn.createAppender(DuckDBConnection.DEFAULT_SCHEMA, "tbl")) {
        appender.beginRow();
        appender.append(10);
        appender.append(3.2);
        appender.append("hello");
        appender.endRow();
        appender.beginRow();
        appender.append(20);
        appender.append(-8.1);
        appender.append("world");
        appender.endRow();
    }
}
```

An appender can also be created with `createAppender(tableName)` for the default schema, or with `createAppender(catalogName, schemaName, tableName)` to target a specific catalog. Buffered rows are written to the table when the appender is closed, or earlier by calling `flush()`. Within a row, `appendNull()` writes a `NULL` and `appendDefault()` writes the column's `DEFAULT` value.

Nested and collection types are appended within a row as well. For a `STRUCT` column, call `beginStruct()`, append each field value in declaration order, then `endStruct()`. For a `UNION` column, call `beginUnion(tag)` — where `tag` is the name of the target `UNION` member — append that member's value, then `endUnion()`. The following row targets a table declared as `CREATE TABLE nested_tbl (point STRUCT(x INTEGER, y INTEGER), value UNION(num INTEGER, str VARCHAR))`:

```java
try (var appender = conn.createAppender("nested_tbl")) {
    appender.beginRow();
    appender.beginStruct();      // enter the STRUCT column
    appender.append(1);          // field x
    appender.append(2);          // field y
    appender.endStruct();
    appender.beginUnion("str");  // select the UNION member named "str"
    appender.append("hello");    // its VARCHAR value
    appender.endUnion();
    appender.endRow();
}
```

For a `LIST` or `ARRAY` column, pass a Java array as a single value, optionally with a boolean null mask where `true` marks the corresponding element as `NULL`:

```java
try (var appender = conn.createAppender("list_tbl")) {
    appender.beginRow();
    appender.append(new int[] {1, 2, 3});               // a LIST value
    appender.append(new int[] {4, 5}, new boolean[] {false, true}); // second element is NULL
    appender.endRow();
}
```

For a `MAP` column, pass a Java `Map` as a single value; each entry is written as a key/value pair, and a `null` map is appended as SQL `NULL`. The following row targets a table declared as `CREATE TABLE map_tbl (m MAP(VARCHAR, INTEGER))`:

```java
try (var appender = conn.createAppender("map_tbl")) {
    appender.beginRow();
    appender.append(Map.of("a", 1, "b", 2));            // a MAP value
    appender.endRow();
}
```

Maximizing Appender Performance

The Appender is already the fastest way to bulk-load data, but a few properties of how it works determine its throughput:

- **Row batching is fixed and internal.** The Appender writes rows in fixed-size batches of 2,048 rows, one DuckDB vector, and flushes a batch automatically as soon as it fills. There is no configurable flush count, so adding an application-level batch size on top of the Appender does not improve performance.
- **Prefer primitive columns over complex types.** Nested types such as `STRUCT`, `LIST`, and `UNION` cost more per value than flat, primitive columns. A schema built from primitive columns appends fastest.
- **Avoid allocating a Java object per row.** Boxing values or constructing a new object for every row adds allocation and garbage-collection pressure. Prefer the primitive `append()` overloads and reuse buffers where you can.
- **Keep strings short where possible.** DuckDB stores strings of up to 12 bytes inline; longer strings are stored out of line and add a small overhead per value.

Single-Value Appender (Legacy)

An earlier appender, `org.duckdb.DuckDBSingleValueAppender`, created with `DuckDBConnection.createSingleValueAppender()`, writes one value at a time with typed `append()` overloads between `beginRow()` and `endRow()`. Both the class and the factory method are deprecated and retained only for backward compatibility.

> **Warning.** New code should use the `DuckDBAppender` described above, which is faster and appends nested and collection types. `DuckDBSingleValueAppender` is kept only for existing applications.

Batch Writer

The DuckDB JDBC driver offers batch write functionality. The batch writer supports prepared statements to mitigate the overhead of query parsing.

> The preferred method for bulk inserts is to use the Appender due to its higher performance. However, when using the Appender is not possible, the batch writer is available as an alternative.

Batch Writer with Prepared Statements

Prepared statements are the recommended way to use the batch writer, because the statement is parsed once and reused for every row. Bind each row's parameters, call `addBatch()` to queue the row, and call `executeBatch()` to write all queued rows in a single operation.

```java
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
PreparedStatement stmt = conn.prepareStatement("INSERT INTO test (x, y, z) VALUES (?, ?, ?);");

stmt.setObject(1, 1);
stmt.setObject(2, 2);
stmt.setObject(3, 3);
stmt.addBatch();

stmt.setObject(1, 4);
stmt.setObject(2, 5);
stmt.setObject(3, 6);
stmt.addBatch();

stmt.executeBatch();
stmt.close();
```

Batch Writer with Vanilla Statements

The batch writer also accepts plain SQL statements that carry their values inline, without parameters. Queue each statement with `addBatch(String)` and submit them together with `executeBatch()`. This avoids parameter binding but does not benefit from the reuse that prepared statements provide.

```
import java.sql.DriverManager;
import java.sql.Statement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
Statement stmt = conn.createStatement();

stmt.execute("CREATE TABLE test (x INTEGER, y INTEGER, z INTEGER)");

stmt.addBatch("INSERT INTO test (x, y, z) VALUES (1, 2, 3);");
stmt.addBatch("INSERT INTO test (x, y, z) VALUES (4, 5, 6);");

stmt.executeBatch();
stmt.close();
```

Further Reading

- Appender — the engine-level Appender interface that the Java `DuckDBAppender` wraps.
- Data Import — DuckDB's other bulk-loading options, including reading directly from CSV, Parquet, and JSON files.
- Run Queries — sending the `INSERT` and `CREATE TABLE` statements the batch writer builds on.
- Define Connections — obtaining the `DuckDBConnection` an Appender is created from.

Run Queries

Overview

Java applications send queries and read result sets through the familiar `Statement`, `PreparedStatement`, and `ResultSet` interfaces. This page covers sending queries and reading DuckDB's nested and composite types. For opening the `Connection` these run on, see Define Connections.

Sending Queries

DuckDB supports the standard JDBC methods to send queries and retrieve result sets. First a `Statement` object has to be created from the `Connection`, this object can then be used to send queries using `execute()` and `executeQuery()`. `execute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `executeQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below are two examples. See also the JDBC `Statement` and `ResultSet` documentations.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");

// create a table
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)");
// insert two items into the table
stmt.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2, 2)");

try (ResultSet rs = stmt.executeQuery("SELECT * FROM items")) {
```

```
    while (rs.next()) {
        System.out.println(rs.getString(1));
        System.out.println(rs.getInt(3));
    }
}
stmt.close();
```

```
jeans
1
hammer
2
```

DuckDB also supports prepared statements as per the JDBC API:

```
import java.sql.PreparedStatement;

try (PreparedStatement stmt = conn.prepareStatement("INSERT INTO items VALUES (?, ?, ?);")) {
    stmt.setString(1, "chainsaw");
    stmt.setDouble(2, 500.0);
    stmt.setInt(3, 42);
    stmt.execute();
    // more calls to execute() possible
}
```

> **Warning.** Do *not* use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

Reading Nested and Composite Types

DuckDB supports nested and composite column types such as `LIST`, `STRUCT`, `MAP`, `UNION`, and `ENUM`. The JDBC driver maps each of these to an idiomatic Java object. Read the values from a `ResultSet` with `getObject`, which returns the following types:

DuckDB type	Java type returned by `getObject`
`LIST`, `ARRAY`	`org.duckdb.DuckDBArray` (implements `java.sql.Array`)
`STRUCT`	`org.duckdb.DuckDBStruct` (implements `java.sql.Struct`)
`MAP`	`java.util.LinkedHashMap`
`UNION`	the resolved member value
`ENUM`	`String`

```
try (ResultSet rs = stmt.executeQuery("SELECT [1, 2, 3] AS l, {'a': 1, 'b': 2} AS s")) {
    while (rs.next()) {
        DuckDBArray list = (DuckDBArray) rs.getObject(1);
        Object[] values = (Object[]) list.getArray();

        DuckDBStruct struct = (DuckDBStruct) rs.getObject(2);
        Map<String, Object> fields = struct.getMap();
    }
}
```

The DuckDB result set also exposes typed accessors as extensions to the JDBC API, including `getArray(int)`, `getStruct(int)`, `getUuid(int)`, `getHugeint(int)`, and `getJsonObject(int)`. `DuckDBStruct.getMap()` returns the struct fields keyed by name.

`DuckDBArray.getResultSet()` exposes the list elements as an `org.duckdb.DuckDBArrayResultSet`, a read-only `ResultSet` with two columns: `INDEX`, the 1-based position of the element, and `VALUE`, the element itself. Iterate it like any other result set:

```java
try (ResultSet rs = stmt.executeQuery("SELECT [10, 20, 30] AS l")) {
    rs.next();
    DuckDBArray list = (DuckDBArray) rs.getObject(1);
    try (ResultSet elements = list.getResultSet()) {
        while (elements.next()) {
            int index = elements.getInt("INDEX"); // 1-based position
            int value = elements.getInt("VALUE"); // the element
            System.out.println(index + " -> " + value);
        }
    }
}
```

A JSON column is returned by `getObject(int)` (and `getJsonObject(int)`) as an `org.duckdb.JsonNode`, a lightweight wrapper over the raw JSON text. Test its type with `isArray()`, `isObject()`, `isString()`, `isNumber()`, `isBoolean()`, and `isNull()`, and recover the JSON source with `toString()`:

```java
try (ResultSet rs = stmt.executeQuery("SELECT '[1, 2, 3]'::JSON AS j")) {
    rs.next();
    JsonNode json = (JsonNode) rs.getObject(1);
    if (json.isArray()) {
        System.out.println(json); // [1, 2, 3]
    }
}
```

Binding Nested and Composite Parameters

To bind a `LIST`/`ARRAY`, `STRUCT`, or `MAP` value as a prepared-statement parameter, build it from the `Connection` and pass it to `setObject()`. Each factory method attaches the SQL type name the driver needs to marshal the value.

- `createArrayOf(typeName, elements)` — the standard JDBC method — returns an `org.duckdb.user.DuckDBUserArray` (a `java.sql.Array`) for a `LIST` or `ARRAY`, where `typeName` is the element type.
- `createStruct(typeName, attributes)` — the standard JDBC method — returns an `org.duckdb.user.DuckDBUserStruct` (a `java.sql.Struct`), where `typeName` is the `STRUCT` type and `attributes` are the field values in declaration order.
- `createMap(typeName, map)` — a DuckDB extension on `DuckDBConnection` — returns an `org.duckdb.user.DuckDBMap` (a `java.util.Map`) for a `MAP`, where `typeName` is the full map type such as `MAP(VARCHAR, INTEGER)`.

```java
import java.sql.Array;
import java.sql.Struct;
import java.util.Map;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");

// LIST / ARRAY
Array list = conn.createArrayOf("INTEGER", new Object[] {1, 2, 3});
try (PreparedStatement stmt = conn.prepareStatement("SELECT ?::INTEGER[]")) {
    stmt.setObject(1, list);
    stmt.execute();
}

// STRUCT
Struct point = conn.createStruct("STRUCT(x DOUBLE, y DOUBLE)", new Object[] {1.0, 2.0});

// MAP
Map<String, Integer> counts = conn.createMap("MAP(VARCHAR, INTEGER)", Map.of("a", 1, "b", 2));
```

Binding Spatial Values

The spatial extension builds its geometry types from STRUCT and LIST types, so they are bound with the same factory methods. A POINT_2D is a struct of two DOUBLE fields, and a LINESTRING is a list of such points. After loading the extension, build the value with createStruct() (or createArrayOf()) and cast the parameter to GEOMETRY; a GEOMETRY result comes back as Well-Known Binary through getBlob():

```
try (Statement stmt = conn.createStatement()) {
    stmt.execute("INSTALL spatial");
    stmt.execute("LOAD spatial");
}

// POINT_2D is a STRUCT(x DOUBLE, y DOUBLE)
Struct point = conn.createStruct("POINT_2D", new Object[] {41.1, 42.2});
try (PreparedStatement stmt = conn.prepareStatement("SELECT ?::POINT_2D::GEOMETRY")) {
    stmt.setObject(1, point);
    try (ResultSet rs = stmt.executeQuery()) {
        rs.next();
        Blob wkb = rs.getBlob(1); // the geometry as Well-Known Binary
    }
}

// A LINESTRING is a LIST of POINT_2D structs
Struct p1 = conn.createStruct("POINT_2D", new Object[] {0.0, 0.0});
Struct p2 = conn.createStruct("POINT_2D", new Object[] {1.0, 1.0});
Array line = conn.createArrayOf("POINT_2D", new Object[] {p1, p2});
```

Binding Temporal Parameters

Date and time parameters follow the same setObject() path. When you bind a java.time or java.sql temporal value, the driver wraps it in the matching internal holder — java.sql.Date/LocalDate become an org.duckdb.DuckDBDate, Timestamp/LocalDateTime become a DuckDBTimestamp, OffsetDateTime becomes a DuckDBTimestampTZ, and Time/LocalTime become a DuckDBTime — and marshals it to the corresponding DuckDB type:

```
import java.time.LocalDate;
import java.time.LocalDateTime;

try (PreparedStatement stmt = conn.prepareStatement("SELECT ?, ?")) {
    stmt.setObject(1, LocalDate.of(2024, 1, 15));
    stmt.setObject(2, LocalDateTime.of(2024, 1, 15, 12, 30));
    stmt.execute();
}
```

Each holder keeps its value relative to the Unix epoch (1970-01-01): DuckDBDate counts whole days through getDaysSinceEpoch(), while DuckDBTimestamp, DuckDBTimestampTZ, and DuckDBTime report microseconds through getMicrosEpoch() (for DuckDBTime, microseconds since midnight):

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.time.OffsetDateTime;
import org.duckdb.DuckDBDate;
import org.duckdb.DuckDBTime;
import org.duckdb.DuckDBTimestamp;
import org.duckdb.DuckDBTimestampTZ;

long days = new DuckDBDate(Date.valueOf("2024-01-15")).getDaysSinceEpoch();                    // 19737
long micros = new DuckDBTimestamp(Timestamp.valueOf("2024-01-15 12:30:00")).getMicrosEpoch();
long zonedMicros = new DuckDBTimestampTZ(OffsetDateTime.parse("2024-01-15T12:30:00Z")).getMicrosEpoch();
long timeMicros = new DuckDBTime(Time.valueOf("12:30:00")).getMicrosEpoch();
```

Inspecting Metadata

The driver implements the standard JDBC metadata interfaces, so an application can introspect the database, a result set, or a prepared statement's parameters without running catalog queries by hand.

- `Connection.getMetaData()` returns an `org.duckdb.DuckDBDatabaseMetaData`, a `java.sql.DatabaseMetaData` that reports database-wide capabilities and exposes catalog objects such as `getTables()`, `getColumns()`, and `getSchemas()` as result sets built from DuckDB's system tables.
- `ResultSet.getMetaData()` returns an `org.duckdb.DuckDBResultSetMetaData` describing the result columns — `getColumnCount()`, `getColumnName(int)`, `getColumnType(int)`, `getColumnTypeName(int)`, `getPrecision(int)`, and `getScale(int)`.
- `PreparedStatement.getParameterMetaData()` returns an `org.duckdb.DuckDBParameterMetaData` describing the statement's parameters (1-based), including `getParameterCount()`, `getParameterType(int)`, and `getParameterTypeName(int)`.

```
try (PreparedStatement stmt = conn.prepareStatement("SELECT * FROM items WHERE value > ?")) {
    stmt.setDouble(1, 10.0);
    ResultSetMetaData rsMeta = stmt.getMetaData();
    for (int i = 1; i <= rsMeta.getColumnCount(); i++) {
        System.out.println(rsMeta.getColumnName(i) + " : " + rsMeta.getColumnTypeName(i));
    }
    System.out.println("parameters: " + stmt.getParameterMetaData().getParameterCount());
}
```

As a DuckDB extension, `DuckDBResultSetMetaData.getReturnType()` reports what a statement returns as an `org.duckdb.StatementReturnType` — `QUERY_RESULT`, `CHANGED_ROWS`, or `NOTHING`:

```
import org.duckdb.DuckDBResultSetMetaData;
import org.duckdb.StatementReturnType;

DuckDBResultSetMetaData meta = (DuckDBResultSetMetaData) rs.getMetaData();
StatementReturnType returnType = meta.getReturnType(); // QUERY_RESULT
```

For a `DECIMAL` column, `getPrecision(int)` and `getScale(int)` report the width and scale that DuckDB records for the type in a `DuckDBColumnTypeMetaData`. For the `value DECIMAL(10, 2)` column of the `items` table, they return `10` and `2`:

```
try (ResultSet rs = stmt.executeQuery("SELECT value FROM items")) {
    ResultSetMetaData rsMeta = rs.getMetaData();
    int precision = rsMeta.getPrecision(1); // 10
    int scale = rsMeta.getScale(1);         // 2
    System.out.println("DECIMAL(" + precision + ", " + scale + ")");
}
```

Further Reading

- Handle Results — Apache Arrow interchange, result streaming, and chunked results beyond the standard `ResultSet`.
- Import Data — the Appender and batch writer, the recommended alternatives to prepared statements for bulk inserts.
- Prepared Statements — DuckDB's SQL-level support for the parameterized queries used here.
- Define Connections — opening the `Connection` that these statements run on.

Handle Results

Overview

Beyond the standard JDBC `ResultSet`, the DuckDB driver can hand results to Apache Arrow, stream large results instead of materializing them, and expose raw columnar data chunks. This page covers each of these result-handling options.

Arrow Methods

The DuckDB result set integrates with Apache Arrow through the Java Arrow bindings. A `DuckDBResultSet` can export its rows as an Arrow stream for consumption by Arrow-based tools, and an Arrow stream produced elsewhere can be registered on a connection and queried as a DuckDB table. Refer to the API Reference for the type signatures.

Arrow Export

The following example exports an Arrow stream from a query result and consumes it using the Java Arrow bindings. Each batch is loaded in turn, and its vectors are read from the `VectorSchemaRoot`.

```
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowReader;
import org.duckdb.DuckDBResultSet;

try (var conn = DriverManager.getConnection("jdbc:duckdb:");
    var stmt = conn.prepareStatement("SELECT * FROM generate_series(2000)");
    var resultset = (DuckDBResultSet) stmt.executeQuery();
    var allocator = new RootAllocator()) {
    try (var reader = (ArrowReader) resultset.arrowExportStream(allocator, 256)) {
        while (reader.loadNextBatch()) {
            System.out.println(reader.getVectorSchemaRoot().getVector("generate_series"));
        }
    }
}
```

Arrow Import

The following example consumes an Arrow stream produced by the Java Arrow bindings and registers it on a DuckDB connection with `registerArrowStream()`. Once registered, the stream is addressed by the name passed to that method and can be queried like any other table.

```
import org.apache.arrow.c.ArrowArrayStream;
import org.apache.arrow.c.Data;
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowStreamReader;
import org.duckdb.DuckDBConnection;
import org.duckdb.DuckDBResultSet;

// Arrow binding
try (var allocator = new RootAllocator();
     ArrowStreamReader reader = null; // should not be null of course
     var arrow_array_stream = ArrowArrayStream.allocateNew(allocator)) {
    Data.exportArrayStream(allocator, reader, arrow_array_stream);

    // DuckDB setup
    try (var conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:")) {
        conn.registerArrowStream("asdf", arrow_array_stream);

        // run a query
        try (var stmt = conn.createStatement();
             var rs = (DuckDBResultSet) stmt.executeQuery("SELECT count(*) FROM asdf")) {
            while (rs.next()) {
                System.out.println(rs.getInt(1));
            }
        }
    }
}
```

Streaming Results

Result streaming is opt-in in the JDBC driver. Enable it by setting the `jdbc_stream_results` config to `true` before running a query. The easiest way to do that is to pass it in the `Properties` object.

```
Properties props = new Properties();
props.setProperty(DuckDBDriver.JDBC_STREAM_RESULTS, String.valueOf(true));

Connection conn = DriverManager.getConnection("jdbc:duckdb:", props);
```

Chunked Results

The JDBC driver can return a query result as a lazily fetched sequence of columnar data chunks via the `org.duckdb.DuckDBChunkedResult` class. This exposes the C API's `duckdb_fetch_chunk` function to Java and avoids the per-row overhead required by the JDBC `ResultSet` interface. Chunk contents are read through the same `DuckDBDataChunkReader` API that is used by user-defined functions.

Call `query()` on a `DuckDBPreparedStatement` to obtain a `DuckDBChunkedResult`, then advance through the chunks with `nextChunk()` and read each column vector by index.

Example:

```
import java.sql.DriverManager;
import org.duckdb.DuckDBConnection;
import org.duckdb.DuckDBPreparedStatement;
import org.duckdb.DuckDBChunkedResult;
import org.duckdb.DuckDBDataChunkReader;
import org.duckdb.DuckDBReadableVector;

try (DuckDBConnection conn = DriverManager
        .getConnection("jdbc:duckdb:")
        .unwrap(DuckDBConnection.class);
     DuckDBPreparedStatement ps = conn.prepare("SELECT ? AS col1")) {

    // statement parameters are 1-based
    ps.setInt(1, 42);

    try (DuckDBChunkedResult res = ps.query()) {

        // advance to the next chunk, returns true on success
        while (res.nextChunk()) {

            // get the current chunk from the result
            DuckDBDataChunkReader chunk = res.chunk();

            // iterate over the chunk columns, all indices are 0-based
            for (long col = 0; col < chunk.columnCount(); col++) {

                // get a vector for the specified column
                DuckDBReadableVector vector = chunk.vector(col);

                // iterate over the vector rows
                for (long row = 0; row < chunk.rowCount(); row++) {
                    int val = vector.getInt(row);
                    System.out.println(val);
                }
            }
        }
    }
}
```

Statement parameters remain 1-based, following the JDBC convention, while chunk columns and rows are 0-based, matching the C API and the user-defined function interfaces.

> The chunked result API currently supports basic data types only. Composite types such as `LIST` and `STRUCT` are not yet readable through this interface. The `query()` method is available on prepared statements only; there is no `query(String)` overload.

Further Reading

- Run Queries — sending the queries whose results this page reads, and the standard `ResultSet` accessors.
- Define Functions — the `DuckDBDataChunkReader` API shared with chunked results is also used to write user-defined functions.
- Define Connections — the `jdbc_stream_results` option that enables result streaming.
- C API — the `duckdb_fetch_chunk` function that the chunked result API exposes to Java.

Write User Defined Functions

Overview

The JDBC driver can register user-defined functions (UDFs) written in Java, including scalar functions and table functions. This page covers building and registering both, from the simple functional-interface overloads to the vectorized interface, along with type mapping and the function registry.

Scalar Functions

Build a scalar function with the `DuckDBFunctions.scalarFunction()` builder — which returns an `org.duckdb.DuckDBScalarFunctionBuilder` — and register it on a connection with `register()`. Once registered, the function can be called from SQL by the name passed to `withName()`. The builder is single-use: it is finalized once `register()` (or `close()`) is called.

The simplest functions map directly to a Java functional interface. Pick the overload that matches the number of arguments and, for primitives, their type:

- `withFunction(Supplier)`, `withFunction(Function)`, and `withFunction(BiFunction)` take zero, one, or two boxed (`Object`) arguments.
- `withIntFunction()`, `withLongFunction()`, and `withDoubleFunction()` take a unary or binary operator over `INTEGER`, `BIGINT`, or `DOUBLE` and avoid boxing.

```
import java.sql.Connection;
import java.sql.DriverManager;
import org.duckdb.DuckDBFunctions;

try (Connection conn = DriverManager.getConnection("jdbc:duckdb:")) {
    DuckDBFunctions.scalarFunction()
        .withName("java_add_one")
        .withParameter(int.class)
        .withReturnType(int.class)
        .withIntFunction(x -> x + 1)
        .register(conn);
}

SELECT java_add_one(41);
```

The binary overloads take two arguments of the same type, declared with `withParameters()`:

```
DuckDBFunctions.scalarFunction()
    .withName("java_weighted_sum")
    .withParameters(double.class, double.class)
    .withReturnType(double.class)
    .withDoubleFunction((x, w) -> x * w + 10.0)
    .register(conn);

SELECT java_weighted_sum(2.5, 4.0);
```

Variadic Arguments

For a variable number of arguments, call `withVarArgs()` to declare the element type and `withVarArgsFunction()` to supply the implementation. The callback receives the variadic arguments as an `Object[]`. `withVarArgs()` has to be set before `withVarArgsFunction()`, and the object-argument `withFunction()` overloads cannot be combined with varargs.

```
import java.sql.Connection;
import java.sql.DriverManager;
import org.duckdb.DuckDBColumnType;
import org.duckdb.DuckDBFunctions;
import org.duckdb.DuckDBLogicalType;

try (Connection conn = DriverManager.getConnection("jdbc:duckdb:");
     DuckDBLogicalType intType = DuckDBLogicalType.of(DuckDBColumnType.INTEGER)) {
    DuckDBFunctions.scalarFunction()
        .withName("java_sum_varargs")
        .withParameter(Integer.class) // fixed argument
        .withVarArgs(intType)         // variadic argument type
        .withReturnType(Integer.class)
        .withVarArgsFunction(args -> {
            int sum = 0;
            for (Object arg : args) {
                sum += (Integer) arg;
            }
            return sum;
        })
        .register(conn);
}

SELECT java_sum_varargs(1, 2, 3, 4);
```

NULL Handling

By default, a scalar function is called for every input row, including rows that contain SQL `NULL`:

- Callbacks that take `Object` arguments (`Function`, `BiFunction`, and the varargs form) receive `null` for a `NULL` input and may return `null` to produce a `NULL` result.
- The primitive callbacks (`withIntFunction()`, `withLongFunction()`, and `withDoubleFunction()`) are never handed a `null`. When DuckDB passes a `NULL`, the driver skips the Java call and writes `NULL` to the output automatically.

Call `withNullInNullOut()` to let the engine short-circuit *NULL in, NULL out*: it may skip the call and return `NULL` whenever an argument is `NULL`. The engine does not always take this shortcut, so an object callback should still check its arguments for `null`.

Volatile Functions

DuckDB may treat a scalar function as deterministic and cache or constant-fold its result for repeated inputs. Call `withVolatile()` for a function whose output can change between calls with the same arguments — for example one that reads external state — so the engine evaluates it for every row.

Error Handling

- Type and value errors raised while a callback reads its inputs or writes its result throw `DuckDBFunctions.FunctionException`, an unchecked `RuntimeException`.
- An out-of-range row or column index throws `IndexOutOfBoundsException`.
- Registration-time problems, such as an undefined return type or an invalid type declaration, throw `SQLException`.

Vectorized Scalar Functions

The examples above process one value per call. To process a whole batch of rows at once, register a `DuckDBScalarFunction` with `withVectorizedFunction()`. The callback receives the input as a `DuckDBDataChunkReader` (up to 2,048 rows) and writes results into a `DuckDBWritableVector`. Read each input column with `input.vector(columnIndex)`, iterate the row indices with `input.stream()`, and write each result at the same row index. This is the scalar-function counterpart to reading query results with chunked results.

```
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:");
     DuckDBLogicalType tsType = DuckDBLogicalType.of(DuckDBColumnType.TIMESTAMP);
     DuckDBLogicalType strType = DuckDBLogicalType.of(DuckDBColumnType.VARCHAR);
     DuckDBLogicalType dblType = DuckDBLogicalType.of(DuckDBColumnType.DOUBLE)) {
    DuckDBFunctions.scalarFunction()
        .withName("java_event_label")
        .withParameters(tsType, strType, dblType)
        .withReturnType(strType)
        .withVectorizedFunction((input, output) -> {
            input.stream().forEach(row -> {
                String value = input.vector(0).getLocalDateTime(row) + " | " +
                               String.valueOf(input.vector(1).getString(row)).trim().toUpperCase() + " | " +
                                input.vector(2).getDouble(row, 0.0d);
                output.setString(row, value);
            });
        })
        .register(conn);
}
```

```
SELECT java_event_label(TIMESTAMP '2026-04-04 12:00:00', 'launch', 4.5);
```

The `DuckDBDataChunkReader`, its `DuckDBReadableVector` columns, and the `DuckDBWritableVector` are valid only during callback execution and must not be retained.

Scalar Function Builder Methods

The `scalarFunction()` builder exposes the following methods:

- `withName(String)`
- `withParameter(Class<?> | DuckDBColumnType | DuckDBLogicalType)`
- `withParameters(Class<?>... | DuckDBColumnType... | DuckDBLogicalType...)`
- `withReturnType(Class<?> | DuckDBColumnType | DuckDBLogicalType)`
- `withFunction(Supplier | Function | BiFunction)`
- `withIntFunction(IntUnaryOperator | IntBinaryOperator)`
- `withLongFunction(LongUnaryOperator | LongBinaryOperator)`
- `withDoubleFunction(DoubleUnaryOperator | DoubleBinaryOperator)`
- `withVarArgs(DuckDBLogicalType)`
- `withVarArgsFunction(Function<Object[], ?>)`
- `withVectorizedFunction(DuckDBScalarFunction)`
- `withVolatile()`
- `withNullInNullOut()`
- `register(java.sql.Connection)`

Table Functions

Build a table function with the `DuckDBFunctions.tableFunction()` builder, which returns an `org.duckdb.DuckDBTableFunctionBuilder`. Like the scalar builder, it is single-use and finalized on `register()` (or `close()`). The implementation is a `DuckDBTableFunction` with the following callbacks:

- `bind` runs when the statement is prepared. It registers the output columns with `addResultColumn()` and reads positional and named call parameters with `getParameter()` and `getNamedParameter()`. It may return bind data that is passed to the later callbacks.
- `init` runs once before execution and returns the global state.
- `localInit` (optional) runs once per execution thread and returns thread-local state. Override it when a parallel scan needs per-thread state; the default returns `null`.
- `apply` writes rows into the output vectors and returns the number of rows written, with access to the bind, global-init, and local-init data. The engine calls it repeatedly until it returns `0`.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.concurrent.atomic.AtomicBoolean;
import org.duckdb.DuckDBFunctions;
import org.duckdb.DuckDBTableFunction;
import org.duckdb.DuckDBTableFunctionBindInfo;
import org.duckdb.DuckDBTableFunctionInitInfo;
import org.duckdb.DuckDBTableFunctionCallInfo;
import org.duckdb.DuckDBDataChunkWriter;
import org.duckdb.DuckDBValue;

try (Connection conn = DriverManager.getConnection("jdbc:duckdb:")) {
    DuckDBFunctions.tableFunction()
        .withName("java_table_basic")
        .withParameter(int.class)
        .withNamedParameter("param1", String.class)
        .withFunction(new DuckDBTableFunction<Integer, AtomicBoolean, Object>() {
            @Override
            public Integer bind(DuckDBTableFunctionBindInfo info) throws Exception {
                info.addResultColumn("col1", Integer.TYPE)
                    .addResultColumn("col2", String.class);
                DuckDBValue param = info.getParameter(0);
                return param.getInt();
            }

            @Override
            public AtomicBoolean init(DuckDBTableFunctionInitInfo info) throws Exception {
                return new AtomicBoolean(false);
            }

            @Override
            public long apply(DuckDBTableFunctionCallInfo info, DuckDBDataChunkWriter output) throws
            Exception {
                Integer bindData = info.getBindData();
                AtomicBoolean done = info.getInitData();
                if (done.get()) {
                    return 0;
                }
                output.vector(0).setInt(0, bindData);
                output.vector(1).setString(0, "foo");
                output.vector(0).setNull(1);
                output.vector(1).setString(1, "bar");
                done.set(true);
                return 2;
            }
        })
        .register(conn);
}
```

The generic type parameters of `DuckDBTableFunction` are, in order, the bind data, the global init data, and the local init data. Call the function from SQL:

```
FROM java_table_basic(42, param1 = 'foobar');
```

Rows are written through a `DuckDBDataChunkWriter`, the write-side counterpart to the `DuckDBDataChunkReader` used for read-

ing results. Each output vector is addressed by column index, and values are set by row index with methods such as `setInt()`, `setString()`, and `setNull()`.

All callback arguments, including the parameter and info objects, are valid only during callback execution and must not be retained. The Java table function interface mirrors DuckDB's C API table functions as closely as possible.

Producing Rows in Chunks

The example above writes every row in a single `apply` call, but a table function is not expected to. DuckDB calls `apply` repeatedly, each time passing a fresh output chunk that holds up to `output.capacity()` rows (2,048 by default). Fill up to that many rows, return how many you wrote, and DuckDB calls `apply` again for the next chunk until you return `0`. A cursor into the source therefore has to live in the init (or local-init) state so it survives across calls, rather than in a local variable.

The `apply` below streams a bounded integer series in chunks. It also dispatches on each column's declared type with `DuckDBWritableVector.getType()`, which is convenient when the same routine populates columns of different types:

```
// Held in the init state so the position survives across apply() calls.
final class Series {
    long next;
    final long end;
    Series(long end) { this.end = end; }
}

DuckDBFunctions.tableFunction()
    .withName("java_series")
    .withParameter(long.class)
    .withFunction(new DuckDBTableFunction<Long, Series, Object>() {
        @Override
        public Long bind(DuckDBTableFunctionBindInfo info) throws Exception {
            info.addResultColumn("n", Long.TYPE)
                .addResultColumn("label", String.class);
            return info.getParameter(0).getLong();
        }

        @Override
        public Series init(DuckDBTableFunctionInitInfo info) throws Exception {
            return new Series(info.getBindData());
        }

        @Override
        public long apply(DuckDBTableFunctionCallInfo info, DuckDBDataChunkWriter output) throws Exception {
            Series cursor = info.getInitData();
            long row = 0;
            // Fill at most one chunk; DuckDB calls apply() again for the rest.
            for (; row < output.capacity() && cursor.next < cursor.end; row++, cursor.next++) {
                for (long col = 0; col < output.columnCount(); col++) {
                    DuckDBWritableVector vector = output.vector(col);
                    switch (vector.getType()) {
                        case BIGINT:  vector.setLong(row, cursor.next); break;
                        case VARCHAR: vector.setString(row, "row-" + cursor.next); break;
                        default:      vector.setNull(row);
                    }
                }
            }
            return row; // returning 0 tells DuckDB the scan is finished
        }
    })
    .register(conn);
```

Calling `FROM java_series(5000)` drives `apply` three times — two full 2,048-row chunks and a final partial one — before the fourth call returns `0`.

Cleaning up Resources

A table function may hold open resources such as files, sockets, or iterators that have to be released when the scan finishes. A state that owns such a resource can implement `DuckDBTableFunctionState` to receive a deterministic `close()` call once DuckDB no longer needs it:

```
final class SourceState implements DuckDBTableFunctionState {
    private final AutoCloseable resource;

    SourceState(AutoCloseable resource) {
        this.resource = resource;
    }

    @Override
    public void close() throws Exception {
        resource.close();
    }
}
```

DuckDB calls `close()` after the function is exhausted, after a `LIMIT`, or on an error, cancellation, or early JDBC close. Cleanup may run on native worker threads with no ordering guarantees, so `close()` should be fast, idempotent, and thread-safe. It must not run SQL or reuse the same connection, and any exception it throws is not propagated to the query. A state that only implements `AutoCloseable`, or a `null` state, is left untouched.

Parallel Execution

To run a table function on multiple threads, set the thread count with `info.setMaxThreads()` in the `init` callback. Parallel output may also require the `preserve_insertion_order` option set to `false`, in the connection string or the connection properties.

Projection Pushdown

By default a table function is asked to produce every column it declared in `bind`, even when the query selects only a few of them. Call `withProjectionPushdown()` on the builder to let DuckDB tell the function which columns are actually needed, so that `apply` can skip the work of filling the rest.

When pushdown is enabled, the `init` callback learns the projected columns from its `DuckDBTableFunctionInitInfo`: `getColumnCount()` returns the number of columns the query requires, and `getColumnIndex(position)` maps each projected position back to the column index used in `bind` and in the output vectors. Store this mapping in the init state and consult it in `apply`.

```
.withProjectionPushdown()
.withFunction(new DuckDBTableFunction<Void, long[], Object>() {
    // ... bind() registers the full set of output columns ...

    @Override
    public long[] init(DuckDBTableFunctionInitInfo info) throws Exception {
        // Record which declared columns the query actually needs.
        long[] projected = new long[(int) info.getColumnCount()];
        for (int i = 0; i < projected.length; i++) {
            projected[i] = info.getColumnIndex(i);
        }
        return projected;
    }
    // ... apply() writes only the columns listed in the projected mapping ...
})
```

Without `withProjectionPushdown()`, `getColumnCount()` reports every declared column and the function must populate all of them.

Inspecting Registered Functions

Every successful `register()` call returns a `DuckDBFunctions.RegisteredFunction` and records the registration in a process-wide, Java-side registry. This registry is bookkeeping for functions registered through the JDBC API; it is not an authoritative view of the DuckDB catalog.

```
List<DuckDBFunctions.RegisteredFunction> functions = DuckDBDriver.registeredFunctions();
```

`registeredFunctions()` returns a read-only snapshot, and each `RegisteredFunction` exposes `name()` and `functionKind()` (`SCALAR` or `TABLE`). Duplicate names may appear if the same name is registered more than once. `DuckDBDriver.clearFunctionsRegistry()` clears the Java-side registry only; it does not de-register the functions from DuckDB.

Type Mapping

`withParameter()`, `withParameters()`, and `withReturnType()` accept three forms of type declaration:

- a Java `Class<?>`, including primitive class literals such as `int.class` or `Integer.TYPE`
- a `DuckDBColumnType` enum value
- a `DuckDBLogicalType` instance

When a type is declared with a Java `Class`, the driver maps it to a DuckDB type as follows:

Java type	DuckDB type
`int`	`INTEGER`
`long`	`BIGINT`
`float`	`FLOAT`
`double`	`DOUBLE`
`String`	`VARCHAR`
`BigDecimal`	`DECIMAL`
`BigInteger`	`HUGEINT`
`LocalDate`, `java.sql.Date`	`DATE`
`LocalDateTime`, `java.sql.Timestamp`, `java.util.Date`	`TIMESTAMP`

For types without a direct Java mapping, declare the type explicitly with `DuckDBColumnType` or `DuckDBLogicalType`. For example, `UHUGEINT` requires an explicit `DuckDBColumnType.UHUGEINT` declaration, and `DuckDBLogicalType.decimal(width, scale)` sets an explicit `DECIMAL` precision and scale.

Composite types such as `LIST` and `STRUCT` are not currently supported in UDFs. Support for them is planned for a future release.

Further Reading

- Handle Results — the `DuckDBDataChunkReader` API used to read a vectorized function's inputs is shared with chunked results.
- C API — the table function interface that the Java table function API mirrors.
- Run Queries — calling registered functions from SQL.
- Configuration — the `preserve_insertion_order` option that parallel table functions may require.

Profile and Monitor

Overview

The JDBC driver exposes several ways to observe a running or completed query: cancelling and timing out queries, polling query progress, retrieving profiling output, and emitting memory-usage events through Java Flight Recorder. This page covers each of them.

Query Progress and Cancellation

A running query can be cancelled from another thread with the standard JDBC `Statement.cancel()` method, which interrupts the query executing on the statement's connection. Setting `Statement.setQueryTimeout(seconds)` arms an automatic cancellation after the given number of seconds.

For long-running queries, `DuckDBPreparedStatement.getQueryProgress()` returns a `QueryProgress` snapshot while the query executes. It is intended to be called from a separate thread and exposes `getPercentage()`, `getRowsProcessed()`, and `getTotalRowsToProcess()`.

```
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.concurrent.CompletableFuture;
import org.duckdb.DuckDBConnection;
import org.duckdb.DuckDBPreparedStatement;
import org.duckdb.QueryProgress;

try (DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
     DuckDBPreparedStatement ps =
         (DuckDBPreparedStatement) conn.prepareStatement("SELECT count(*) FROM range(1_000_000_000)")) {
    // Run the query on a background thread so its progress can be polled while it executes.
    CompletableFuture<Void> query = CompletableFuture.runAsync(() -> {
        try {
            ps.executeQuery().close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    });

    while (!query.isDone()) {
        QueryProgress progress = ps.getQueryProgress();
        System.out.printf("%.1f%% (%d / %d rows)%n", progress.getPercentage(),
            progress.getRowsProcessed(), progress.getTotalRowsToProcess());
        Thread.sleep(100);
    }
    query.join();
}
```

Profiling

Cast a connection to `DuckDBConnection` and call `getProfilingInformation(ProfilerPrintFormat)` to retrieve profiling output for the most recent query on that connection, after enabling profiling with `SET enable_profiling`. The `ProfilerPrintFormat` enum selects the output format: `DEFAULT`, `TEXT`, `QUERY_TREE`, `QUERY_TREE_OPTIMIZER`, `NO_OUTPUT`, `JSON`, `HTML`, `GRAPHVIZ`, `YAML`, and `MERMAID`.

```
try (DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
     Statement stmt = conn.createStatement()) {
    stmt.execute("SET enable_profiling = 'json'");
    stmt.executeQuery("SELECT count(*) FROM range(1000)").close();
    String profile = conn.getProfilingInformation(ProfilerPrintFormat.JSON);
    System.out.println(profile);
}
```

Memory Monitoring with JFR

On a JFR-capable JVM, the driver can emit Java Flight Recorder events describing DuckDB's internal memory usage, so that any JFR-aware tool — JDK Mission Control, the `jfr` command line tool, continuous profilers — can ingest DuckDB memory metrics alongside the rest of the JVM signal. The feature is strictly opt-in and is a silent no-op on JVMs without JFR support: nothing is emitted unless the application both sets the JDBC property on a connection and has an active JFR recording that enables the event.

Enabling Emission

Monitoring is opt-in per connection: set the `jdbc_jfr_memory_monitor` property to an identifier for the connection's database instance. The value is an arbitrary label that is attached to every event as its `component` field, so that operators can attribute memory to logical components. An absent or empty value leaves monitoring disabled.

```
Properties props = new Properties();
props.setProperty("jdbc_jfr_memory_monitor", "analytics-db");
try (Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", props)) {
    // work with the connection while a JFR recording is active
}
```

The same option can be set in the URL:

```
jdbc:duckdb:/tmp/my_database;jdbc_jfr_memory_monitor=analytics-db
```

Property value	Effect
absent	No events emitted for this connection.
empty string	No events emitted, the same as absent.
non-empty string	Events emitted, tagged with the given value.

The JDBC property is only an enable and label switch. It does not control whether JFR is recording or the sampling period; those are governed by the JFR recording settings.

Controlling the Period and Enabled State

The sampling rate and enabled state are JFR-native settings. Configure them in a `.jfc` profile, through JMC, or programmatically with the `jdk.jfr.Recording` API:

```
try (Recording r = new Recording()) {
    r.enable("duckdb.MemoryUsage").withPeriod(Duration.ofSeconds(1));
    r.start();
    // ... application work ...
    r.stop();
    r.dump(Path.of("app.jfr"));
}
```

The equivalent `.jfc` snippet:

```
<event name="duckdb.MemoryUsage">
  <setting name="enabled">true</setting>
  <setting name="period">1 s</setting>
</event>
```

When no recording enables the event, the driver does no work: the underlying `duckdb_memory()` query is never issued.

Event Schema

While a recording is active with the event enabled, the driver emits one `duckdb.MemoryUsage` event per DuckDB memory tag per JFR tick, with the following fields:

Field	Type	Description
`component`	`String`	Application-supplied identifier, the `jdbc_jfr_memory_monitor` value.
`tag`	`String`	DuckDB memory tag, such as `BASE_TABLE`, `HASH_TABLE`, or `ALLOCATOR`.
`dbAddress`	`long`	Native address of the DuckDB instance, a stable per-instance identifier.
`memoryUsageBytes`	`long`	Bytes currently allocated for this tag.
`temporaryStorageBytes`	`long`	Bytes spilled to temporary storage for this tag.

The standard JFR `startTime`, `duration`, and `eventThread` fields are also present. Stack traces are disabled for this event.

Attribution

The monitor is keyed on the native DuckDB instance address, the `dbAddress` field, not on the JDBC connection. There is one sample stream per distinct instance, so shared memory is not double-counted. The `component` label is captured from the first opted-in connection to an instance, and later opted-in connections to the same instance do not change it. The monitor is created when the first opted-in connection opens and torn down when the last one closes; opening a new opted-in connection afterward starts a fresh monitor.

Whether two `getConnection` calls share a `dbAddress` follows the same rules as instance caching:

URL or operation	Same `dbAddress`?
`jdbc:duckdb:` (unnamed in-memory)	No — a fresh instance per call.
`jdbc:duckdb:memory:<label>`	Yes, when `<label>` matches.
`jdbc:duckdb:<path>`	Yes, when the path matches.
`conn.duplicate()`	Yes, always.

Because connections that share a database instance also share a `dbAddress`, and therefore a single `component` label, per-component attribution requires each component to open a connection that resolves to a distinct instance. An unnamed in-memory `jdbc:duckdb:` URL creates a connection-private instance, which makes it the simplest choice; give each such connection a distinct `jdbc_jfr_memory_monitor` value.

Requirements

The feature needs a JFR-capable JVM:

- OpenJDK and HotSpot 11 and later include JFR.
- Amazon Corretto 8, OpenJDK 8u272 and later, and several other Java 8 distributions include the JFR backport (the `jdk.jfr` package).
- On a JVM without `jdk.jfr`, such as some stripped Java 8 builds, the feature is a silent no-op: the `jdbc_jfr_memory_monitor` property is ignored and no classes that depend on `jdk.jfr` are loaded.

No additional JVM flags are required.

Inspecting a Recording

Inspect a captured recording with JDK Mission Control or the `jfr` command line tool bundled with the JDK:

```
jfr summary app.jfr | grep duckdb.MemoryUsage
jfr print --events duckdb.MemoryUsage app.jfr
jfr metadata app.jfr | sed -n '/class MemoryUsage/,/^}/p'
```

Further Reading

- Define Connections — the `jdbc_jfr_memory_monitor` property and the query-cancel scheduler thread the driver runs.
- Profiling — DuckDB's query profiling output and the formats `getProfilingInformation()` returns.
- Run Queries — running the queries whose progress, timing, and profiles this page observes.

Deploy as Native Image

Overview

The JDBC driver runs inside GraalVM Native Image executables. Graal Native Image compiles a Java application ahead of time into a standalone binary that starts in milliseconds and requires no JVM on the target machine, which suits command line tools, serverless functions, and small containers.

Recent driver versions ship the JNI reachability metadata that `native-image` needs, and the build picks it up automatically from the class path.

Two things remain for the application to configure: native access and the location of DuckDB's shared library.

> Driver versions up to and including 1.5.5 do not ship this metadata. For those versions, generate it by running the application once on the JVM with the tracing agent:
>
> ```
> java -agentlib:native-image-agent=config-output-dir=<config_dir> -cp <classpath> <MainClass>
> ```
>
> Next, pass the directory to the build with `-H:ConfigurationFileDirectories=<config_dir>`.

Requirements

GraalVM for JDK 22 or later is required. Older GraalVM releases initialize the driver's classes at image build time, which bakes a build machine path into the executable and fails at run time with `UnsatisfiedLinkError`.

Pass `--enable-native-access=ALL-UNNAMED` to `native-image`, or add it as a build argument in the Native Build Tools Maven or Gradle plugin.

Load the driver explicitly before opening the first connection:

```
Class.forName("org.duckdb.DuckDBDriver");
```

Driver auto-registration through `ServiceLoader` is not always visible to Graal Native Image's closed world analysis, and the explicit load makes registration deterministic.

Providing the Shared Library

DuckDB's engine is a native library bundled inside the driver JAR, one file per platform. Choose one of two ways to make it available to the executable.

Option 1: Embed the Library in the Executable

Add a resource entry naming your platform's library, either in a configuration directory passed via `-H:ConfigurationFileDirectories` or under `META-INF/native-image` in your own project:

```
{ "resources": { "includes": [ { "pattern": "libduckdb_java\\.so_linux_amd64" } ] } }
```

The bundled library names are `libduckdb_java.so_linux_amd64`, `libduckdb_java.so_linux_arm64`, `libduckdb_java.so_osx_universal`, and `libduckdb_java.so_windows_amd64`.

Name the one for your target platform explicitly. A wildcard matching all of them adds roughly 260 MB to the executable.

This option produces a single self contained file, around 120 MB. The driver extracts the library to a temporary directory when the first connection opens, which adds up to a second of startup time and requires a writable temporary directory.

Option 2: Ship the Library Next to the Executable

Build without any resource entry and place the shared library in the same directory as the executable. The file works under its bundled name, for example `libduckdb_java.so_linux_amd64`, or under the platform convention: `libduckdb_java.so` on Linux, `libduckdb_java.dylib` on macOS, `duckdb_java.dll` on Windows. The lookup is anchored to the executable rather than the working directory, so the program runs correctly from anywhere.

This option produces a small executable, under 20 MB for a simple application, with no extraction cost when connections open. The library can be extracted from the driver JAR, or taken from the `-nolib` distribution together with its separate library artifact.

Building

A minimal build, with the driver JAR in the current directory:

```
javac -cp duckdb_jdbc-<version>.jar App.java
native-image --no-fallback --enable-native-access=ALL-UNNAMED \
    -cp duckdb_jdbc-<version>.jar:. -o app App
```

`--no-fallback` makes the build fail outright instead of producing an image that still requires a JVM. For Maven and Gradle projects, the Native Build Tools plugins wrap the same build and run it during `mvn package` or `gradle nativeCompile`.

Further Reading

- Troubleshoot — the Native Image errors that indicate missing metadata or a missing shared library.
- Define Connections — driver registration and connection configuration.
- Java (JDBC) Client — installing the driver from Maven Central.

Troubleshoot

Overview

This page collects common issues encountered when using the DuckDB JDBC driver, together with their workarounds. If you run into a problem that is not covered here, search the driver's issue tracker on GitHub.

Driver Class Not Found

This error occurs when the DuckDB JDBC driver is not on the application's classpath, typically because the build tool has not resolved the dependency. If the Java application is unable to find the DuckDB driver, it may throw the following error:

```
Exception in thread "main" java.sql.SQLException: No suitable driver found for jdbc:duckdb:
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:706)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:252)
    ...
```

And when trying to load the class manually, it may result in this error:

```
Exception in thread "main" java.lang.ClassNotFoundException: org.duckdb.DuckDBDriver
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:641)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:188)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:520)
    at java.base/java.lang.Class.forName0(Native Method)
    at java.base/java.lang.Class.forName(Class.java:375)
    ...
```

These errors stem from the DuckDB Maven/Gradle dependency not being detected. To ensure that it is detected, force refresh the Maven configuration in your IDE.

Parquet String Column Returns a Blob

Parquet files written by some legacy writers do not set the `UTF8` flag on string columns, so DuckDB reads them as `BLOB`. `ResultSet.getObject()` then returns a `DuckDBBlobResult` and `ResultSet.getString()` returns the bytes rendered as an escaped string rather than the expected text. Enable the `binary_as_string` setting to read these columns as VARCHAR:

```
try (Statement stmt = conn.createStatement()) {
    stmt.execute("SET binary_as_string = true;");
}
```

The same option can be passed to `read_parquet` directly, for example `read_parquet('file.parquet', binary_as_string = true)`.

This behavior is tracked in duckdb-java issue #113.

Native Image: `NoSuchMethodError` When Opening a Connection

In a GraalVM Native Image executable, opening the first connection may fail with:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
    ...
Caused by: java.lang.NoSuchMethodError: <class and method>
    at com.oracle.svm.core.jni.functions.JNIFunctions$Support.getMethodID(JNIFunctions.java)
    ...
```

The driver resolves its entire JNI surface eagerly when its native library initializes, so a single class, method, or field missing from the reachability metadata fails the whole initialization. This indicates that the metadata compiled into the image is missing or older than the driver: upgrade to a driver version that ships its own metadata, or regenerate the metadata with the tracing agent against the exact driver version in use.

The error is sometimes wrapped in a misleading message:

```
java.lang.UnsatisfiedLinkError: Unsupported JNI version 0xffffffff, required by <path>/libduckdb_
java.<suffix>
```

This is what the library's `JNI_OnLoad` reports when an internal lookup failed, and the cause is the same missing metadata, not a JNI version problem.

Native Image: `UnsatisfiedLinkError: Can't load library`

In a Native Image executable, the first connection may fail with:

```
java.lang.UnsatisfiedLinkError: Can't load library: duckdb_java | java.library.path = [.]
    ...
Caused by: java.io.FileNotFoundException: DuckDB JNI library not found, path: '<path>/libduckdb_
java.<suffix>'
```

The shared library was neither embedded in the executable nor found next to it. Either add a resource entry for your platform's library or place the library file beside the executable. Both options are described on the Deploy as Native Image page.

Warning about a Restricted Method in `java.lang.System`

On JDK 24 and later, loading the driver prints:

```
WARNING: A restricted method in java.lang.System has been called
WARNING: java.lang.System::load has been called by org.duckdb.DuckDBNative ...
WARNING: Use --enable-native-access=ALL-UNNAMED to avoid a warning for callers in this module
```

This is the JDK's native access integrity check and is harmless. Silence it by running the JVM with `--enable-native-access=ALL-UNNAMED`, or the module name of the driver if you place it on the module path. Future JDK releases will turn this warning into an error, so adding the flag is recommended.

Further Reading

- Deploy as Native Image — building standalone executables with GraalVM, including both ways to provide the shared library.
- Java (JDBC) Client — installing the driver from Maven Central, the fix for the driver-not-found errors above.
- Define Connections — driver registration, configuration options, and instance behavior behind many connection-time errors.
- Parquet Files — the `binary_as_string` setting and other options for reading Parquet string columns correctly.

Node.js (Neo)

Node.js Client (Neo)

> Installation To use the DuckDB Node.js client, visit the Node.js installation page.
>
> The latest stable version of the DuckDB Node.js (Neo) client is {{ latest_version }}.

An API for using DuckDB in Node.js.

The primary package, @duckdb/node-api, is a high-level API meant for applications. It depends on low-level bindings that adhere closely to DuckDB's C API, available separately as @duckdb/node-bindings.

Roadmap

Some features are not yet complete:

- Binding and appending the MAP and UNION data types
- Appending default values row-by-row
- User-defined types & functions
- Profiling info
- Table description
- APIs for Arrow

See the issues list on GitHub for the most up-to-date roadmap.

Platforms

The Node.js (Neo) client supports the following platforms:

- `linux_amd64`
- `linux_amd64_musl`
- `linux_arm64`
- `linux_arm64_musl`
- `osx_amd64`
- `osx_arm64`
- `windows_amd64`
- `windows_arm64`

Examples

Get Basic Information

```
import duckdb from '@duckdb/node-api';

console.log(duckdb.version());

console.log(duckdb.configurationOptionDescriptions());
```

Connect

```
import { DuckDBConnection } from '@duckdb/node-api';

const connection = await DuckDBConnection.create();
```

This uses the default instance. For advanced usage, you can create instances explicitly.

Create Instance

```
import { DuckDBInstance } from '@duckdb/node-api';
```

Create with an in-memory database:

```
const instance = await DuckDBInstance.create(':memory:');
```

Equivalent to the above:

```
const instance = await DuckDBInstance.create();
```

Read from and write to a database file, which is created if needed:

```
const instance = await DuckDBInstance.create('my_duckdb.db');
```

Set configuration options:

```
const instance = await DuckDBInstance.create('my_duckdb.db', {
  threads: '4'
});
```

Instance Cache

Multiple instances in the same process should not attach the same database.

To prevent this, an instance cache can be used:

```
const instance = await DuckDBInstance.fromCache('my_duckdb.db');
```

This uses the default instance cache. For advanced usage, you can create instance caches explicitly:

```
import { DuckDBInstanceCache } from '@duckdb/node-api';

const cache = new DuckDBInstanceCache();
const instance = await cache.getOrCreateInstance('my_duckdb.db');
```

Connect to Instance

```
const connection = await instance.connect();
```

Disconnect

Connections will be disconnected automatically soon after their reference is dropped, but you can also disconnect explicitly if and when you want:

```
connection.disconnectSync();
```

or, equivalently:

```
connection.closeSync();
```

Run SQL

```
const result = await connection.run('from test_all_types()');
```

The SQL string may contain multiple statements separated by semicolons. In that case, all of the statements are executed in order, and the returned result is the result of the *last* statement:

```
const reader = await connection.runAndReadAll(
  `create table t (id integer);
   insert into t values (1);
   from t;`
);
reader.getRows();
// [ [1] ]
```

To run multiple statements and access each statement's result individually, use `extractStatements` instead.

Parameterize SQL

```
const prepared = await connection.prepare('select $1, $2, $3');
prepared.bindVarchar(1, 'duck');
prepared.bindInteger(2, 42);
prepared.bindList(3, listValue([10, 11, 12]), LIST(INTEGER));
const result = await prepared.run();
```

or:

```
const prepared = await connection.prepare('select $a, $b, $c');
prepared.bind({
  'a': 'duck',
  'b': 42,
  'c': listValue([10, 11, 12]),
}, {
  'a': VARCHAR,
  'b': INTEGER,
  'c': LIST(INTEGER),
});
const result = await prepared.run();
```

or even:

```
const result = await connection.run('select $a, $b, $c', {
  'a': 'duck',
  'b': 42,
  'c': listValue([10, 11, 12]),
}, {
  'a': VARCHAR,
  'b': INTEGER,
  'c': LIST(INTEGER),
});
```

Unspecified types will be inferred:

```
const result = await connection.run('select $a, $b, $c', {
  'a': 'duck',
  'b': 42,
  'c': listValue([10, 11, 12]),
});
```

Specifying Values

Values of many data types are represented using one of the JS primitives `boolean`, `number`, `bigint`, or `string`. Also, any type can have `null` values.

Values of some data types need to be constructed using special functions. These are:

Type	Function
ARRAY	arrayValue
BIT	bitValue
BLOB	blobValue
DATE	dateValue
DECIMAL	decimalValue
INTERVAL	intervalValue
LIST	listValue
MAP	mapValue
STRUCT	structValue
TIME	timeValue
TIMETZ	timeTZValue
TIMESTAMP	timestampValue
TIMESTAMPTZ	timestampTZValue
TIMESTAMP_S	timestampSecondsValue
TIMESTAMP_MS	timestampMillisValue
TIMESTAMP_NS	timestampNanosValue
UNION	unionValue
UUID	uuidValue

Stream Results

Streaming results evaluate lazily when rows are read.

```
const result = await connection.stream('from range(10_000)');
```

Inspect Result Metadata

Get column names and types:

```
const columnNames = result.columnNames();
const columnTypes = result.columnTypes();
```

Read Result Data

Run and read all data:

```
const reader = await connection.runAndReadAll('from test_all_types()');
const rows = reader.getRows();
// OR: const columns = reader.getColumns();
```

Stream and read up to (at least) some number of rows:

```
const reader = await connection.streamAndReadUntil(
  'from range(5000)',
  1000
);
```

```
const rows = reader.getRows();
// rows.length === 2048. (Rows are read in chunks of 2048.)
```

Read rows incrementally:

```
const reader = await connection.streamAndRead('from range(5000)');
reader.readUntil(2000);
// reader.currentRowCount === 2048 (Rows are read in chunks of 2048.)
// reader.done === false
reader.readUntil(4000);
// reader.currentRowCount === 4096
// reader.done === false
reader.readUntil(6000);
// reader.currentRowCount === 5000
// reader.done === true
```

Get Result Data

Result data can be retrieved in a variety of forms:

```
const reader = await connection.runAndReadAll(
  'from range(3) select range::int as i, 10 + i as n'
);

const rows = reader.getRows();
// [ [0, 10], [1, 11], [2, 12] ]

const rowObjects = reader.getRowObjects();
// [ { i: 0, n: 10 }, { i: 1, n: 11 }, { i: 2, n: 12 } ]

const columns = reader.getColumns();
// [ [0, 1, 2], [10, 11, 12] ]

const columnsObject = reader.getColumnsObject();
// { i: [0, 1, 2], n: [10, 11, 12] }
```

Result Value Types

By default, each result value is represented in JavaScript as shown below. A DuckDB `NULL` is always represented as JavaScript `null`, regardless of the column's type.

Types represented as a JavaScript primitive:

DuckDB type	JavaScript representation
`BOOLEAN`	`boolean`
`TINYINT`, `SMALLINT`, `INTEGER`, `UTINYINT`, `USMALLINT`, `UINTEGER`	`number`
`FLOAT`, `DOUBLE`	`number`
`BIGINT`, `HUGEINT`, `UBIGINT`, `UHUGEINT`	`bigint`
`VARCHAR`, `ENUM`	`string`

Types represented as a specialized object (see Inspect Data Values for their properties):

DuckDB type	JavaScript representation
ARRAY	DuckDBArrayValue
BIT	DuckDBBitValue
BLOB	DuckDBBlobValue
DATE	DuckDBDateValue
DECIMAL	DuckDBDecimalValue
INTERVAL	DuckDBIntervalValue
LIST	DuckDBListValue
MAP	DuckDBMapValue
STRUCT	DuckDBStructValue
TIME	DuckDBTimeValue
TIME WITH TIME ZONE	DuckDBTimeTZValue
TIMESTAMP	DuckDBTimestampValue
TIMESTAMP WITH TIME ZONE	DuckDBTimestampTZValue
TIMESTAMP_S	DuckDBTimestampSecondsValue
TIMESTAMP_MS	DuckDBTimestampMillisValue
TIMESTAMP_NS	DuckDBTimestampNanosValue
UNION	DuckDBUnionValue
UUID	DuckDBUUIDValue

To retrieve values as JavaScript built-ins or as values that can be losslessly serialized to JSON instead of these specialized objects, use the `JS` or `Json` result methods described in Convert Result Data.

Convert Result Data

By default, data values that cannot be represented as JS built-ins are returned as specialized JS objects; see `Inspect Data Values` below.

To retrieve data in a different form, such as JS built-ins or values that can be losslessly serialized to JSON, use the `JS` or `Json` forms of the above result data methods.

Custom converters can be supplied as well. See the implementations of JSDuckDBValueConverter and JsonDuckDBValueConverters for how to do this.

Examples (using the `Json` forms):

```
const reader = await connection.runAndReadAll(
  'from test_all_types() select bigint, date, interval limit 2'
);

const rows = reader.getRowsJson();
// [
//   [
//     "-9223372036854775808",
//     "5877642-06-25 (BC)",
//     { "months": 0, "days": 0, "micros": "0" }
//   ],
//   [
//     "9223372036854775807",
//     "5881580-07-10",
```

```
//     { "months": 999, "days": 999, "micros": "999999999" }
//   ]
// ]

const rowObjects = reader.getRowObjectsJson();
// [
//   {
//     "bigint": "-9223372036854775808",
//     "date": "5877642-06-25 (BC)",
//     "interval": { "months": 0, "days": 0, "micros": "0" }
//   },
//   {
//     "bigint": "9223372036854775807",
//     "date": "5881580-07-10",
//     "interval": { "months": 999, "days": 999, "micros": "999999999" }
//   }
// ]

const columns = reader.getColumnsJson();
// [
//   [ "-9223372036854775808", "9223372036854775807" ],
//   [ "5877642-06-25 (BC)", "5881580-07-10" ],
//   [
//     { "months": 0, "days": 0, "micros": "0" },
//     { "months": 999, "days": 999, "micros": "999999999" }
//   ]
// ]

const columnsObject = reader.getColumnsObjectJson();
// {
//   "bigint": [ "-9223372036854775808", "9223372036854775807" ],
//   "date": [ "5877642-06-25 (BC)", "5881580-07-10" ],
//   "interval": [
//     { "months": 0, "days": 0, "micros": "0" },
//     { "months": 999, "days": 999, "micros": "999999999" }
//   ]
// }
```

These methods handle nested types as well:

```
const reader = await connection.runAndReadAll(
  'from test_all_types() select int_array, struct, map, "union" limit 2'
);

const rows = reader.getRowsJson();
// [
//   [
//     [],
//     { "a": null, "b": null },
//     [],
//     { "tag": "name", "value": "Frank" }
//   ],
//   [
//     [ 42, 999, null, null, -42],
//     { "a": 42, "b": "🦆🦆🦆🦆🦆" },
//     [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose" }
//     ],
//     { "tag": "age", "value": 5 }
//   ]
// ]

const rowObjects = reader.getRowObjectsJson();
// [
//   {
```

```
//     "int_array": [],
//     "struct": { "a": null, "b": null },
//     "map": [],
//     "union": { "tag": "name", "value": "Frank" }
//   },
//   {
//     "int_array": [ 42, 999, null, null, -42 ],
//     "struct": { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" },
//     "map": [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose" }
//     ],
//     "union": { "tag": "age", "value": 5 }
//   }
// ]

const columns = reader.getColumnsJson();
// [
//   [
//     [],
//     [42, 999, null, null, -42]
//   ],
//   [
//     { "a": null, "b": null },
//     { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" }
//   ],
//   [
//     [],
//     [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose"}
//     ]
//   ],
//   [
//     { "tag": "name", "value": "Frank" },
//     { "tag": "age", "value": 5 }
//   ]
// ]

const columnsObject = reader.getColumnsObjectJson();
// {
//   "int_array": [
//     [],
//     [42, 999, null, null, -42]
//   ],
//   "struct": [
//     { "a": null, "b": null },
//     { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" }
//   ],
//   "map": [
//     [],
//     [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose" }
//     ]
//   ],
//   "union": [
//     { "tag": "name", "value": "Frank" },
//     { "tag": "age", "value": 5 }
//   ]
// }
```

Column names and types can also be serialized to JSON:

```
const columnNamesAndTypes = reader.columnNamesAndTypesJson();
// {
```

```
//   "columnNames": [
//     "int_array",
//     "struct",
//     "map",
//     "union"
//   ],
//   "columnTypes": [
//     {
//       "typeId": 24,
//       "valueType": {
//         "typeId": 4
//       }
//     },
//     {
//       "typeId": 25,
//       "entryNames": [
//         "a",
//         "b"
//       ],
//       "entryTypes": [
//         {
//           "typeId": 4
//         },
//         {
//           "typeId": 17
//         }
//       ]
//     },
//     {
//       "typeId": 26,
//       "keyType": {
//         "typeId": 17
//       },
//       "valueType": {
//         "typeId": 17
//       }
//     },
//     {
//       "typeId": 28,
//       "memberTags": [
//         "name",
//         "age"
//       ],
//       "memberTypes": [
//         {
//           "typeId": 17
//         },
//         {
//           "typeId": 3
//         }
//       ]
//     }
//   ]
// }

const columnNameAndTypeObjects = reader.columnNameAndTypeObjectsJson();
// [
//   {
//     "columnName": "int_array",
//     "columnType": {
//       "typeId": 24,
//       "valueType": {
//         "typeId": 4
//       }
//     }
```

```
//   },
//   {
//     "columnName": "struct",
//     "columnType": {
//       "typeId": 25,
//       "entryNames": [
//         "a",
//         "b"
//       ],
//       "entryTypes": [
//         {
//           "typeId": 4
//         },
//         {
//           "typeId": 17
//         }
//       ]
//     }
//   },
//   {
//     "columnName": "map",
//     "columnType": {
//       "typeId": 26,
//       "keyType": {
//         "typeId": 17
//       },
//       "valueType": {
//         "typeId": 17
//       }
//     }
//   },
//   {
//     "columnName": "union",
//     "columnType": {
//       "typeId": 28,
//       "memberTags": [
//         "name",
//         "age"
//       ],
//       "memberTypes": [
//         {
//           "typeId": 17
//         },
//         {
//           "typeId": 3
//         }
//       ]
//     }
//   }
// ]
```

Fetch Chunks

Fetch all chunks:

```
const chunks = await result.fetchAllChunks();
```

Fetch one chunk at a time:

```
const chunks = [];
while (true) {
  const chunk = await result.fetchChunk();
  // Last chunk will have zero rows.
  if (chunk.rowCount === 0) {
```

```
    break;
  }
  chunks.push(chunk);
}
```

For materialized (non-streaming) results, chunks can be read by index:

```
const rowCount = result.rowCount;
const chunkCount = result.chunkCount;
for (let i = 0; i < chunkCount; i++) {
  const chunk = result.getChunk(i);
  // ...
}
```

Get chunk data:

```
const rows = chunk.getRows();

const rowObjects = chunk.getRowObjects(result.deduplicatedColumnNames());

const columns = chunk.getColumns();

const columnsObject =
  chunk.getColumnsObject(result.deduplicatedColumnNames());
```

Get chunk data (one value at a time)

```
const columns = [];
const columnCount = chunk.columnCount;
for (let columnIndex = 0; columnIndex < columnCount; columnIndex++) {
  const columnValues = [];
  const columnVector = chunk.getColumnVector(columnIndex);
  const itemCount = columnVector.itemCount;
  for (let itemIndex = 0; itemIndex < itemCount; itemIndex++) {
    const value = columnVector.getItem(itemIndex);
    columnValues.push(value);
  }
  columns.push(columnValues);
}
```

Inspect Data Types

```
import { DuckDBTypeId } from '@duckdb/node-api';

if (columnType.typeId === DuckDBTypeId.ARRAY) {
  const arrayValueType = columnType.valueType;
  const arrayLength = columnType.length;
}

if (columnType.typeId === DuckDBTypeId.DECIMAL) {
  const decimalWidth = columnType.width;
  const decimalScale = columnType.scale;
}

if (columnType.typeId === DuckDBTypeId.ENUM) {
  const enumValues = columnType.values;
}

if (columnType.typeId === DuckDBTypeId.LIST) {
  const listValueType = columnType.valueType;
}

if (columnType.typeId === DuckDBTypeId.MAP) {
  const mapKeyType = columnType.keyType;
  const mapValueType = columnType.valueType;
```

```
}

if (columnType.typeId === DuckDBTypeId.STRUCT) {
  const structEntryNames = columnType.names;
  const structEntryTypes = columnType.valueTypes;
}

if (columnType.typeId === DuckDBTypeId.UNION) {
  const unionMemberTags = columnType.memberTags;
  const unionMemberTypes = columnType.memberTypes;
}

// For the JSON type (https://duckdb.org/docs/current/data/json/json_type)
if (columnType.alias === 'JSON') {
  const json = JSON.parse(columnValue);
}
```

Every type implements toString. The result is both human-friendly and readable by DuckDB in an appropriate expression.

```
const typeString = columnType.toString();
```

Inspect Data Values

```
import { DuckDBTypeId } from '@duckdb/node-api';

if (columnType.typeId === DuckDBTypeId.ARRAY) {
  const arrayItems = columnValue.items; // array of values
  const arrayString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.BIT) {
  const bools = columnValue.toBools(); // array of booleans
  const bits = columnValue.toBits(); // array of 0s and 1s
  const bitString = columnValue.toString(); // string of '0's and '1's
}

if (columnType.typeId === DuckDBTypeId.BLOB) {
  const blobBytes = columnValue.bytes; // Uint8Array
  const blobString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.DATE) {
  const dateDays = columnValue.days;
  const dateString = columnValue.toString();
  const { year, month, day } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.DECIMAL) {
  const decimalWidth = columnValue.width;
  const decimalScale = columnValue.scale;
  // Scaled-up value. Represented number is value/(10^scale).
  const decimalValue = columnValue.value; // bigint
  const decimalString = columnValue.toString();
  const decimalDouble = columnValue.toDouble();
}

if (columnType.typeId === DuckDBTypeId.INTERVAL) {
  const intervalMonths = columnValue.months;
  const intervalDays = columnValue.days;
  const intervalMicros = columnValue.micros; // bigint
  const intervalString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.LIST) {
```

```
  const listItems = columnValue.items; // array of values
  const listString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.MAP) {
  const mapEntries = columnValue.entries; // array of { key, value }
  const mapString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.STRUCT) {
  // { name1: value1, name2: value2, ... }
  const structEntries = columnValue.entries;
  const structString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_MS) {
  const timestampMillis = columnValue.milliseconds; // bigint
  const timestampMillisString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_NS) {
  const timestampNanos = columnValue.nanoseconds; // bigint
  const timestampNanosString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_S) {
  const timestampSecs = columnValue.seconds; // bigint
  const timestampSecsString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_TZ) {
  const timestampTZMicros = columnValue.micros; // bigint
  const timestampTZString = columnValue.toString();
  const {
    date: { year, month, day },
    time: { hour, min, sec, micros },
  } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP) {
  const timestampMicros = columnValue.micros; // bigint
  const timestampString = columnValue.toString();
  const {
    date: { year, month, day },
    time: { hour, min, sec, micros },
  } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.TIME_TZ) {
  const timeTZMicros = columnValue.micros; // bigint
  const timeTZOffset = columnValue.offset;
  const timeTZString = columnValue.toString();
  const {
    time: { hour, min, sec, micros },
    offset,
  } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.TIME) {
  const timeMicros = columnValue.micros; // bigint
  const timeString = columnValue.toString();
  const { hour, min, sec, micros } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.UNION) {
```

```
  const unionTag = columnValue.tag;
  const unionValue = columnValue.value;
  const unionValueString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.UUID) {
  const uuidHugeint = columnValue.hugeint; // bigint
  const uuidString = columnValue.toString();
}

// other possible values are: null, boolean, number, bigint, or string
```

Displaying Timezones

Converting a TIMESTAMP_TZ value to a string depends on a timezone offset. By default, this is set to the offset for the local timezone when the Node process is started.

To change it, set the `timezoneOffsetInMinutes` property of `DuckDBTimestampTZValue`:

```
DuckDBTimestampTZValue.timezoneOffsetInMinutes = -8 * 60;
const pst = DuckDBTimestampTZValue.Epoch.toString();
// 1969-12-31 16:00:00-08

DuckDBTimestampTZValue.timezoneOffsetInMinutes = +1 * 60;
const cet = DuckDBTimestampTZValue.Epoch.toString();
// 1970-01-01 01:00:00+01
```

Note that the timezone offset used for this string conversion is distinct from the `TimeZone` setting of DuckDB.

The following sets this offset to match the `TimeZone` setting of DuckDB:

```
const reader = await connection.runAndReadAll(
  `select (timezone(current_timestamp) / 60)::int`
);
DuckDBTimestampTZValue.timezoneOffsetInMinutes =
  reader.getColumns()[0][0];
```

Append To Table

```
await connection.run(
  `create or replace table target_table(i integer, v varchar)`
);

const appender = await connection.createAppender('target_table');

appender.appendInteger(42);
appender.appendVarchar('duck');
appender.endRow();

appender.appendInteger(123);
appender.appendVarchar('mallard');
appender.endRow();

appender.flushSync();

appender.appendInteger(17);
appender.appendVarchar('goose');
appender.endRow();

appender.closeSync(); // also flushes
```

Append Data Chunk

```
await connection.run(
  `create or replace table target_table(i integer, v varchar)`
);

const appender = await connection.createAppender('target_table');

const chunk = DuckDBDataChunk.create([INTEGER, VARCHAR]);
chunk.setColumns([
  [42, 123, 17],
  ['duck', 'mallard', 'goose'],
]);
// OR:
// chunk.setRows([
//   [42, 'duck'],
//   [123, 'mallard'],
//   [17, 'goose'],
// ]);

appender.appendDataChunk(chunk);
appender.flushSync();
```

See "Specifying Values" above for how to supply values to the appender.

Extract Statements

```
const extractedStatements = await connection.extractStatements(`
  create or replace table numbers as from range(?);
  from numbers where range < ?;
  drop table numbers;
`);
const parameterValues = [10, 7];
const statementCount = extractedStatements.count;
for (let stmtIndex = 0; stmtIndex < statementCount; stmtIndex++) {
  const prepared = await extractedStatements.prepare(stmtIndex);
  let parameterCount = prepared.parameterCount;
  for (let paramIndex = 1; paramIndex <= parameterCount; paramIndex++) {
    prepared.bindInteger(paramIndex, parameterValues.shift());
  }
  const result = await prepared.run();
  // ...
}
```

Control Evaluation of Tasks

```
import { DuckDBPendingResultState } from '@duckdb/node-api';

async function sleep(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

const prepared = await connection.prepare('from range(10_000_000)');
const pending = prepared.start();
while (pending.runTask() !== DuckDBPendingResultState.RESULT_READY) {
  console.log('not ready');
  await sleep(1);
}
console.log('ready');
const result = await pending.getResult();
// ...
```

Ways to Run SQL

```
// Run to completion but don't yet retrieve any rows.
// Optionally take values to bind to SQL parameters,
// and (optionally) types of those parameters,
// either as an array (for positional parameters),
// or an object keyed by parameter name.
const result = await connection.run(sql);
const result = await connection.run(sql, values);
const result = await connection.run(sql, values, types);

// Run to completion but don't yet retrieve any rows.
// Wrap in a DuckDBDataReader for convenient data retrieval.
const reader = await connection.runAndRead(sql);
const reader = await connection.runAndRead(sql, values);
const reader = await connection.runAndRead(sql, values, types);

// Run to completion, wrap in a reader, and read all rows.
const reader = await connection.runAndReadAll(sql);
const reader = await connection.runAndReadAll(sql, values);
const reader = await connection.runAndReadAll(sql, values, types);

// Run to completion, wrap in a reader, and read at least
// the given number of rows. (Rows are read in chunks, so more than
// the target may be read.)
const reader = await connection.runAndReadUntil(sql, targetRowCount);
const reader =
  await connection.runAndReadUntil(sql, targetRowCount, values);
const reader =
  await connection.runAndReadUntil(sql, targetRowCount, values, types);

// Create a streaming result and don't yet retrieve any rows.
const result = await connection.stream(sql);
const result = await connection.stream(sql, values);
const result = await connection.stream(sql, values, types);

// Create a streaming result and don't yet retrieve any rows.
// Wrap in a DuckDBDataReader for convenient data retrieval.
const reader = await connection.streamAndRead(sql);
const reader = await connection.streamAndRead(sql, values);
const reader = await connection.streamAndRead(sql, values, types);

// Create a streaming result, wrap in a reader, and read all rows.
const reader = await connection.streamAndReadAll(sql);
const reader = await connection.streamAndReadAll(sql, values);
const reader = await connection.streamAndReadAll(sql, values, types);

// Create a streaming result, wrap in a reader, and read at least
// the given number of rows.
const reader = await connection.streamAndReadUntil(sql, targetRowCount);
const reader =
  await connection.streamAndReadUntil(sql, targetRowCount, values);
const reader =
  await connection.streamAndReadUntil(sql, targetRowCount, values, types);

// Prepared Statements

// Prepare a possibly-parametered SQL statement to run later.
const prepared = await connection.prepare(sql);

// Bind values to the parameters.
prepared.bind(values);
prepared.bind(values, types);

// Run the prepared statement. These mirror the methods on the connection.
```

```
const result = prepared.run();

const reader = prepared.runAndRead();
const reader = prepared.runAndReadAll();
const reader = prepared.runAndReadUntil(targetRowCount);

const result = prepared.stream();

const reader = prepared.streamAndRead();
const reader = prepared.streamAndReadAll();
const reader = prepared.streamAndReadUntil(targetRowCount);

// Pending Results

// Create a pending result.
const pending = await connection.start(sql);
const pending = await connection.start(sql, values);
const pending = await connection.start(sql, values, types);

// Create a pending, streaming result.
const pending = await connection.startStream(sql);
const pending = await connection.startStream(sql, values);
const pending = await connection.startStream(sql, values, types);

// Create a pending result from a prepared statement.
const pending = await prepared.start();
const pending = await prepared.startStream();

while (pending.runTask() !== DuckDBPendingResultState.RESULT_READY) {
  // optionally sleep or do other work between tasks
}

// Retrieve the result. If not yet READY, will run until it is.
const result = await pending.getResult();

const reader = await pending.read();
const reader = await pending.readAll();
const reader = await pending.readUntil(targetRowCount);
```

Ways to Get Result Data

```
// From a result

// Asynchronously retrieve data for all rows:
const columns = await result.getColumns();
const columnsJson = await result.getColumnsJson();
const columnsObject = await result.getColumnsObject();
const columnsObjectJson = await result.getColumnsObjectJson();
const rows = await result.getRows();
const rowsJson = await result.getRowsJson();
const rowObjects = await result.getRowObjects();
const rowObjectsJson = await result.getRowObjectsJson();

// From a reader

// First, (asynchronously) read some rows:
await reader.readAll();
// or:
await reader.readUntil(targetRowCount);

// Then, (synchronously) get result data for the rows read:
const columns = reader.getColumns();
const columnsJson = reader.getColumnsJson();
const columnsObject = reader.getColumnsObject();
```

```
const columnsObjectJson = reader.getColumnsObjectJson();
const rows = reader.getRows();
const rowsJson = reader.getRowsJson();
const rowObjects = reader.getRowObjects();
const rowObjectsJson = reader.getRowObjectsJson();

// Individual values can also be read directly:
const value = reader.value(columnIndex, rowIndex);

// Using chunks

// If desired, one or more chunks can be fetched from a result:
const chunk = await result.fetchChunk();
const chunks = await result.fetchAllChunks();

// And then data can be retrieved from each chunk:
const columnValues = chunk.getColumnValues(columnIndex);
const columns = chunk.getColumns();
const rowValues = chunk.getRowValues(rowIndex);
const rows = chunk.getRows();

// Or, values can be visited:
chunk.visitColumnValues(columnIndex,
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);
chunk.visitColumns((column, columnIndex, type) => { /* ... */ });
chunk.visitColumnMajor(
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);
chunk.visitRowValues(rowIndex,
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);
chunk.visitRows((row, rowIndex) => { /* ... */ });
chunk.visitRowMajor(
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);

// Or converted:
// The `converter` argument implements `DuckDBValueConverter`,
// which has the single method convertValue(value, type).
const columnValues = chunk.convertColumnValues(columnIndex, converter);
const columns = chunk.convertColumns(converter);
const rowValues = chunk.convertRowValues(rowIndex, converter);
const rows = chunk.convertRows(converter);

// The reader abstracts these low-level chunk manipulations
// and is recommended for most cases.
```

ODBC

ODBC API Overview

> Installation To use the DuckDB ODBC client, visit the ODBC installation page.
>
> The latest stable version of the DuckDB ODBC client is {{ latest_version }}.

> This page describes the DuckDB ODBC driver, which lets other applications connect *to DuckDB*. To connect *from DuckDB to other databases* over ODBC, use the odbc extension.

The ODBC (Open Database Connectivity) is a C-style API that provides access to different flavors of Database Management Systems (DBMSs). The ODBC API consists of the Driver Manager (DM) and the ODBC drivers.

The Driver Manager is part of the system library, e.g., unixODBC, which manages the communications between the user applications and the ODBC drivers. Typically, applications are linked against the DM, which uses Data Source Name (DSN) to look up the correct ODBC driver.

The ODBC driver is a DBMS implementation of the ODBC API, which handles all the internals of that DBMS.

The DM maps user application calls of ODBC functions to the correct ODBC driver that performs the specified function and returns the proper values.

DuckDB ODBC Driver

DuckDB supports the ODBC version 3.0 according to the Core Interface Conformance.

The ODBC driver is available for all operating systems. Visit the installation page for direct links.

ODBC API on Linux

Driver Manager

A driver manager is required to manage communication between applications and the ODBC driver. We tested and support `unixODBC` that is a complete ODBC driver manager for Linux. Users can install it from the command line:

On Debian-based distributions (Ubuntu, Mint, etc.), run:

```
sudo apt-get install unixodbc odbcinst
```

On Fedora-based distributions (Amazon Linux, RHEL, CentOS, etc.), run:

```
sudo yum install unixODBC
```

Setting Up the Driver

1. Download the ODBC Linux Asset corresponding to your architecture:
    - x86_64 (AMD64)
    - arm64 (AArch64)
2. The package contains the following files:

- `libduckdb_odbc.so`: the DuckDB driver.
- `unixodbc_setup.sh`: a setup script to aid the configuration on Linux.

To extract them, run:

```
mkdir duckdb_odbc && unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

3. The `unixodbc_setup.sh` script performs the configuration of the DuckDB ODBC Driver. It is based on the unixODBC package that provides some commands to handle the ODBC setup and test like `odbcinst` and `isql`.

   Run the following commands with either option `-u` or `-s` to configure DuckDB ODBC.

   The `-u` option based on the user home directory to setup the ODBC init files.

   ```
   ./unixodbc_setup.sh -u
   ```

   The `-s` option changes the system level files that will be visible for all users, because of that it requires root privileges.

   ```
   sudo ./unixodbc_setup.sh -s
   ```

   The option `--help` shows the usage of `unixodbc_setup.sh` prints the help.

   ```
   ./unixodbc_setup.sh --help
   ```

   ```
   Usage: ./unixodbc_setup.sh <level> [options]

   Example: ./unixodbc_setup.sh -u -db ~/database_path -D ~/driver_path/libduckdb_odbc.so

   Level:
   -s: System-level, using 'sudo' to configure DuckDB ODBC at the system-level, changing the files:
   /etc/odbc[inst].ini
   -u: User-level, configuring the DuckDB ODBC at the user-level, changing the files: ~/.odbc[inst].ini.

   Options:
   -db database_path>: the DuckDB database file path, the default is ':memory:' if not provided.
   -D driver_path: the driver file path (i.e., the path for libduckdb_odbc.so), the default is using the
   base script directory
   ```

4. The ODBC setup on Linux is based on the `.odbc.ini` and `.odbcinst.ini` files.

   These files can be placed to the user home directory `/home/<username>`{:.language-sql .highlight} or in the system `/etc`{:.language-sql .highlight} directory. The Driver Manager prioritizes the user configuration files over the system files.

   For the details of the configuration parameters, see the ODBC configuration page.

ODBC API on Windows

Setup

Using the DuckDB ODBC API on Windows requires the following steps:

1. Microsoft Windows requires an ODBC Driver Manager to manage communication between applications and the ODBC drivers. The Driver Manager on Windows is provided in a DLL file `odbccp32.dll`, and other files and tools. For detailed information check out the Common ODBC Component Files.
2. DuckDB releases the ODBC driver as an asset. For Windows, download it from the Windows ODBC asset (x86_64/AMD64).
3. The archive contains the following artifacts:

   - `duckdb_odbc.dll`: the DuckDB driver compiled for Windows.
   - `duckdb_odbc_setup.dll`: a setup DLL used by the Windows ODBC Data Source Administrator tool.
   - `odbc_install.exe`: an installation script to aid the configuration on Windows.

   Decompress the archive to a directory (e.g., `duckdb_odbc`).

4. The `odbc_install.exe` binary performs the configuration of the DuckDB ODBC Driver on Windows. It depends on the `Odbccp32.dll` that provides functions to configure the ODBC registry entries.

   Inside the permanent directory (e.g., `duckdb_odbc`), double-click on the `odbc_install.exe`.

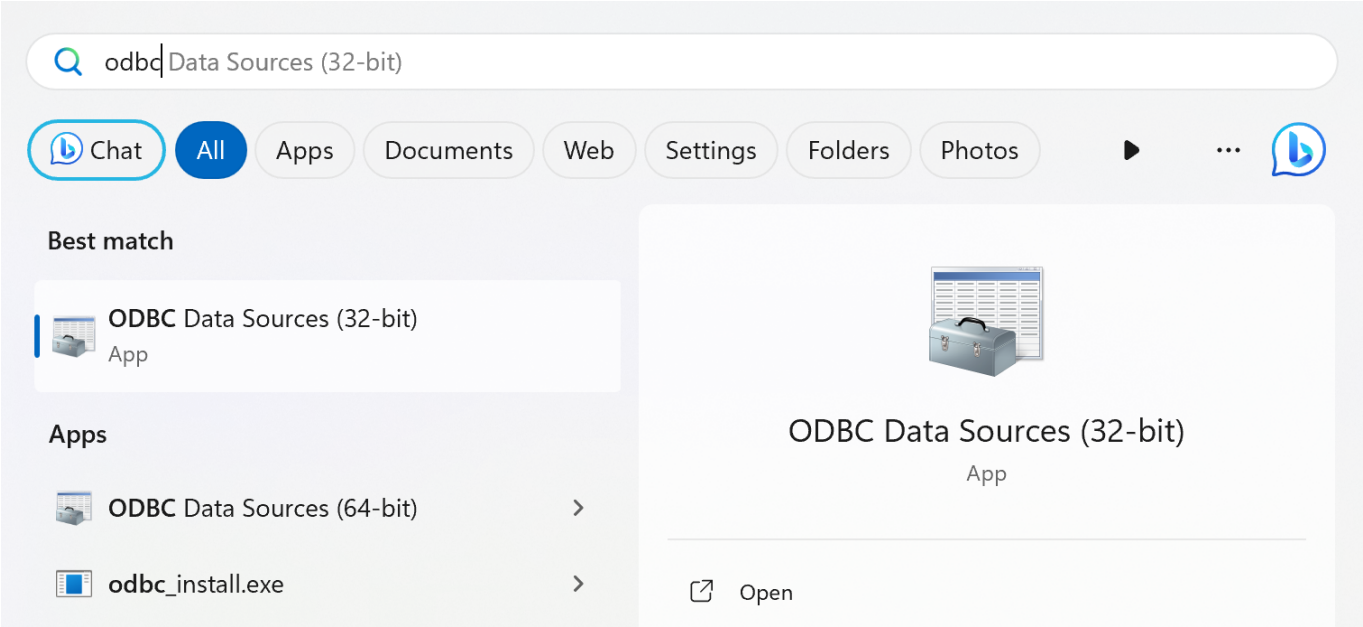
   Windows administrator privileges are required. In case of a non-administrator, a User Account Control prompt will occur.

5. `odbc_install.exe` adds a default DSN configuration into the ODBC registries with a default database `:memory:`.

DSN Windows Setup

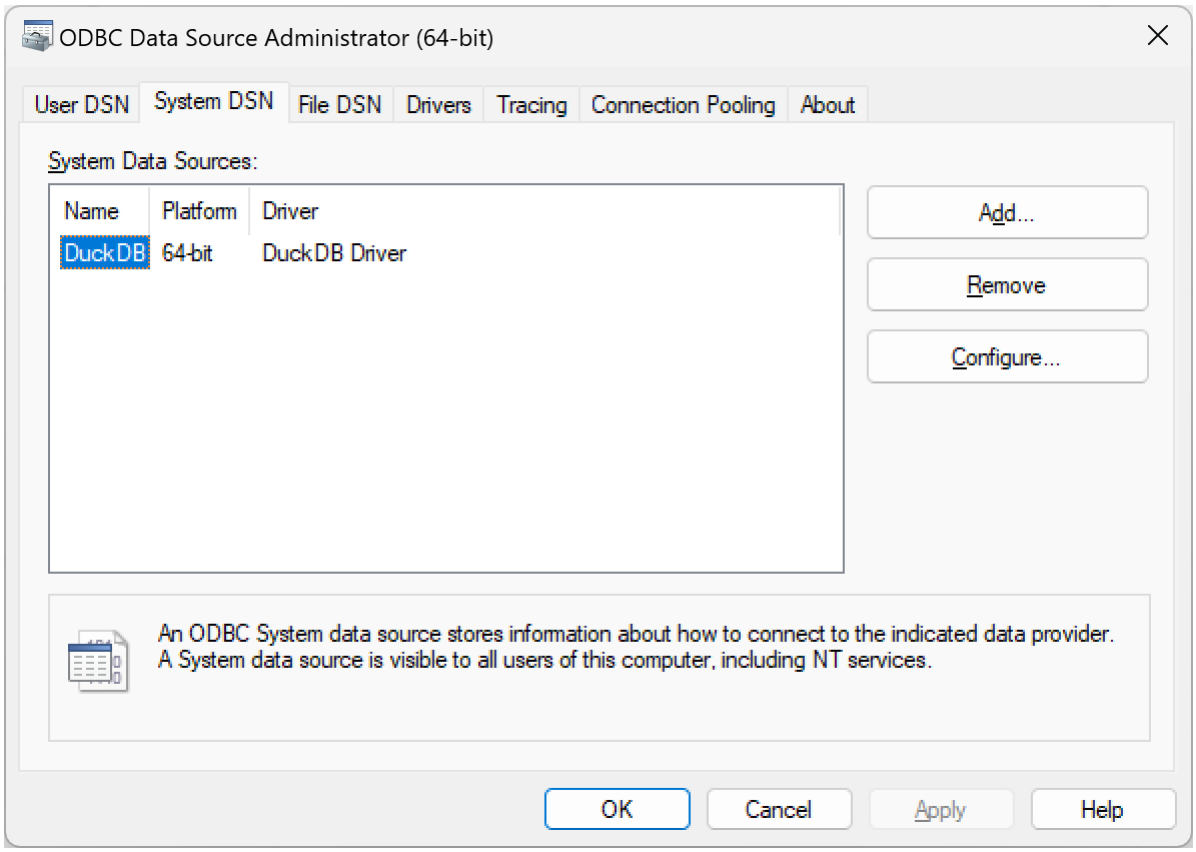
After the installation, it is possible to change the default DSN configuration or add a new one using the Windows ODBC Data Source Administrator tool `odbcad32.exe`.

It also can be launched through the Windows start:



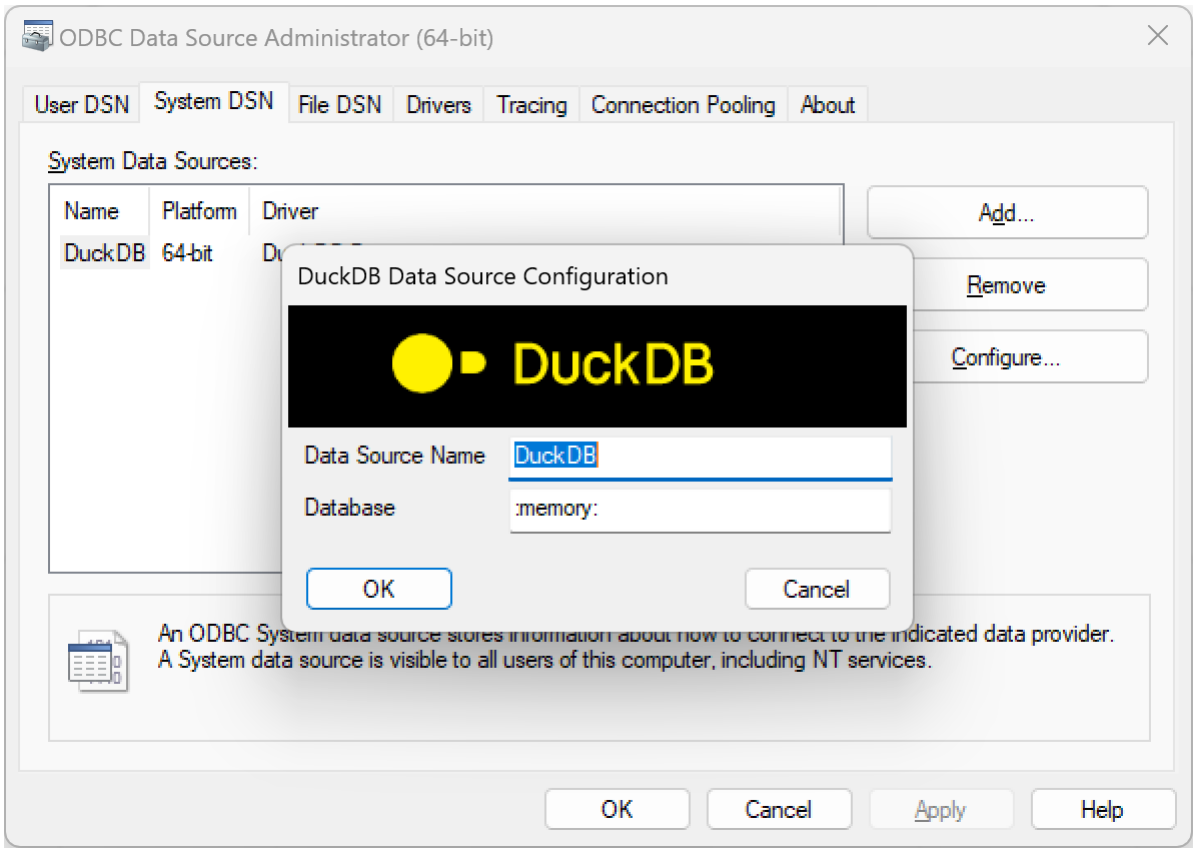
Default DuckDB DSN

The newly installed DSN is visible on the ***System DSN*** in the Windows ODBC Data Source Administrator tool:



Changing DuckDB DSN

When selecting the default DSN (i.e., DuckDB) or adding a new configuration, the following setup window will display:



This window allows you to set the DSN and the database file path associated with that DSN.

More Detailed Windows Setup

There are two ways to configure the ODBC driver, either by altering the registry keys as detailed below, or by connecting with `SQLDriverConnect`. A combination of the two is also possible.

Furthermore, the ODBC driver supports all the configuration options included in DuckDB.

> If a configuration is set in both the connection string passed to `SQLDriverConnect` and in the `odbc.ini` file, the one passed to `SQLDriverConnect` will take precedence.

For the details of the configuration parameters, see the ODBC configuration page.

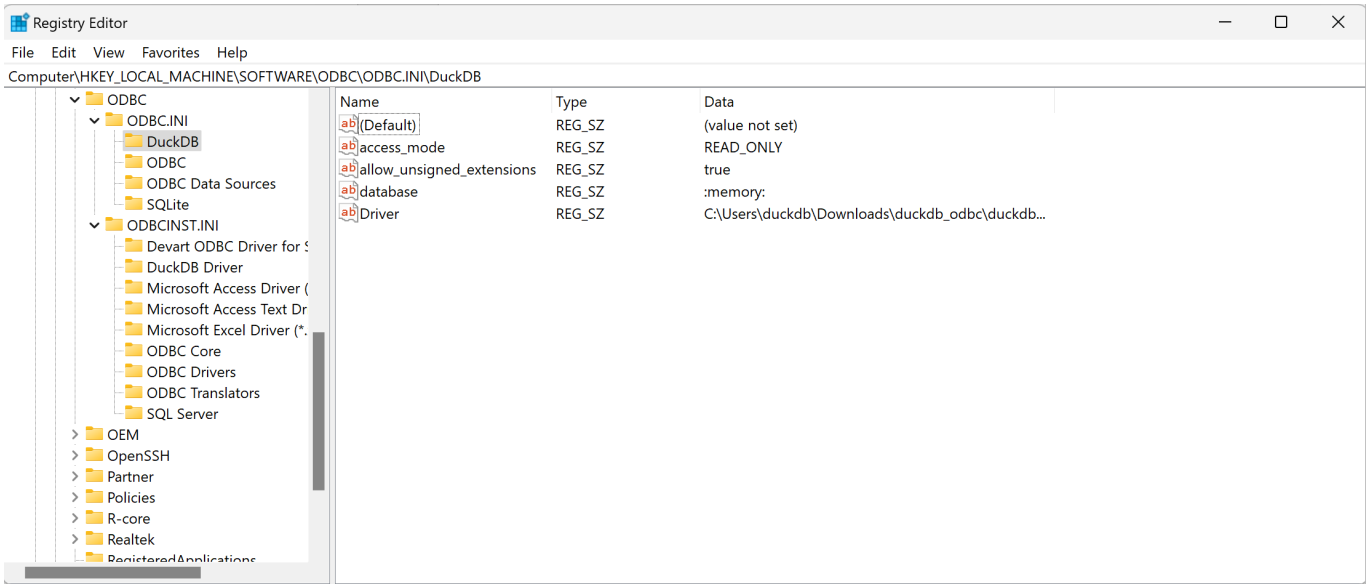
Registry Keys

The ODBC setup on Windows is based on registry keys (see Registry Entries for ODBC Components). The ODBC entries can be placed at the current user registry key (HKCU) or the system registry key (HKLM).

We have tested and used the system entries based on `HKLM->SOFTWARE->ODBC`. The `odbc_install.exe` changes this entry that has two subkeys: `ODBC.INI` and `ODBCINST.INI`.

The `ODBC.INI` is where users usually insert DSN registry entries for the drivers.

For example, the DSN registry for DuckDB would look like this:



The `ODBCINST.INI` contains one entry for each ODBC driver and other keys predefined for Windows ODBC configuration.

Updating the ODBC Driver

When a new version of the ODBC driver is released, installing the new version will overwrite the existing one. However, the installer doesn't always update the version number in the registry. To ensure the correct version is used, check that `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\DuckDB Driver` has the most recent version, and `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\DuckDB\Driver` has the correct path to the new driver.

ODBC API on macOS

1. A driver manager is required to manage communication between applications and the ODBC driver. DuckDB supports `unixODBC`, which is a complete ODBC driver manager for macOS and Linux. Users can install it from the command line via Homebrew:

   ```
   brew install unixodbc
   ```

2. DuckDB releases ODBC drivers for macOS on Apple Silicon (`arm64`) and Intel (`x86_64`) CPUs. To download the driver for Apple Silicon, run:

```
wget https://github.com/duckdb/duckdb-odbc/releases/download/v{{ latest_package }}/duckdb_
odbc-macos-arm64.zip
```

To download the driver for Intel, run:

```
wget https://github.com/duckdb/duckdb-odbc/releases/download/v{{ latest_package }}/duckdb_
odbc-macos-amd64.zip
```

1. The archive contains the `libduckdb_odbc.dylib` artifact. To extract it to a directory, run (on Intel, replace `arm64` with `amd64`):

   ```
   mkdir duckdb_odbc && unzip duckdb_odbc-macos-arm64.zip -d duckdb_odbc
   ```

2. There are two ways to configure the ODBC driver, either by initializing via the configuration files, or by connecting with `SQLDriverConnect`. A combination of the two is also possible.

   Furthermore, the ODBC driver supports all the configuration options included in DuckDB.

   > If a configuration is set in both the connection string passed to `SQLDriverConnect` and in the `odbc.ini` file, the one passed to `SQLDriverConnect` will take precedence.

   For the details of the configuration parameters, see the ODBC configuration page.

3. After the configuration, to validate the installation, it is possible to use an ODBC client. unixODBC uses a command line tool called `isql`.

   Use the DSN defined in `odbc.ini` as a parameter of `isql`.

```
isql DuckDB
```

```
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| echo [string]                         |
| quit                                  |
|                                       |
+---------------------------------------+
```

```
SQL> SELECT 42;
```

```
+------------+
| 42         |
+------------+
| 42         |
+------------+

SQLRowCount returns -1
1 rows fetched
```

ODBC Configuration

This page documents the files using the ODBC configuration, `odbc.ini` and `odbcinst.ini`. These are either placed in the home directory as dotfiles (`.odbc.ini` and `.odbcinst.ini`, respectively) or in a system directory. For platform-specific details, see the pages for Linux, macOS, and Windows.

`odbc.ini` and `.odbc.ini`

The `odbc.ini` file contains the DSNs for the drivers, which can have specific knobs. An example of `odbc.ini` with DuckDB:

```
[DuckDB]
Driver = DuckDB Driver
Database = /path/to/database.duckdb
access_mode = read_only
```

The lines correspond to the following parameters:

- `[DuckDB]`: between the brackets is a DSN for the DuckDB.
- `Driver`: Describes the driver's name, as well as where to find the configurations in the `odbcinst.ini`.
- `Database`: Describes the database name used by DuckDB, can also be a file path to a `.db` in the system.
- `access_mode`: The mode in which to connect to the database.

`odbcinst.ini` and `.odbcinst.ini`

The `odbcinst.ini` file contains general configurations for the ODBC installed drivers in the system. A driver section starts with the driver name between brackets, and then it follows specific configuration knobs belonging to that driver.

Example of `odbcinst.ini` with the DuckDB:

```
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace

[DuckDB Driver]
Driver = /path/to/libduckdb_odbc.dylib
```

The lines correspond to the following parameters:

- `[ODBC]`: The DM configuration section.
- `Trace`: Enables the ODBC trace file using the option `yes`.
- `TraceFile`: The absolute system file path for the ODBC trace file.
- `[DuckDB Driver]`: The section of the DuckDB installed driver.
- `Driver`: The absolute system file path of the DuckDB driver. Change to match your configuration.

Python

Python API

> Installation To use the DuckDB Python client, visit the Python installation page.
>
> The latest stable version of the DuckDB Python client is 2.0.0.

Installation

The DuckDB Python API can be installed using pip: `pip install duckdb`. Please see the installation page: `conda install python-duckdb -c conda-forge`.

Python version: DuckDB requires Python 3.11 or newer. Free-threaded builds of CPython are not supported.

If you are upgrading from DuckDB 1.x, see Migrating Python Code to DuckDB 2.0.

Basic API Usage

The most straight-forward manner of running SQL queries using DuckDB is using the `duckdb.sql` command.

```
import duckdb

duckdb.sql("SELECT 42").show()
```

This runs the query on an **in-memory database** that is stored globally inside the Python module. The first call that needs this database creates it.

`duckdb.sql()` returns a **relation**: an object you can query further or convert to Python data. A query that only reads data runs when you fetch or display its result. Reading it again runs the query again. Use `duckdb.execute()` for statements such as `CREATE TABLE` and `INSERT`. See Lazy Evaluation for details.

Relations can be referenced in subsequent queries by storing them inside variables, and using them as tables. This way queries can be constructed incrementally.

```
import duckdb

r1 = duckdb.sql("SELECT 42 AS i")
duckdb.sql("SELECT i * 2 AS k FROM r1").show()
```

Data Input

DuckDB can ingest data from a wide variety of formats – both on-disk and in-memory. See the data ingestion page for more information.

```
import duckdb

duckdb.read_csv("example.csv")                # read a CSV file into a Relation
duckdb.read_parquet("example.parquet")        # read a Parquet file into a Relation
duckdb.read_json("example.json")              # read a JSON file into a Relation

duckdb.sql("SELECT * FROM 'example.csv'")     # directly query a CSV file
duckdb.sql("SELECT * FROM 'example.parquet'") # directly query a Parquet file
duckdb.sql("SELECT * FROM 'example.json'")    # directly query a JSON file
```

DataFrames

DuckDB can directly query Pandas DataFrames, Polars DataFrames and Arrow tables. Note that these are read-only, i.e., editing these tables via INSERT or UPDATE statements is not possible.

Pandas To directly query a Pandas DataFrame, run:

```
import duckdb
import pandas as pd

pandas_df = pd.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM pandas_df")
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

Polars To directly query a Polars DataFrame, run:

```
import duckdb
import polars as pl

polars_df = pl.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM polars_df")
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

PyArrow To directly query a PyArrow table, run:

```
import duckdb
import pyarrow as pa

arrow_table = pa.Table.from_pydict({"a": [42]})
duckdb.sql("SELECT * FROM arrow_table")
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

Result Conversion

DuckDB supports converting query results efficiently to a variety of formats. See the result conversion page for more information.

```
import duckdb

duckdb.sql("SELECT 42").fetchall()        # Python objects
duckdb.sql("SELECT 42").df()              # Pandas DataFrame
duckdb.sql("SELECT 42").pl()              # Polars DataFrame
duckdb.sql("SELECT 42").to_arrow_table()  # Arrow Table
duckdb.sql("SELECT 42").to_arrow_reader() # Arrow RecordBatchReader
duckdb.sql("SELECT 42").fetchnumpy()      # NumPy Arrays
```

Writing Data to Disk

DuckDB supports writing Relation objects directly to disk in a variety of formats. The COPY statement can be used to write data to disk using SQL as an alternative.

```
import duckdb

duckdb.sql("SELECT 42").write_parquet("out.parquet")  # Write to a Parquet file
duckdb.sql("SELECT 42").write_csv("out.csv")          # Write to a CSV file
duckdb.execute("COPY (SELECT 42) TO 'out.parquet'")   # Copy to a Parquet file
```

Connection Options

Applications can open a new DuckDB connection via the `duckdb.connect()` method.

Using an In-Memory Database

When using DuckDB through `duckdb.sql()`, it operates on an **in-memory** database, i.e., no tables are persisted on disk. Invoking the `duckdb.connect()` method without arguments returns a connection, which also uses an in-memory database:

```
import duckdb

con = duckdb.connect()
con.sql("SELECT 42 AS x").show()
```

Persistent Storage

The `duckdb.connect(dbname)` creates a connection to a **persistent** database. Any data written to that connection will be persisted, and can be reloaded by reconnecting to the same file, both from Python and from other DuckDB clients.

```
import duckdb

# create a connection to a file called 'file.db'
con = duckdb.connect("file.db")
# create a table and load data into it
con.execute("CREATE TABLE test (i INTEGER)")
con.execute("INSERT INTO test VALUES (42)")
# query the table
con.table("test").show()
# explicitly close the connection
con.close()
# Note: connections also closed implicitly when they go out of scope
```

When you reconnect with a context manager, the connection is closed at the end of the `with` block:

```
import duckdb

with duckdb.connect("file.db") as con:
    con.execute("INSERT INTO test VALUES (43)")
    con.table("test").show()
    # the context manager closes the connection automatically
```

Configuration

The `duckdb.connect()` method takes three arguments: `database`, `read_only` and `config`. Settings go into the `config` dictionary, where configuration options can be specified. For example:

```
import duckdb

con = duckdb.connect(config={"threads": 1})
```

Besides its worker threads, each database starts a pool of asynchronous I/O threads: by default four times the number of CPU cores, at most 256. To start fewer, set `async_threads`:

```
con = duckdb.connect(config={"threads": 4, "async_threads": 4})
```

To specify the storage version, pass the `storage_compatibility_version` option. DuckDB 1.5 cannot open a database file that DuckDB 2.0 wrote in its default format. To keep a file readable by DuckDB 1.5, write it with `v1.5.0`:

```
con = duckdb.connect("compatible.db", config={"storage_compatibility_version": "v1.5.0"})
```

Connection Object and Module

Nearly every method of a connection object is also a function of the `duckdb` module, which runs it on a global in-memory database. A few differ: `duckdb.df(frame)` builds a relation from a DataFrame, as `con.from_df(frame)` does, while `con.df()` fetches the result of `execute()`; `duckdb.arrow(obj)` builds a relation from an Arrow object; and `description` and `rowcount` are functions of the module but properties of a connection.

> If you are developing a package designed for others to use, and use DuckDB in the package, it is recommended that you create connection objects instead of using the methods on the `duckdb` module. That is because the `duckdb` module uses a shared global database, which can cause hard to debug issues if used from within multiple different packages.

Using Connections in Parallel Python Programs

A connection runs one call at a time. A call from another thread while the connection is busy does not wait: it raises a `ResourceInUseException` saying `connection is busy with another call; it refuses instead of waiting, so use a separate connection from cursor()`.

`duckdb.sql()`, the other functions of the `duckdb` module and `duckdb.connect(":default:")` all use one shared default connection, so the same holds for them. To run queries in parallel, give each thread its own cursor:

```
import threading

import duckdb

con = duckdb.connect()
con.execute("CREATE TABLE numbers AS SELECT range AS n FROM range(1000)")


def count_even(cursor, results):
    results.append(cursor.sql("SELECT count(*) FROM numbers WHERE n % 2 = 0").fetchall())


results = []
threads = [threading.Thread(target=count_even, args=(con.cursor(), results)) for _ in range(4)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
print(results)
```

```
[[(500,)], [(500,)], [(500,)], [(500,)]]
```

About `cursor()` The `DuckDBPyConnection.cursor()` method opens a new connection to the same database. A cursor has its own transaction and its own session settings, such as `search_path`, so cursors can run queries in parallel. A global setting, such as `threads` or `memory_limit`, applies to the whole database, also when a cursor sets it. Closing a connection closes its cursors.

Code that runs inside a query, such as a Python function called from SQL, cannot use the connection that runs the query: the call raises an error saying `code running inside a query cannot use that query's connection; use a separate connection from cursor()`. Let the function use a cursor instead.

Loading and Installing Extensions

The `install_extension()` and `load_extension()` methods install and load an extension, as the `INSTALL` and `LOAD` statements do. Many core extensions are installed and loaded automatically when a query uses them.

Community Extensions

To load community extensions, use the `repository="community"` argument with the `install_extension` method.

For example, install and load the h3 community extension as follows:

```
import duckdb

con = duckdb.connect()
con.install_extension("h3", repository="community")
con.load_extension("h3")
```

Unsigned Extensions

To load unsigned extensions, use:

```
con = duckdb.connect(config={"allow_unsigned_extensions": "true"})
```

> **Warning.** Only load unsigned extensions from sources you trust. Avoid loading unsigned extensions over HTTP. Consult the Securing DuckDB page for guidelines on how to set up DuckDB in a secure manner.

Data Ingestion

This page contains examples for data ingestion to Python using DuckDB. First, import the DuckDB package:

```
import duckdb
```

Then, proceed with any of the following sections.

CSV Files

CSV files can be read using the `read_csv` function, called either from within Python or directly from within SQL. By default, the `read_csv` function attempts to auto-detect the CSV settings by sampling from the provided file.

Read from a file using fully auto-detected settings:

```
duckdb.read_csv("example.csv")
```

Read multiple CSV files from a folder:

```
duckdb.read_csv("folder/*.csv")
```

Specify options on how the CSV is formatted internally:

```
duckdb.read_csv("example.csv", header = False, sep = ",")
```

Override types of the first two columns:

```
duckdb.read_csv("example.csv", dtype = ["int", "varchar"])
```

Directly read a CSV file from within SQL:

```
duckdb.sql("SELECT * FROM 'example.csv'")
```

Call `read_csv` from within SQL:

```
duckdb.sql("SELECT * FROM read_csv('example.csv')")
```

Add a `filename` column that holds the file each row was read from:

```
duckdb.read_csv("example.csv", filename=True)
```

Read a file-like object, such as an `io.BytesIO` or an open file; this needs the `fsspec` package:

```
import io

duckdb.read_csv(io.BytesIO(b"a,b\n1,2\n"))
```

See the CSV Import page for more information. To write a relation to a CSV file, use `write_csv`. If you pass `overwrite=False` then it refuses to replace a file that exists.

Parquet Files

Parquet files can be read using the `read_parquet` function, called either from within Python or directly from within SQL.

Read from a single Parquet file:

```
duckdb.read_parquet("example.parquet")
```

Read multiple Parquet files from a folder:

```
duckdb.read_parquet("folder/*.parquet")
```

Read a Parquet file over https:

```
duckdb.read_parquet("https://some.url/some_file.parquet")
```

Read a list of Parquet files:

```
duckdb.read_parquet(["file1.parquet", "file2.parquet", "file3.parquet"])
```

Directly read a Parquet file from within SQL:

```
duckdb.sql("SELECT * FROM 'example.parquet'")
```

Call `read_parquet` from within SQL:

```
duckdb.sql("SELECT * FROM read_parquet('example.parquet')")
```

See the Parquet Loading page for more information. To write a relation to a Parquet file, use `write_parquet`; with `overwrite=False`, it refuses to replace a file that exists.

JSON Files

JSON files can be read using the `read_json` function, called either from within Python or directly from within SQL. By default, the `read_json` function will automatically detect if a file contains newline-delimited JSON or regular JSON, and will detect the schema of the objects stored within the JSON file.

Read from a single JSON file:

```
duckdb.read_json("example.json")
```

Read multiple JSON files from a folder:

```
duckdb.read_json("folder/*.json")
```

Directly read a JSON file from within SQL:

```
duckdb.sql("SELECT * FROM 'example.json'")
```

Call `read_json` from within SQL:

```
duckdb.sql("SELECT * FROM read_json('example.json')")
```

Directly Accessing DataFrames and Arrow Objects

DuckDB can query a Pandas DataFrame by the name of its variable, as if it were a table:

```
import duckdb
import pandas as pd

test_df = pd.DataFrame.from_dict({"i": [1, 2, 3, 4], "j": ["one", "two", "three", "four"]})
print(duckdb.sql("SELECT * FROM test_df").fetchall())
```

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

The same works for Pandas Series, Polars DataFrames, LazyFrames and Series, NumPy arrays, a `dict` of one-dimensional NumPy arrays, relations and Arrow objects.

The masked elements in a NumPy masked array are `NULL`. In a `dict` of arrays, names that differ only in case get a suffix, so the keys `a` and `A` become the columns `a` and `A_1`.

Use `register()` to give a DataFrame or Arrow object a SQL name. This also works for objects stored in a dictionary or class attribute.

```
import duckdb
import pandas as pd

my_dictionary = {}
my_dictionary["test_df"] = pd.DataFrame.from_dict({"i": [1, 2, 3, 4], "j": ["one", "two", "three", "four"]})
duckdb.register("test_df_view", my_dictionary["test_df"])
print(duckdb.sql("SELECT * FROM test_df_view").fetchall())
```

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

You can also create a table in DuckDB from the contents of the DataFrame (or the view):

```
# create a new table from the contents of a DataFrame
duckdb.execute("CREATE TABLE test_df_table AS SELECT * FROM test_df")
# insert into an existing table from the contents of the registered view
duckdb.execute("INSERT INTO test_df_table SELECT * FROM test_df_view")
```

A relation's `create` method also makes a table. A name with a catalog, such as `other_db.main.test_df_table`, creates the table in that attached database.

Only variables that are visible to Python code at the location of the `sql()` or `execute()` call can be queried by name. To also look in the functions that called it, set `python_scan_all_frames` to `true`. Accessing these variables is made possible by replacement scans. To disable replacement scans entirely, use:

```
SET python_enable_replacements = false;
```

The precedence of objects with the same name is as follows:

- Objects explicitly registered via `register()`
- Native DuckDB tables and views
- Replacement scans

DuckDB supports querying multiple types of Apache Arrow objects including tables, record batches, arrays and chunked arrays, datasets, RecordBatchReaders and scanners. See the Python guides for more examples.

- The filters of a query are pushed into Arrow datasets and Polars LazyFrames. Tables, record batches, readers, scanners and Polars DataFrames are read whole, and the filter applies to what they return.
- A `RecordBatchReader` can be read once. A second scan raises `InvalidInputException: This RecordBatchReader has already been consumed and cannot be scanned again.` To scan its data more than once, read it into a table first, for example with `read_all()`.
- Arrow tables and Polars DataFrames are scanned on all threads, even when they hold a single chunk. A query that depends on the order of rows without an `ORDER BY`, such as one using `first()`, can return different rows between runs.

Custom Arrow Objects

DuckDB can also consume any object that implements the Arrow PyCapsule interface through an `__arrow_c_stream__` method. Such objects can be queried directly through replacement scans and passed to `duckdb.from_arrow()`:

```
import duckdb
import pyarrow as pa

class MyArrowStream:
    def __init__(self, table):
        self._table = table

    def __arrow_c_stream__(self, requested_schema=None):
        return self._table.__arrow_c_stream__(requested_schema)

obj = MyArrowStream(pa.table({"i": [1, 2, 3]}))
print(duckdb.sql("SELECT sum(i) AS s FROM obj").fetchall())
print(duckdb.from_arrow(obj).fetchall())
```

```
[(6,)]
[(1,), (2,), (3,)]
```

Pandas DataFrames – `object` Columns

`pandas.DataFrame` columns of an `object` dtype require some special care, since this stores values of arbitrary type. To convert these columns to DuckDB, we first go through an analyze phase before converting the values. In this analyze phase a sample of all the rows of the column are analyzed to determine the target type. This sample size is by default set to 1000. If the type picked during the analyze step is incorrect, this will result in `Invalid Input Error: Failed to cast value`, in which case you will need to increase the sample size. The sample size can be changed by setting the `pandas_analyze_sample` config option.

```
# example setting the sample size to 100k
duckdb.execute("SET GLOBAL pandas_analyze_sample = 100_000")
```

Appending Rows

Use the appender to insert rows produced by Python. The `with` block writes the rows to the table when it finishes without an exception:

```
duckdb.execute("CREATE TABLE people (id INTEGER, name VARCHAR)")
with duckdb.appender("people") as appender:
    appender.append_row(1, "Mark")
```

Appender

Use an appender to insert rows that your Python code produces. It collects rows and writes them together in one SQL statement; this is called flushing. The `with` block below flushes on successful exit:

```
import duckdb

con = duckdb.connect()
con.execute("CREATE TABLE people (id INTEGER PRIMARY KEY, name VARCHAR)")

with con.appender("people") as appender:
    appender.append_row(1, "Mark")
    appender.append_rows([(2, "Hannes"), (3, "Gabor")])

print(con.sql("SELECT * FROM people ORDER BY id").fetchall())
```

```
[(1, 'Mark'), (2, 'Hannes'), (3, 'Gabor')]
```

For data already in a DataFrame or Arrow table, `INSERT INTO people SELECT * FROM df` is faster than the appender. For pandas DataFrames, `con.append("people", df)` is also faster. See Directly Accessing DataFrames and Arrow Objects.

Use `con.appender(table)` for table operations or `con.query_appender(statement, columns)` for custom SQL. For the appender in other clients, see the Appender page.

Both functions are also available on the `duckdb` module. They use the default connection unless you supply the `connection` argument.

Appending to a Table

`con.appender(table)` resolves the table once, when the appender is created, the way SQL resolves a table name. It appends to the table's columns in declared order and skips generated columns. `append_row` takes one value per column, and `append_rows` takes any iterable of rows. Values are converted exactly as `executemany` parameters are.

To append to some of the columns, list them in `columns`. The other columns take their default value, per row:

```
con.execute("CREATE SEQUENCE event_ids")
con.execute(
    "CREATE TABLE events (id INTEGER DEFAULT nextval('event_ids'), kind VARCHAR, note VARCHAR DEFAULT
    'none')"
)

appender = con.appender("events", columns=["kind"])
appender.append_row("login")
appender.append_row("logout")
appender.flush()
```

Updates, Upserts and Deletes

The `action` argument picks the statement: `"insert"` (the default), `"upsert"`, `"update"` or `"delete"`. An upsert inserts new rows and updates existing rows that have the same primary key or unique key.

```
con.execute("CREATE TABLE stock (sku VARCHAR PRIMARY KEY, qty INTEGER)")
con.execute("INSERT INTO stock VALUES ('widget', 3), ('gadget', 5)")

with con.appender("stock", action="upsert") as appender:
    appender.append_row("widget", 10)
    appender.append_row("gizmo", 1)

with con.appender("stock", action="update", key=["sku"]) as appender:
    appender.append_row("gadget", 0)

with con.appender("stock", action="delete", key=["sku"]) as appender:
    appender.append_row("gizmo")

print(con.sql("SELECT * FROM stock ORDER BY sku").fetchall())
```

```
[('gadget', 0), ('widget', 10)]
```

- `upsert` runs an `INSERT OR REPLACE`, so the table needs exactly one primary key or unique constraint. It takes `columns` like an insert; on a conflict, only the listed columns change.
- `update` matches rows on the `key` columns and sets every other appended column. Use `columns` to choose the appended columns.
- `delete` appends only the `key` columns, in the order of `key`, and deletes the rows that match.
- A `NULL` key matches nothing.
- A plain insert writes the rows in append order while `preserve_insertion_order` is on, which is the default. An upsert does not keep append order.
- When a flush contains duplicate keys, an upsert keeps one row per key value, including `NULL`; an update applies one input row; and `MERGE INTO` raises an error.

Custom Statements

`con.query_appender` runs your SQL statement when you flush. Declare the appended columns as a dictionary of names to types. Your statement reads the appended rows as `appender_data`.

To transform the rows first, supply a `source` query. This query reads the appended rows as `appender_buffer`, and your statement reads its result as `appender_data`. The example below uses `source` to look up the product ID for each product code (`sku`). The `source` can be any query that returns rows; DuckDB runs it as you wrote it.

```
con.execute("CREATE TABLE products (id INTEGER, sku VARCHAR)")
con.execute("INSERT INTO products VALUES (1, 'widget'), (2, 'gadget')")
con.execute("CREATE TABLE orders (product_id INTEGER, qty INTEGER)")

appender = con.query_appender(
    "INSERT INTO orders FROM appender_data",
    {"sku": "VARCHAR", "qty": "INTEGER"},
    source="SELECT p.id AS product_id, b.qty FROM appender_buffer AS b JOIN products AS p USING (sku)",
)
appender.append_row("gadget", 4)
appender.append_row("widget", 2)
appender.flush()
```

- The statement is one `INSERT` (including `INSERT OR REPLACE` and `ON CONFLICT`), `UPDATE`, `DELETE` or `MERGE INTO`.
- A type is anything the Types API accepts, such as `"INTEGER"`, `duckdb.sqltypes.INTEGER` or `int`.
- A declared column that the statement does not read is allowed.
- DataFrames and other Python objects named in `source` or in the statement are looked up once, where the appender is created, and reused at every flush. If you later create a DuckDB table with the same name, later flushes use that table instead.

Flushing

Each `flush()` applies the rows collected since the previous flush. It runs in your transaction if you started one with `BEGIN` or `con.begin()`, and otherwise commits by itself. A flush with no rows does nothing.

A `with` block flushes on successful exit. When an exception leaves the block, the buffered rows are discarded and the exception propagates. Outside a `with` block, call `flush()` to write the rows and `clear()` to discard them. There is no `close()` method. The appender remains usable after `flush()`, `clear()` or leaving a `with` block.

> **Warning.** Garbage collection discards buffered rows and emits a `ResourceWarning`. Call `flush()` or use a `with` block to write the rows.

Errors and Retrying

- The appender checks its statement when you create it, without running it. Invalid SQL, incorrect column counts, unsupported statement types, `RETURNING` clauses and SQL parameter placeholders such as `?` raise an exception at creation. These errors leave the transaction usable. Cast errors and constraint violations raise an exception during `flush()`.
- A flush that raises because an earlier result is still open or another call is running on the connection keeps its buffered rows, and so does an interrupted flush. Finish the result or wait for the other call, then call `flush()` again. Other failures discard the rows and raise an exception.
- An interrupted flush inside your transaction keeps its buffered rows but aborts the transaction. Roll back before retrying. Start a new transaction if the retry must be part of one.
- If you press Ctrl-C while a Python function called from SQL is running during a flush, DuckDB treats it as an error in that function and discards the buffered rows.

Memory Use

The rows waiting to be flushed count toward DuckDB's `memory_limit`. When they no longer fit in memory, DuckDB can store them in temporary files. If DuckDB has no temporary directory, a full buffer makes every query on the connection raise an `OutOfMemoryExcep-`

tion until you call `flush()` or `clear()`. If a write into the buffer fails, `append_row`, `append_rows` and `flush()` raise an exception until you call `clear()`.

Connections and Threads

Concurrent calls on the same appender raise an exception rather than wait. Calls from a callback during a flush also raise an exception. Use a separate appender for each thread. You can append rows or call `clear()` while an `execute()` result is open, but must finish that result before calling `flush()`: call `con.fetchall()`, call `con.fetchone()` repeatedly until it returns None, or call `con.complete()` or `con.abort()`.

Conversion between DuckDB and Python

This page documents the rules for converting DuckDB results to Python and Python objects to DuckDB.

Result Conversion: DuckDB Results to Python

A result is read from a relation or, when using the DB-API / `con.execute()`, from the connection. `con.sql()`, `con.table()`, the readers and the other relational methods return relations. Each has its own methods:

Result	Relation	DB-API 2.0 (`con.execute()`)
Python tuples	`fetchall()`, `iter_rows(batch_size=1024)`	`fetchone()`, `fetchmany(size=1)`, `fetchall()`
Python dicts	`to_dicts()`	
NumPy	`fetchnumpy()`	`fetchnumpy()`
Pandas	`df()`	`df()`, `fetch_df_chunk()`
Apache Arrow	`to_arrow_table()`, `to_arrow_reader()`	`to_arrow_table()`, `to_arrow_reader()`, both make a copy
Polars	`pl()`	
PyTorch	`torch()`	`torch()`
TensorFlow	`tf()`	`tf()`

Relations are recommended for most conversion, except when you want to do row-based work. A relation has no `fetchone()`, `fetchmany()` or `fetch_df_chunk()` because it is stateless, i.e. it does not hold a result itself. To get specific rows of a result with the relational API you would use `limit(n).fetchall()`, `iter_rows()` or `to_arrow_reader()` instead. Note that with the DB-API, after a partial read with `fetchone()` or `fetchmany()`, the `df()`, `fetchnumpy()` and Arrow methods of the connection return only the remaining rows.

Examples

Below are some examples. See the Python guides for more. To run the examples, use this table:

```
import duckdb

con = duckdb.connect()
con.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)")
con.execute("""
    INSERT INTO items VALUES
        ('jeans', 20.0, 1),
```

```
        ('hammer', 42.2, 2),
        ('laptop', 2000.0, 1),
        ('chainsaw', 500.0, 10),
        ('iphone', 300.0, 2)
""")
```

Fetch as Pandas DataFrame:

```
df = con.execute("SELECT * FROM items").df()
print(df)
```

```
       item   value  count
0     jeans    20.0      1
1    hammer    42.2      2
2    laptop  2000.0      1
3  chainsaw   500.0     10
4    iphone   300.0      2
```

Fetch as dictionary of NumPy arrays:

```
arr = con.execute("SELECT * FROM items").fetchnumpy()
print(arr)
```

```
{'item': array(['jeans', 'hammer', 'laptop', 'chainsaw', 'iphone'], dtype=object), 'value': array([  20. ,
42.2, 2000. ,  500. ,  300. ]), 'count': array([ 1,  2,  1, 10,  2], dtype=int32)}
```

Fetch as an Arrow table. Converting to Pandas afterwards just for pretty printing:

```
tbl = con.sql("SELECT * FROM items").to_arrow_table()
print(tbl.to_pandas())
```

```
       item    value  count
0     jeans    20.00      1
1    hammer    42.20      2
2    laptop  2000.00      1
3  chainsaw   500.00     10
4    iphone   300.00      2
```

NumPy

- `fetchnumpy()` fetches the data as a dictionary of NumPy arrays. A column without `NULL` values is a plain array. A column with `NULL` values is a masked array.

Pandas

- `df()` fetches the data as a Pandas DataFrame
- `fetchdf()` and, on the connection, `fetch_df()` are older names of `df()`
- `fetch_df_chunk(vectors_per_chunk)` fetches the next portion of the results of `con.execute()` into a DataFrame. Each chunk has at most the vector size (2048 by default) times `vectors_per_chunk` (1 by default) rows.

Apache Arrow

- `to_arrow_table()` fetches the data as an Arrow table
- `to_arrow_reader(batch_size)` returns an Arrow record batch reader with *at most* `batch_size` rows per batch (the default is 131072)
- `arrow()` is an older name of `to_arrow_reader()`, on relations only
- A relation is also an Arrow stream through the Arrow PyCapsule Interface. `pyarrow.table(rel)` returns an Arrow table and `polars.DataFrame(rel)` a Polars DataFrame, which unlike `pl()` needs no pyarrow

After `con.execute()`, `to_arrow_table()` and `to_arrow_reader()` first fetch the rows and then convert them, so they make an extra copy of the result, and the reader returns its first batch only after the whole query ran. They warn: `UserWarning: Arrow output on a DB-API result makes an extra copy of the rows, use con.sql(query).to_arrow_table() instead.` The module functions `duckdb.to_arrow_table()` and `duckdb.to_arrow_reader()` read the result of `duckdb.execute()` and warn the same way. For Arrow output, run the query with `con.sql(query)`.

> **Deprecated.** on relations, `fetch_arrow_table()`, `fetch_record_batch()` and `fetch_arrow_reader()` are deprecated aliases of `to_arrow_table()` and `to_arrow_reader()`. After `con.execute()`, `fetch_arrow_table()`, `fetch_record_batch()`, `arrow()` and `pl()` no longer exist, and neither do the module functions `duckdb.fetch_arrow_table()`, `duckdb.fetch_record_batch()` and `duckdb.pl()`.

DuckDB has a single `INTERVAL` type that stores months, days and microseconds, so it maps onto Arrow's `month_day_nano_interval` type. On import, every Arrow interval and duration type, including `duration[s]`, `duration[ms]`, `duration[us]` and `duration[ns]`, is read into that one `INTERVAL` type. As a result, a `duration` column that is passed through DuckDB and read back as Arrow is returned as a `month_day_nano_interval` rather than as its original `duration` type. This is expected. The resolution and unit of the original Arrow type are not preserved.

Polars

- `pl()` fetches the data of a relation as a Polars DataFrame, and `pl(lazy=True)` returns a Polars LazyFrame

Object Conversion: Python Object to DuckDB

Python objects are converted when they are passed as query parameters (`execute()`, `executemany()` and `sql(..., params=...)`), as rows of `values()`, as constants in expressions and as the value of a `duckdb.Value`. The rules are the same on all of these paths, except where a section below says otherwise.

This is a mapping of Python object types to DuckDB Logical Types:

- None → `NULL`
- `bool` → `BOOLEAN`
- `datetime.timedelta` → `INTERVAL`
- `str` → `VARCHAR`
- `bytearray` → `BLOB`
- `memoryview` → `BLOB`
- `decimal.Decimal` → `DECIMAL` / `DOUBLE` / `FLOAT`
- `uuid.UUID` → `UUID`

The rest of the conversion rules are as follows.

`str`

A `str` that UTF-8 cannot encode, such as one that holds a lone surrogate (`"\ud800"`), raises an `InvalidInputException` that says it `cannot encode str as UTF-8`. The connection stays usable.

`memoryview`

A `memoryview` converts to a `BLOB` of the bytes it shows. As a query parameter, it must be contiguous: a strided view such as `memoryview(b"abcdef")[::2]` raises `BufferError: memoryview: underlying buffer is not C-contiguous`. `values()` and expressions accept it and read its bytes in order, here `b'ace'`.

`decimal.Decimal`

A `Decimal` becomes a `DECIMAL` whose width and scale follow from its digits and its exponent: `Decimal("12.5")` is a `DECIMAL(3,1)`, `Decimal("1E+2")` is a `DECIMAL(3,0)` that holds 100 and `Decimal("1E-7")` is a `DECIMAL(8,7)`. A value that needs more than 38 digits becomes a `DOUBLE`, such as `Decimal("1E+38")` or `Decimal(0.1)`, which holds the 55 decimal digits of the binary float 0.1. NaN and the infinities become `FLOAT`.

`int`

Since integers can be of arbitrary size in Python, there is not a one-to-one conversion possible for ints. An `int` becomes the first of these types that fits it:

- `INTEGER`
- `BIGINT`
- `UBIGINT`
- `HUGEINT`
- `UHUGEINT`

So `2**31` is a `BIGINT`, `2**63` a `UBIGINT`, `2**64` and `-(2**63) - 1` are `HUGEINT` and `2**127` is a `UHUGEINT`. An integer that does not fit in 128 bits raises `InvalidInputException: Python integer too large for 128-bit integer type`. A subclass of `int` converts by the value it stores.

When using the DuckDB Value class, it's possible to set a target type, which will influence the conversion:

```
import duckdb
from duckdb.sqltypes import TINYINT

con = duckdb.connect()
print(con.execute("SELECT typeof(?), typeof(?), typeof(?)", [5, 2**64, duckdb.Value(5,
TINYINT)]).fetchall())
```

```
[('INTEGER', 'HUGEINT', 'TINYINT')]
```

`float`

A `float` is always a `DOUBLE`.

`datetime.datetime`

For `datetime` we will check `pandas.isnull` if it's available and return `NULL` if it returns `true`.

If the `datetime` has tzinfo, we will use `TIMESTAMPTZ`, otherwise it becomes `TIMESTAMP`. Fetching a `TIMESTAMPTZ` value into Python needs the `pytz` package, which is not installed with DuckDB; without it, the fetch raises `InvalidInputException: Required module 'pytz' failed to import`.

`datetime.datetime.max` and `datetime.datetime.min` convert to ordinary timestamps, not to infinity. In the other direction, the timestamp `infinity` is returned as `datetime.datetime.max` and `-infinity` as `datetime.datetime.min`.

`datetime.time`

If the `time` has tzinfo, we will use `TIMETZ`, otherwise it becomes `TIME`.

`datetime.date`

`date` converts to the `DATE` type. `datetime.date.max` and `datetime.date.min` convert to ordinary dates; the date `infinity` is returned as `datetime.date.max` and `-infinity` as `datetime.date.min`.

`bytes`

`bytes` converts to `BLOB` by default. When it's used to construct a Value object of type `BIT`, it maps to `BIT` instead, and its bytes are read as the characters `0` and `1`, as in `duckdb.Value(b"0101", duckdb.sqltypes.BIT)`.

`list`

`list` becomes a `LIST` type of the "most permissive" type of its children. For example, `[12345, "test"]` becomes `VARCHAR[]`, because 12345 can convert to `VARCHAR` but `test` can not convert to `INTEGER`:

```
print(con.execute("SELECT ?", [[12345, "test"]]).fetchall())
```

```
[(['12345', 'test'],)]
```

A `NaN` stays `NaN` inside a list, a struct, a map or a `duckdb.Value` of a nested type. A list that mixes `datetime` values with and without tzinfo becomes a `TIMESTAMPTZ` list as a query parameter and in `values()`, which read the values without tzinfo as UTC.

`dict`

The `dict` object can convert to either `STRUCT(...)` or `MAP(..., ...)`, depending on its keys and on where it is used:

- A dict whose keys are all strings becomes a `STRUCT` with one field per key.
- A dict with exactly the two keys `key` and `value`, each holding a list and both lists of the same length, becomes a `MAP` of the two lists zipped together when passed as a query parameter and in `values()`. In an expression, it is a `STRUCT` like any other dict with string keys.
- A dict with a key that is not a string becomes a `MAP` in `values()` and in expressions, written as a SQL MAP literal. As a query parameter, it becomes a `STRUCT` whose field names are the keys converted to strings. Keys or values of types that SQL cannot combine raise an error.

A dict in the `key` and `value` form:

```
my_map_dict = {
    "key": [
        1, 2, 3
    ],
    "value": [
        "one", "two", "three"
    ]
}

duckdb.values(my_map_dict).show()
```

becomes a `MAP(INTEGER, VARCHAR)`:

```
┌──────────────────────────┐
│           col0           │
│ map(integer, varchar)    │
├──────────────────────────┤
│ {1=one, 2=two, 3=three}  │
└──────────────────────────┘
```

A dict with string keys becomes a `STRUCT`:

```
my_struct_dict = {
    "name": "duck",
    "count": 2,
    "tags": [1, 2, 3],
    "active": True
}

duckdb.values(my_struct_dict).show()
```

```
┌──────────────────────────────────────────────────────────────────────┐
│                                 col0                                 │
│ struct("name" varchar, count integer, tags integer[], active boolean) │
├──────────────────────────────────────────────────────────────────────┤
│ {'name': duck, 'count': 2, 'tags': [1, 2, 3], 'active': true}        │
└──────────────────────────────────────────────────────────────────────┘
```

As a query parameter, a dict with other keys becomes a STRUCT with string field names:

```
my_mixed_dict = {1: "one", "2": 2, "three": [1, 2, 3], False: True}
print(con.execute("SELECT typeof(?)", [my_mixed_dict]).fetchone()[0])
```

```
STRUCT("1" VARCHAR, "2" INTEGER, three INTEGER[], "False" BOOLEAN)
```

A dict returned by a function converts to the function's return type. For a MAP return type, a dict in the `key` and `value` form gives the entries of the map, so the return type has to describe those entries:

```
duckdb_conn = duckdb.connect()

def get_map() -> dict[str, list[str] | list[int]]:
    return {
        "key": [
            1, 2, 3
        ],
        "value": [
            "one", "two", "three"
        ]
    }

duckdb_conn.create_function("get_map", get_map, return_type=dict[int, str])

duckdb_conn.sql("SELECT get_map()").show()
```

```
┌──────────────────────────┐
│        get_map()         │
│ map(bigint, varchar)     │
├──────────────────────────┤
│ {1=one, 2=two, 3=three}  │
└──────────────────────────┘
```

Without `return_type`, the annotation `dict[str, list[str] | list[int]]` is the return type, so each value, such as `"one"`, has to convert to a `UNION(u1 VARCHAR[], u2 BIGINT[])`, and the query fails:

```
duckdb_conn.create_function("get_map_error", get_map)

duckdb_conn.sql("SELECT get_map_error()").show()
```

```
_duckdb.ConversionException: Conversion Error: Could not convert value of type VARCHAR to UNION(u1
VARCHAR[], u2 BIGINT[])
```

A function annotated to return a `dict` returns a MAP, due to automatic conversion. To return a STRUCT, the `return_type` has to be provided:

```
from duckdb import list_type, struct_type
from duckdb.sqltypes import BOOLEAN, INTEGER, VARCHAR

def get_struct() -> dict[str, str | int | list[int] | bool]:
    return my_struct_dict

duckdb_conn.create_function("get_struct_as_map", get_struct)

duckdb_conn.sql("SELECT get_struct_as_map()").show()

duckdb_conn.create_function("get_struct", get_struct, return_type=struct_type({
    "name": VARCHAR,
    "count": INTEGER,
```

```
    "tags": list_type(INTEGER),
    "active": BOOLEAN
}))

duckdb_conn.sql("SELECT get_struct()").show()
```

```
┌──────────────────────────────────────────────────────────────────────┐
│                          get_struct_as_map()                         │
│ map(varchar, union(u1 varchar, u2 bigint, u3 bigint[], u4 boolean))  │
├──────────────────────────────────────────────────────────────────────┤
│ {name=duck, count=2, tags=[1, 2, 3], active=true}                    │
└──────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────┐
│                              get_struct()                            │
│ struct("name" varchar, count integer, tags integer[], active boolean)│
├──────────────────────────────────────────────────────────────────────┤
│ {'name': duck, 'count': 2, 'tags': [1, 2, 3], 'active': true}        │
└──────────────────────────────────────────────────────────────────────┘
```

A STRUCT type, as a return type or as the type of a `duckdb.Value`, takes each field from the dict key of the same name, ignoring case. A dict whose keys do not match the fields raises `InvalidInputException: We could not convert the object ... to the desired target type`.

`tuple`

`tuple` converts to `LIST` by default, when it's used to construct a Value object of type STRUCT it will convert to STRUCT instead.

NumPy Objects

`numpy.ndarray` and `numpy.datetime64` are converted by calling `tolist()` and converting the result of that.

A NumPy integer scalar such as `numpy.int64(5)` is read by its value in `values()` and in expressions. As a query parameter, it raises `NotImplementedException: Unable to transform python value of type ...`; pass its Python value, `numpy.int64(5).item()`, instead.

Python DB API

The standard DuckDB Python API provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249 similar to the SQLite Python API.

Connection

To use the module, you must first create a `DuckDBPyConnection` object that represents a connection to a database. This is done through the `duckdb.connect` method, which takes three arguments: `database`, `read_only` and `config`.

The `config` keyword argument can be used to provide a `dict` that contains key-value pairs referencing settings understood by DuckDB.

In-Memory Connection

The special value `:memory:` can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Python process).

Named In-memory Connections The special value `:memory:` can also be postfixed with a name, for example: `:memory:conn3`. When a name is provided, subsequent `duckdb.connect` calls will create a new connection to the same database, sharing the catalogs (views, tables, macros etc.).

Using `:memory:` without a name will always create a new and separate database instance.

Default Connection

The `duckdb` module has a default connection to an unnamed **in-memory database**. Functions such as `duckdb.sql()` and `duckdb.execute()` use the default connection; the first function that needs it creates it. Some module functions differ from connection methods; see Connection Object and Module.

The special value `:default:` and the `duckdb.default_connection()` function return the default connection:

```
import duckdb

duckdb.execute("CREATE TABLE tbl AS SELECT 42 AS a")
con = duckdb.connect(":default:")
con.sql("SELECT * FROM tbl")
# or
duckdb.default_connection().sql("SELECT * FROM tbl")
```

```
┌─────────┐
│    a    │
│  int32  │
├─────────┤
│      42 │
└─────────┘
```

To make another connection the default connection, pass it to `duckdb.set_default_connection()`:

```
import duckdb

con = duckdb.connect()
con.execute("CREATE TABLE answer AS SELECT 42 AS a")
duckdb.set_default_connection(con)
duckdb.sql("SELECT * FROM answer").fetchall()
```

```
[(42,)]
```

File-Based Connection

If the `database` is a file path, a connection to a persistent database is established. If the file does not exist the file will be created (the extension of the file is irrelevant and can be `.db`, `.duckdb` or anything else).

`read_only` Connections If you would like to connect in read-only mode, you can set the `read_only` flag to `True`. If the file does not exist, it is **not** created when connecting in read-only mode. Read-only mode is required if multiple Python processes want to access the same database file at the same time.

Within one Python process, all connections to a database file must use the same configuration, `read_only` included.

```
import duckdb

# to start an in-memory database
con = duckdb.connect(database=":memory:")
# to use a database file (not shared between processes)
con = duckdb.connect(database="my-db.duckdb", read_only=False)
con.close()
# to use a database file (shared between processes)
con = duckdb.connect(database="my-db.duckdb", read_only=True)
con.close()
```

```
# to explicitly get the default connection
con = duckdb.connect(database=":default:")
```

You can open another connection to the same database with the `cursor()` method. A cursor has its own transaction and its own session settings, and cursors can run queries at the same time. A single connection runs one call at a time: a call from another thread while it is busy raises a `ResourceInUseException` instead of waiting, so give each thread its own cursor. Closing a connection closes its cursors. See About `cursor()` for details.

Connections are closed implicitly when they go out of scope or if they are explicitly closed using `close()`. Once the last connection to a database instance is closed, the database instance is closed as well.

Querying

SQL queries can be sent to DuckDB using the `execute()` method of connections. Once a query has been executed, results can be retrieved using the `fetchone`, `fetchmany` and `fetchall` methods on the connection. `fetchone()` returns the next row, or None when no rows are left. `fetchmany(size)` returns a list of at most `size` rows, one by default. `fetchall()` returns all remaining rows. `df()` and `fetchnumpy()` return the remaining rows as a pandas DataFrame or as NumPy arrays. Below are some short examples:

```
import duckdb

con = duckdb.connect()
# create a table
con.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)")
# insert two items into the table
con.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2, 2)")
print(con.rowcount)

# retrieve the items again
con.execute("SELECT * FROM items")
print(con.fetchall())

# retrieve the items one at a time
con.execute("SELECT * FROM items")
print(con.fetchone())
print(con.fetchmany(5))
print(con.fetchone())
```

```
2
[('jeans', Decimal('20.00'), 1), ('hammer', Decimal('42.20'), 2)]
('jeans', Decimal('20.00'), 1)
[('hammer', Decimal('42.20'), 2)]
None
```

The `description` property of the connection object contains the column names as per the standard. The `rowcount` property holds the number of rows that the last `INSERT`, `UPDATE` or `DELETE` changed, counted over all parameter sets of an `executemany()` call. It is `-1` after a query and after a statement with a `RETURNING` clause.

Query Parameters

Pass parameter values as the second argument to `execute()`. Use `?` with a list in placeholder order, `$1` to reuse a list value or `$name` with a dictionary. Pass values separately from the SQL text. The query then runs as a prepared statement, and the values are converted according to the conversion rules.

```
rows = con.execute(
    "SELECT item FROM items WHERE value > $minimum",
    {"minimum": 40},
).fetchall()
print(rows)
```

```
[('hammer',)]
```

Parameter names are not case sensitive: the key `"ABC"` binds `$abc`, so do not pass two keys that differ only in case.

Insert a row with `?` placeholders, and several rows with `executemany()`, which runs the statement once for each list of values:

```
con.execute("INSERT INTO items VALUES (?, ?, ?)", ["laptop", 2000, 1])
con.executemany("INSERT INTO items VALUES (?, ?, ?)", [["chainsaw", 500, 10], ["iphone", 300, 2]])

con.execute("SELECT item FROM items WHERE value > ?", [400])
print(con.fetchall())

[('laptop',), ('chainsaw',)]
```

`$1` stands for the first value of the list, so a value can appear more than once:

```
con.execute("SELECT $1, $1, $2", ["duck", "goose"])
print(con.fetchall())

[('duck', 'duck', 'goose')]
```

> **Warning.** Do *not* use `executemany` to insert large amounts of data into DuckDB. For rows that Python code produces, use the appender; see the data ingestion page for other options.

Open Results

`execute()` starts the statement and returns the connection. A query runs until its first rows are ready, so an error up to that point raises from `execute()`, and an error further into the query raises from the fetch that reaches it. Any other statement, `INSERT ... RETURNING` included, runs to its end inside `execute()`.

The result of a query stays open until a fetch returns no rows, or until `fetchall()`, `df()` or `fetchnumpy()` reads the rest of it. After a partial read with `fetchone()` or `fetchmany()`, these methods return the rows not fetched yet. `con.complete()` runs an open result to its end without reading its rows, and `con.abort()` cancels it, which fails an open transaction when the statement may write. Both are also functions of the `duckdb` module. The next `execute()` replaces an open result. If the replaced result may have side effects then it runs to its end first.

While a result is open, everything that runs a query on the connection raises an error, including reading a relation, `register()` and `create_function()`. Building a relation, the `description` property and `cursor()` still work, so a cursor can run other queries in the meantime.

```
con.execute("SELECT * FROM items")
con.sql("SELECT 42").fetchall()

_duckdb.ResourceInUseException: Resource In Use Error: connection has an unfinished result from execute();
call complete() or abort(), or fetch until a fetch returns no rows, before other work on this connection,
or use a separate connection from cursor()

con.complete()
con.sql("SELECT 42").fetchall()

[(42,)]
```

Arrow and Polars Output

A result of `execute()` has no `fetch_arrow_table()`, `fetch_record_batch()`, `arrow()` or `pl()` method. Its `to_arrow_table()` and `to_arrow_reader()` methods copy the rows and emit a `UserWarning`, and that reader does not stream: its first batch arrives after the whole query ran. For Arrow and Polars output, use a relation instead: `con.sql(query).to_arrow_table()`, `con.sql(query).pl()` or `con.sql(query).to_arrow_reader()`, which streams.

Transactions

`con.begin()`, `con.commit()` and `con.rollback()` start, commit and roll back a transaction, as the BEGIN, COMMIT and ROLLBACK statements do. Inside a transaction, any statement run with `execute()` that fails aborts the transaction, including one with a syntax error or one that names a missing table. After that, `commit()` rolls the transaction back without raising an error, and any other statement raises a `TransactionException` until you roll back:

```
con.begin()
con.execute("INSERT INTO items VALUES ('drill', 80, 1)")
try:
    con.execute("SELECT * FROM missing_table")
except duckdb.CatalogException:
    pass
con.execute("SELECT count(*) FROM items")
```

```
_duckdb.TransactionException: TransactionContext Error: Current transaction is aborted (please ROLLBACK)
```

```
con.rollback()
con.execute("SELECT count(*) FROM items").fetchall()
```

```
[(5,)]
```

Relational API

The Relational API builds a query one step at a time from Python. A relation, a `DuckDBPyRelation`, is a piece of SQL text: the methods of the Transformation and Functions sections return a new relation whose text extends the one they were called on.

Lazy Evaluation

Building a relation runs nothing: DuckDB binds the text once to learn the columns and their types, and holds no data. For example, a relation over a billion rows is built at once:

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("FROM range(1_000_000_000) t(id) SELECT id, id % 7 AS bucket")
print(rel.sql_query())
```

```
FROM range(1_000_000_000) t(id) SELECT id, id % 7 AS bucket
```

`sql_query()` returns exactly the text that runs. The query runs when a method of the Output section reads the relation, and it runs again on every read, so each read sees the current data. Printing a relation, or calling `show()`, also runs it and displays the first rows:

```
import duckdb

duckdb_conn = duckdb.connect()
rel = duckdb_conn.sql("FROM range(1_000_000_000) t(id) SELECT id, id % 7 AS bucket")
rel.filter("bucket = 3").order("id").limit(3).show()
```

id int64	bucket int64
3	3
10	3
17	3

Only a statement that reads, such as a `SELECT`, gives a lazy relation. Any other statement passed to `sql()`, such as `CREATE TABLE` or `INSERT`, runs at the call, and the relation holds what it returned. The `params` list or dict is copied when the relation is created, and its values are converted each time the relation runs. A column is also an attribute of the relation, as in `rel.price`, unless a method has its name: a column called `offset` is reached as `rel["offset"]`, since `rel.offset` is the `offset()` method.

To run a query once and keep its rows, call `materialize()`:

```
import duckdb

duckdb_conn = duckdb.connect()
duckdb_conn.sql("CREATE TABLE items (id INTEGER)")
counted = duckdb_conn.sql("SELECT count(*) AS n FROM items")
held = counted.materialize()
duckdb_conn.sql("INSERT INTO items VALUES (1), (2)")
print(counted.fetchall(), held.fetchall())
```

```
[(2,)] [(0,)]
```

Relation Creation

This section contains the details on how a relation is created. A relation over a query that reads is lazy evaluated.

Name	Description
from_arrow	Create a relation object from an Arrow object.
from_csv_auto	Alias of read_csv().
from_df	Create a relation object from the DataFrame in df.
from_parquet	Alias of read_parquet().
from_query	Alias of sql().
query	Alias of sql().
read_csv	Create a relation object from the CSV file in 'name'.
read_json	Create a relation object from the JSON file in 'name'.
read_parquet	Create a relation object from the Parquet path(s) or file-like object(s) in 'path_or_buffer'.
sql	Run a SQL query. A statement that only reads returns a relation that runs the query each time it is read, parameter values bound at every run. Any other statement runs once, now, and gives a relation over what it returned. Statements before the last one run immediately.
table	Create a relation object for the named table.
table_function	Create a relation object from the named table function with given parameters.
values	Create a relation object from the passed values.
view	Create a relation object for the named view.

from_arrow

```
from_arrow(self: _duckdb.DuckDBPyConnection, arrow_object: object) -> DuckDBPyRelation
```

Create a relation object from an Arrow object.

Parameters

- **arrow_object** : object

  The Arrow object to create a relation from: a table, a record batch, a record batch reader, a dataset, a scanner, a polars DataFrame or an object that exports the Arrow PyCapsule interface.

Example

```
import duckdb
import pyarrow as pa

ids = pa.array([1], type=pa.int8())
texts = pa.array(['a'], type=pa.string())
example_table = pa.table([ids, texts], names=["id", "text"])

duckdb_conn = duckdb.connect()

rel = duckdb_conn.from_arrow(example_table)

rel.show()
```

Result

id int8	text varchar
1	a

---

from_csv_auto

```
from_csv_auto(self: _duckdb.DuckDBPyConnection, *args: object, **kwargs: object) -> DuckDBPyRelation
```

Alias of `read_csv()`.

---

from_df

```
from_df(self: _duckdb.DuckDBPyConnection, df: object) -> DuckDBPyRelation
```

Create a relation object from the DataFrame in df.

Parameters

- **df** : pandas.DataFrame

  A pandas DataFrame to be converted into a DuckDB relation.

Example

```
import duckdb
import pandas as pd

df = pd.DataFrame(data = {'id': [1], "text":["a"]})

duckdb_conn = duckdb.connect()

rel = duckdb_conn.from_df(df)

rel.show()
```

Result

id int64	text varchar
1	a

---

from_parquet

```
from_parquet(self: _duckdb.DuckDBPyConnection, path_or_buffer: object, binary_as_string: bool = False,
**kwargs: Any) -> DuckDBPyRelation
```

Alias of `read_parquet()`.

---

from_query

```
from_query(self: _duckdb.DuckDBPyConnection, query: object, *, alias: str = '', params: object = None) ->
DuckDBPyRelation | None
```

Alias of `sql()`.

---

query

```
query(self: _duckdb.DuckDBPyConnection, query: object, *, alias: str = '', params: object = None) ->
DuckDBPyRelation | None
```

Alias of `sql()`.

---

read_csv

```
read_csv(self: _duckdb.DuckDBPyConnection, *args: object, **kwargs: object) -> DuckDBPyRelation
```

Create a relation object from the CSV file in 'name'.

The options other than `path_or_buffer` are keyword arguments; an unknown one raises `InvalidInputException`. Most are the option of the same name of the SQL `read_csv` function; `sep` and `delimiter` are `delim`, `dtype` is `dtypes`, `na_values` is `null_str`, `skiprows` is `skip`, `quotechar` is `quote`, `escapechar` is `escape`, `date_format` is `dateformat`, `timestamp_format` is `timestampformat`, `lineterminator` is `new_line` and `decimal` is `decimal_separator`.

Aliases: `from_csv_auto`

Parameters

- **path_or_buffer** : str, list[str], file-like object

  Path to the file, a glob pattern, a list of paths or a file-like object such as `io.StringIO` or `io.BytesIO`.

- **header** : bool, int, keyword-only, default: None

  Whether the first line holds the column names; `0` means `True`. Detected when omitted.

- **compression** : str, keyword-only, default: None

  Compression of the files: `'none'`, `'gzip'` or `'zstd'`. Detected from the file extension when omitted.

- **sep** : str, keyword-only, default: None

  Delimiter between the columns, up to 4 bytes; cannot be combined with `delimiter`. Detected when omitted.

- **delimiter** : str, keyword-only, default: None

  Same as `sep`.

- **dtype** : dict[str, str], list[str], keyword-only, default: None

  Column types, as a dict of column name to type or a list of types by position; a type is a type name or a `DuckDBPyType`.

- **na_values** : str, list[str], keyword-only, default: None

  Strings that are read as `NULL`.

- **skiprows** : int, keyword-only, default: None

  Number of lines to skip at the start of each file.

- **quotechar** : str, keyword-only, default: None

  Character used to quote values.

- **escapechar** : str, keyword-only, default: None

  Character that escapes the quote character inside a quoted value.

- **comment** : str, keyword-only, default: None

  Character that starts a comment: a line is read only up to it, and a line that starts with it is skipped.

- **thousands** : str, keyword-only, default: None

  Character that separates the thousands in numbers; it must differ from `decimal`.

- **encoding** : str, keyword-only, default: None

  Only UTF-8 is accepted, written `'utf-8'` or `'utf8'`; any other encoding raises `BinderException`.

- **parallel** : bool, keyword-only, default: None

  Whether to use the parallel CSV reader.

- **date_format** : str, keyword-only, default: None

  Format used to parse dates.

- **timestamp_format** : str, keyword-only, default: None

  Format used to parse timestamps.

- **sample_size** : int, keyword-only, default: None

  Number of lines sampled to detect the dialect and the types.

- **files_to_sniff** : int, keyword-only, default: None

  Number of files sampled to detect the schema when reading several files; `-1` samples all of them.

- **auto_detect** : bool, keyword-only, default: None

  Whether to detect the dialect, the header and the types. Detected when omitted.

- **all_varchar** : bool, keyword-only, default: None

  Read every column as VARCHAR, without detecting the types.

- **normalize_names** : bool, keyword-only, default: None

  Normalize the column names; for example, `My Col` becomes `my_col`.

- **null_padding** : bool, keyword-only, default: None

  Pad a line that lacks columns with `NULL` values on the right.

- **strict_mode** : bool, keyword-only, default: None

  Whether to refuse a file that is not structurally valid CSV; with `False`, the reader tries to read it anyway.

- **names** : list[str], keyword-only, default: None

  Column names, as a list.

- **lineterminator** : str, CSVLineTerminator, keyword-only, default: None

  The line ending: one of `duckdb.CSVLineTerminator.LINE_FEED`, `duckdb.CSVLineTerminator.CARRIAGE_RETURN_LINE_FEED`, `'\\n'` and `'\\r\\n'`, the last two being escape text as written in Python; a string that holds the newline character itself, such as `'\n'`, is refused.

- **columns** : dict[str, str], keyword-only, default: None

  Column names and types, as a dict of column name to type name; the types are then not detected.

- **auto_type_candidates** : list[str], keyword-only, default: None

  Types the reader may choose from when it detects the column types; VARCHAR is always one of them.

- **max_line_size** : int, str, keyword-only, default: None

  Maximum line size, in bytes.

- **ignore_errors** : bool, keyword-only, default: None

  Skip the lines that fail to parse.

- **store_rejects** : bool, keyword-only, default: None

  Skip the lines with errors and store them in the rejects table.

- **rejects_table** : str, keyword-only, default: None

  Name of the temporary table that holds the faulty lines.

- **rejects_scan** : str, keyword-only, default: None

  Name of the temporary table that holds information on the scans that found faulty lines.

- **rejects_limit** : int, keyword-only, default: None

  Maximum number of faulty lines stored for each file; `0` means no limit.

- **force_not_null** : list[str], keyword-only, default: None

  Columns whose values are never read as `NULL`.

- **buffer_size** : int, keyword-only, default: None

  Size of the buffers used to read the files, in bytes.

- **decimal** : str, keyword-only, default: None

  Decimal separator of numbers.

- **allow_quoted_nulls** : bool, keyword-only, default: None

  Whether a quoted value can be read as `NULL`.

- **filename** : bool, keyword-only, default: None

  Add a column `filename` with the path of the file each row came from.

- **hive_partitioning** : bool, keyword-only, default: None

  Interpret the path as a Hive partitioned path.

- **union_by_name** : bool, keyword-only, default: None

  Align the columns of several files by name instead of by position.

- **hive_types** : dict[str, str], keyword-only, default: None

  Types of the Hive partition columns, as a dict of column name to type name.

- **hive_types_autocast** : bool, keyword-only, default: None

  Whether to detect the types of the Hive partition columns.

Example

```
import csv
import duckdb

duckdb_conn = duckdb.connect()

with open('code_example.csv', 'w', newline='') as csvfile:
    fieldnames = ['id', 'text']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'id': '1', 'text': 'a'})

rel = duckdb_conn.read_csv("code_example.csv")

rel.show()
```

Result

id int64	text varchar
1	a

`read_json`

```
read_json(self: _duckdb.DuckDBPyConnection, path_or_buffer: object, **kwargs: Any) -> DuckDBPyRelation
```

Create a relation object from the JSON file in 'name'.

The options other than `path_or_buffer` are keyword arguments; an unknown one raises `TypeError`. Each is the option of the same name of the SQL `read_json` function.

Parameters

- **path_or_buffer** : str, list[str], file-like object

  Path to the file, a glob pattern, a list of paths or a file-like object such as `io.StringIO` or `io.BytesIO`.

- **columns** : dict[str, str], keyword-only, default: None

  Key names and types, as a dict of key name to type name; the schema is then not detected.

- **sample_size** : int, keyword-only, default: None

  Number of objects sampled to detect the types; `-1` samples all of them.

- **maximum_depth** : int, keyword-only, default: None

  Maximum nesting depth to which the types are detected; `-1` detects all levels.

- **records** : str, keyword-only, default: None

  `'true'` to read each object as a row of columns, `'false'` to read each as one value, `'auto'` to detect it.

- **format** : str, keyword-only, default: None

  `'auto'`, `'unstructured'`, `'newline_delimited'` or `'array'`.

- **date_format** : str, keyword-only, default: None

  Format used to parse dates.

- **timestamp_format** : str, keyword-only, default: None

  Format used to parse timestamps.

- **compression** : str, keyword-only, default: None

  Compression of the files: `'none'`, `'gzip'`, `'zstd'` or `'auto_detect'`. Detected from the file extension when omitted.

- **maximum_object_size** : int, keyword-only, default: None

  Maximum size of a JSON object, in bytes.

- **ignore_errors** : bool, keyword-only, default: None

  Read the objects that fail to parse as `NULL` rows; only with `format='newline_delimited'`.

- **convert_strings_to_integers** : bool, keyword-only, default: None

  Whether strings that hold integers are read as numbers.

- **field_appearance_threshold** : float, keyword-only, default: None

  When the fields of an object appear, on average, in a smaller fraction of the sampled objects than this, the object is read as a `MAP`.

- **map_inference_threshold** : int, keyword-only, default: None

  An object with more fields than this is read as a `MAP` instead of a `STRUCT`; `-1` never reads a `MAP`.

- **maximum_sample_files** : int, keyword-only, default: None

  Maximum number of files sampled to detect the schema.

- **filename** : bool, keyword-only, default: None

  Add a column `filename` with the path of the file each row came from.

- **hive_partitioning** : bool, keyword-only, default: None

  Interpret the path as a Hive partitioned path.

- **union_by_name** : bool, keyword-only, default: None

  Align the columns of several files by name instead of by position.

- **hive_types** : dict[str, str], keyword-only, default: None

  Types of the Hive partition columns, as a dict of column name to type name.

- **hive_types_autocast** : bool, keyword-only, default: None

  Whether to detect the types of the Hive partition columns.

Example

```python
import duckdb
import json

with open("code_example.json", mode="w") as f:
    json.dump([{'id': 1, "text":"a"}], f)

duckdb_conn = duckdb.connect()

rel = duckdb_conn.read_json("code_example.json")

rel.show()
```

Result

id int64	text varchar
1	a

`read_parquet`

```
read_parquet(self: _duckdb.DuckDBPyConnection, path_or_buffer: object, binary_as_string: bool = False,
**kwargs: Any) -> DuckDBPyRelation
```

Create a relation object from the Parquet path(s) or file-like object(s) in 'path_or_buffer'.

Each option is the option of the same name of the SQL `read_parquet` function.

Aliases: `from_parquet`

Parameters

- **path_or_buffer** : str, list[str], file-like object

  Path to the Parquet file, a glob pattern, a list of paths or a file object opened in binary mode.

- **binary_as_string** : bool, default: False

  Read binary columns as strings, for files from writers that do not mark their strings as UTF-8.

- **file_row_number** : bool, keyword-only, default: False

  Add a column `file_row_number` with the number of each row within its file.

- **filename** : bool, keyword-only, default: False

  Add a column `filename` with the path of the file each row came from.

- **hive_partitioning** : bool, keyword-only, default: False

  Interpret the path as a Hive partitioned path.

- **union_by_name** : bool, keyword-only, default: False

  Align the columns of several files by name instead of by position.

- **compression** : str, keyword-only, default: None

  Accepted and ignored: a Parquet file records its own compression.

Example

```
import duckdb
import pyarrow as pa
import pyarrow.parquet as pq

ids = pa.array([1], type=pa.int8())
texts = pa.array(['a'], type=pa.string())
example_table = pa.table([ids, texts], names=["id", "text"])

pq.write_table(example_table, "code_example.parquet")

duckdb_conn = duckdb.connect()

rel = duckdb_conn.read_parquet("code_example.parquet")

rel.show()
```

Result

id int8	text varchar
1	a

`sql`

```
sql(self: _duckdb.DuckDBPyConnection, query: object, *, alias: str = '', params: object = None) ->
DuckDBPyRelation | None
```

Run a SQL query. A statement that only reads returns a relation that runs the query each time it is read, parameter values bound at every run. Any other statement runs once, now, and gives a relation over what it returned. Statements before the last one run immediately.

Aliases: `from_query`, `query`

Parameters

- **query** : str

  The SQL text; when it holds several statements, the ones before the last run at the call.

- **alias** : str, keyword-only, default: ''

  Alias of the relation.

- **params** : list, dict, keyword-only, default: None

  Values of the query parameters: a list for `?` and `$1` markers, a dict for `$name` markers. The list or dict is copied at the call, as is any list, tuple or dict inside it; the values in it are converted each time the relation runs, so a `bytearray` changed in place afterwards is read changed.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("SELECT * FROM range(1, 4) tbl(id) WHERE id > $minimum", params={"minimum": 1})

rel.show()
```

Result

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       2 │
│       3 │
└─────────┘
```

---

table

```
table(self: _duckdb.DuckDBPyConnection, table_name: str) -> DuckDBPyRelation
```

Create a relation object for the named table.

Parameters

- **table_name** : str

  Name of the table to create a relation from.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("CREATE TABLE code_example AS SELECT * FROM range(1, 2) tbl(id)")

rel = duckdb_conn.table("code_example")

rel.show()
```

Result

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

---

table_function

```
table_function(self: _duckdb.DuckDBPyConnection, name: str, parameters: object = None) -> DuckDBPyRelation
```

Create a relation object from the named table function with given parameters.

Parameters

- **name** : str

  Name of the table function to call.

- **parameters** : list, default: None

  The arguments to pass to the table function, as a list.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("""
    CREATE MACRO get_record_for(x) AS TABLE
    SELECT x * range AS result FROM range(1, 2)
""")

rel = duckdb_conn.table_function(name="get_record_for", parameters=[1])

rel.show()
```

Result

```
┌─────────┐
│ result  │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

`values`

```
values(self: _duckdb.DuckDBPyConnection, *args: object) -> DuckDBPyRelation
```

Create a relation object from the passed values.

Python values are converted once, at the call, so a later change to the Python objects does not change the relation. Such a relation reads rows held in this process, which `create_view()` refuses.

Parameters

- ***args** : list, tuple, object

  The rows: one list holding the values of one row, several tuples for several rows or the values of one row as separate arguments. A value is a Python value or an Expression.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.values([1, 'a'])

rel.show()
```

Result

```
┌─────────┬─────────┐
│  col0   │  col1   │
│  int32  │ varchar │
├─────────┼─────────┤
│       1 │ a       │
└─────────┴─────────┘
```

`view`

```
view(self: _duckdb.DuckDBPyConnection, view_name: str) -> DuckDBPyRelation
```

Create a relation object for the named view.

Parameters

- **view_name** : str

  Name of the view to create a relation from.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("CREATE VIEW code_example AS SELECT * FROM range(1, 2) tbl(id)")

rel = duckdb_conn.view("code_example")

rel.show()
```

Result

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

Relation Definition Details

This section contains the details on how to inspect a relation.

Name	Description
`alias`	Get the name of the current alias.
`columns`	Return a list containing the names of the columns of the relation.
`describe`	Gives basic statistics (e.g., min, max) and if NULL exists for each column of the relation.
`description`	Return the description of the result.
`dtypes`	Alias of `types`.
`explain`	The plan the engine runs for this relation, as text; under Jupyter the default is shown as HTML.
`preview`	The first `limit` rows drawn as a table, with the footer `__repr__` prints for the relation.
`query`	Run the given SQL query in sql_query on the view named virtual_table_name that refers to the relation object.
`set_alias`	Rename the relation object to new alias.
`shape`	Tuple of # of rows, # of columns in relation.
`show`	Display a summary of the data.
`sql_query`	Get the SQL query that is equivalent to the relation; values given as `params=` stay markers.
`type`	Get the type of the relation.
`types`	Return a list containing the types of the columns of the relation.

alias

Get the name of the current alias.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.set_alias("my_relation").alias
```

Result

```
'my_relation'
```

---

columns

Return a list containing the names of the columns of the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.columns
```

Result

```
['id', 'item', 'category', 'price', 'in_stock']
```

---

describe

```
describe(self) -> DuckDBPyRelation
```

Gives basic statistics (e.g., min, max) and if NULL exists for each column of the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.describe()
```

Result

aggr varchar	id double	item varchar	category varchar	price double	in_stock varchar
count	6.0	6	6	6.0	6
mean	3.5	NULL	NULL	38.5	NULL
stddev	1.8708286933869707	NULL	NULL	33.25507480069771	NULL
min	1.0	drill	clothing	7.0	false
max	6.0	t-shirt	tools	89.0	true
median	3.5	NULL	NULL	31.0	NULL

---

description

Return the description of the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.description
```

Result

```
[('id', INTEGER, None, None, None, None, None), ('item', VARCHAR, None, None, None, None, None),
('category', VARCHAR, None, None, None, None, None), ('price', INTEGER, None, None, None, None, None),
('in_stock', BOOLEAN, None, None, None, None, None)]
```

---

dtypes

Alias of `types`.

---

explain

```
explain(self, type: object = ExplainType.STANDARD, format: str | None = None, *, analyze: bool = False) ->
str
```

The plan the engine runs for this relation, as text; under Jupyter the default is shown as HTML.

With `analyze=True` or with `type` set to `'analyze'` or `duckdb.ExplainType.ANALYZE`, the query runs and the plan shows the rows and the time of each operator, as `EXPLAIN ANALYZE` does.

Parameters

- **type** : str, ExplainType, default: ExplainType.STANDARD

  `'standard'` or `duckdb.ExplainType.STANDARD` for the plan, `'analyze'` or `duckdb.ExplainType.ANALYZE` to run the query and profile it.

- **format** : str, default: None

  The output format, such as `'text'`, `'json'`, `'html'`, `'graphviz'` or `'yaml'`. The default draws the plan as text, and as HTML under Jupyter.

- **analyze** : bool, keyword-only, default: False

  Run the query and profile it, as `type='analyze'` does.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

print(rel.explain())
```

Result

```
┌ Column Data Scan ┐
 ~6 rows
└──────────────────┘
```

`preview`

```
preview(self, limit: int = 10) -> str
```

The first `limit` rows drawn as a table, with the footer `__repr__` prints for the relation.

Parameters

- **limit** : int, default: 10

  The number of rows to draw.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

print(rel.preview(3))
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
3	jacket	clothing	65	true

`query`

```
query(self, virtual_table_name: str, sql_query: str) -> DuckDBPyRelation | None
```

Run the given SQL query in sql_query on the view named virtual_table_name that refers to the relation object.

The name is this query's own: it is a common table expression over the relation, and a catalog entry for the length of the call, so a table function or a query that defines the name itself can bind against it. Nothing of it is left afterwards. A statement that does not only read, which includes a SELECT whose common table expression writes, runs at the call and the relation holds what it returned.

Parameters

- **virtual_table_name** : str

  The name under which `sql_query` reads the relation.

- **sql_query** : str

  The SQL query, a single statement, that reads the relation by the name `virtual_table_name`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.query(virtual_table_name="items", sql_query="SELECT max(price) AS max_price FROM items")
```

Result

```
┌───────────┐
│ max_price │
│   int32   │
├───────────┤
│        89 │
└───────────┘
```

---

set_alias

```
set_alias(self, alias: str) -> DuckDBPyRelation
```

Rename the relation object to new alias.

Parameters

- **alias** : str

  The alias name to assign to the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.set_alias("items").select("items.item, items.price")
```

Result

item varchar	price int32
jeans	20
t-shirt	8
jacket	65
hammer	42
screwdriver	7
drill	89

---

shape

Tuple of # of rows, # of columns in relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.shape
```

Result

```
(6, 5)
```

---

show

```
show(self, *, max_width: int | None = None, max_rows: int | None = None, max_col_width: int | None = None,
null_value: str | None = None, render_mode: object = None) -> None
```

Display a summary of the data.

Parameters

- **max_width** : int, keyword-only, default: None

  Maximum display width for the entire output in characters.

- **max_rows** : int, keyword-only, default: None

  Maximum number of rows to display.

- **max_col_width** : int, keyword-only, default: None

  Maximum number of characters to display per column.

- **null_value** : str, keyword-only, default: None

  String to display in place of NULL values.

- **render_mode** : RenderMode, keyword-only, default: None

  `duckdb.RenderMode.ROWS` to draw a row per line, `duckdb.RenderMode.COLUMNS` to draw a column per line.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.show()
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
3	jacket	clothing	65	true
4	hammer	tools	42	true
5	screwdriver	tools	7	true
6	drill	tools	89	true

`sql_query`

```
sql_query(self) -> str
```

Get the SQL query that is equivalent to the relation; values given as `params=` stay markers.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
```

```
    ) AS items(id, item, category, price, in_stock)
""")

print(rel.sql_query())
```

Result

```
SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
```

---

type

Get the type of the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.type
```

Result

```
'QUERY_RELATION'
```

---

types

Return a list containing the types of the columns of the relation.

Aliases: `dtypes`

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
```

```
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.types
```

Result

```
[INTEGER, VARCHAR, VARCHAR, INTEGER, BOOLEAN]
```

Transformation

This section contains the methods which can be used to chain queries. They build a new query from the relation and run nothing.

Name	Description
`aggregate`	Compute the aggregate aggr_expr by the optional groups group_expr on the relation.
`apply`	Compute the function of a single column or a list of columns by the optional groups on the relation.
`cross`	Create cross/cartesian product of two relational objects.
`distinct`	Retrieve distinct rows from this relation object, or the first row of each distinct value of on.
`drop`	Remove columns.
`except_`	Create the set except of this relation object with another relation object in other_rel.
`filter`	Filter the relation object by the filter in filter_expr.
`group_by`	Begin a grouped aggregation; continue with `.agg(...)`.
`head`	The first n rows.
`intersect`	Create the set intersection of this relation object with another relation object in other_rel.
`join`	Join the relation object with another relation object in other_rel.
`limit`	Only retrieve the first n rows from this relation object, starting at offset.
`map`	Calls the passed function on the relation.
`offset`	Skip n rows.
`order`	Reorder the relation object by order_expr, which is SQL order text or an expression.
`pivot`	Turn the given values of `column` into columns holding `aggregate`; the values are explicit.
`project`	Alias of `select()`.
`rename`	Rename columns, given as old=new.
`sample`	A random subset: `n` rows, or `percent` of them, with `seed` repeating a draw.
`select`	Project the relation object by the projection in project_expr.
`select_dtypes`	Alias of `select_types()`.
`select_types`	Select columns from the relation, by filtering based on type(s).
`sort`	Reorder the relation object by the provided expressions.
`union`	Create the set union of this relation object with another relation object in other_rel.
`union_by_name`	Rows from both relations, matching columns by name; keeps duplicates unless `all` is false.

Name	Description
`unnest`	Expand list columns to one row per element.
`unpivot`	Turn the named columns into rows; `name` and `value` name the two new columns.
`with_columns`	Add columns, replacing any of the same name in place.

aggregate

```
aggregate(self, aggr_expr: object, group_expr: object = '') -> DuckDBPyRelation
```

Compute the aggregate aggr_expr by the optional groups group_expr on the relation.

Parameters

- **aggr_expr** : str, list[Expression]

  The columns and aggregates to compute: SQL text, or a list of Expressions, in which a string is a column name.

- **group_expr** : str, Expression, None, default: ''

  The columns to group by, as SQL text or an Expression. `''` or None groups by every column of `aggr_expr` that is not an aggregate (`GROUP BY ALL`).

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.aggregate("category, max(price) AS max_price").order("category")
```

Result

category varchar	max_price int32
clothing	65
tools	89

---

apply

```
apply(self, function_name: str, function_aggr: str, group_expr: str = '', function_parameter: str = '',
projected_columns: str = '') -> DuckDBPyRelation
```

Compute the function of a single column or a list of columns by the optional groups on the relation.

Parameters

- **function_name** : str

  Name of the function to apply over the relation.

- **function_aggr** : str

  The list of columns to apply the function over.

- **group_expr** : str, default: ''

  Optional SQL expression for grouping.

- **function_parameter** : str, default: ''

  Optional parameters to pass into the function.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.apply("max", "price", group_expr="category", projected_columns="category").order("category")
```

Result

category varchar	max(price) int32
clothing	65
tools	89

`cross`

```
cross(self, other_rel: DuckDBPyRelation) -> DuckDBPyRelation
```

Create cross/cartesian product of two relational objects.

Parameters

- **other_rel** : DuckDBPyRelation

  Another relation to perform a cross product with.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

sizes = duckdb_conn.sql("SELECT * FROM (VALUES ('S'), ('M')) AS sizes(size)")

rel.filter("category = 'clothing'").select("item").cross(sizes).order("item, size")
```

Result

item varchar	size varchar
jacket	M
jacket	S
jeans	M
jeans	S
t-shirt	M
t-shirt	S

`distinct`

```
distinct(self, on: object = None) -> DuckDBPyRelation
```

Retrieve distinct rows from this relation object, or the first row of each distinct value of on.

Parameters

- **on** : str, Expression, list, default: None

  A column name, an Expression or a list of them: keep one row for each distinct value of these columns, as SQL `DISTINCT ON` does. Omitted, only fully duplicate rows are removed.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
```

```
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.select("category").distinct().order("category")
```

Result

category
varchar
clothing
tools

---

drop

```
drop(self, *columns: str) -> DuckDBPyRelation
```

Remove columns.

Parameters

- ***columns** : str

  Names of the columns to remove; a name the relation does not have raises `BinderException`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.drop("id", "in_stock")
```

Result

item	category	price
varchar	varchar	int32
jeans	clothing	20
t-shirt	clothing	8
jacket	clothing	65
hammer	tools	42
screwdriver	tools	7
drill	tools	89

---

except_

```
except_(self, other_rel: DuckDBPyRelation) -> DuckDBPyRelation
```

Create the set except of this relation object with another relation object in other_rel.

The difference is EXCEPT ALL, so duplicates count: a row that appears three times in this relation and once in `other_rel` appears twice in the result.

Parameters

- **other_rel** : DuckDBPyRelation

  The relation to combine with this one; it must have the same number of columns.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

categories = rel.select("category")
expensive = rel.filter("price > 30").select("category")

categories.except_(expensive).order("category")
```

Result

```
┌──────────┐
│ category │
│ varchar  │
├──────────┤
│ clothing │
│ clothing │
│ tools    │
└──────────┘
```

---

filter

```
filter(self, filter_expr: object) -> DuckDBPyRelation
```

Filter the relation object by the filter in filter_expr.

Parameters

- **filter_expr** : str, Expression

  The filter expression to apply over the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.filter("price > 30")
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
3	jacket	clothing	65	true
4	hammer	tools	42	true
6	drill	tools	89	true

---

group_by

```
group_by(self, *columns: object) -> GroupedRelation
```

Begin a grouped aggregation; continue with `.agg(...)`.

`group_by()` returns a grouped relation whose only method is `agg(*aggregates)`. `agg()` reads its arguments as `aggregate()` reads `aggr_expr`: one string is SQL text, which may list several aggregates; among several arguments, a string is a column name and an Expression is an aggregate. It returns a relation whose first columns are the group columns.

Parameters

- ***columns** : str, Expression

  The columns to group by: column names or Expressions.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")
```

```
rel.group_by("category").agg("sum(price) AS total, count(*) AS n").order("category")
```

Result

category varchar	total int128	n int64
clothing	93	3
tools	138	3

---

head

```
head(self, n: int = 5) -> DuckDBPyRelation
```

The first n rows.

Parameters

- **n** : int, default: 5

  The number of rows to keep.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.head(3)
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
3	jacket	clothing	65	true

---

intersect

```
intersect(self, other_rel: DuckDBPyRelation) -> DuckDBPyRelation
```

Create the set intersection of this relation object with another relation object in other_rel.

The intersection is `INTERSECT ALL`, so duplicates count: a row that appears three times in this relation and twice in `other_rel` appears twice in the result.

Parameters

- **other_rel** : DuckDBPyRelation

  The relation to combine with this one; it must have the same number of columns.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

categories = rel.select("category")
expensive = rel.filter("price > 30").select("category")

categories.intersect(expensive).order("category")
```

Result

```
┌──────────┐
│ category │
│ varchar  │
├──────────┤
│ clothing │
│ tools    │
│ tools    │
└──────────┘
```

---

join

```
join(self, other_rel: object, condition: object = None, how: str = 'inner', suffix: str | None = None) ->
DuckDBPyRelation
```

Join the relation object with another relation object in other_rel.

The condition is SQL text, a column name or a list of them for a USING join, an Expression, or a function of the two sides. Types supported are 'inner', 'left', 'right', 'outer', 'full', 'semi', 'anti', 'cross', 'positional', 'natural' and 'asof'. With `suffix`, a column of the right side whose name the left side already has is renamed with the suffix appended.

An `'inner'`, `'left'`, `'right'`, `'outer'`, `'full'`, `'semi'`, `'anti'` or `'asof'` join needs a `condition`, and raises `InvalidInputException` without one; a `'cross'`, `'positional'` or `'natural'` join takes none. The joins section of the FROM clause says what each join type does. A CROSS join is also provided through the `cross()` method.

Parameters

- **other_rel** : DuckDBPyRelation

  The relation to join with the current relation.

- **condition** : str, list[str], Expression, Callable, default: None

  The join condition: SQL text such as `'a.id = b.id'`, a column name or a list of column names for a `USING` join, an Expression or a function that takes the two sides and returns an Expression, where `side['id']` is the column `id` of that side.

- **how** : str, default: 'inner'

  The type of join: `'inner'`, `'left'`, `'right'`, `'outer'`, `'full'`, `'semi'`, `'anti'`, `'cross'`, `'positional'`, `'natural'` or `'asof'`.

- **suffix** : str, default: None

  Rename each column of the right side whose name the left side also has, by appending the suffix; the columns of a `USING` or natural join, which appear once, keep their names.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

left = duckdb_conn.sql("SELECT * FROM (VALUES (1, 'a'), (2, 'b'), (3, 'c')) AS t(id, name)")
right = duckdb_conn.sql("SELECT * FROM (VALUES (1, 'x'), (3, 'z'), (4, 'w')) AS t(id, name)")

print(left.join(right, "id").order("id"))

left.join(right, lambda a, b: a["id"] == b["id"], how="left", suffix="_right").order("id")
```

Result

id int32	name varchar	name varchar
1	a	x
3	c	z

id int32	name varchar	id_right int32	name_right varchar
1	a	1	x
2	b	NULL	NULL
3	c	3	z

`limit`

```
limit(self, n: int, offset: int = 0) -> DuckDBPyRelation
```

Only retrieve the first n rows from this relation object, starting at offset.

To read the first rows into Python, as `fetchone()` and `fetchmany()` did on a relation before DuckDB 2.0, use `rel.limit(1).fetchall()` and `rel.limit(n).fetchall()`. To read all the rows a batch at a time, use `iter_rows()`.

Parameters

- **n** : int

  The maximum number of rows to return.

- **offset** : int, default: 0

  The number of rows to skip before starting to return rows.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.limit(1)
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
1	jeans	clothing	20	true

---

map

```
map(self, map_function: Callable[..., Any], *, schema: object = None) -> DuckDBPyRelation
```

Calls the passed function on the relation.

Parameters

- **map_function** : Callable

  A Python function that takes a DataFrame and returns a transformed DataFrame.

- **schema** : dict, keyword-only, default: None

  The columns of the output relation, as a dict of name to type. Without it, the function is called once on an empty DataFrame when the relation is built, to learn the columns it returns.

Example

```
import duckdb
from pandas import DataFrame

def multiply_by_2(df: DataFrame):
    df["id"] = df["id"] * 2
    return df
```

```
duckdb_conn = duckdb.connect()
rel = duckdb_conn.sql("SELECT range AS id, 'dummy' AS text FROM range(1, 3)")

rel.map(multiply_by_2, schema={"id": int, "text": str})
```

Result

id int64	text varchar
2	dummy
4	dummy

`offset`

```
offset(self, n: int) -> DuckDBPyRelation
```

Skip n rows.

Parameters

- **n** : int

  The number of rows to skip.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.order("id").offset(4)
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
5	screwdriver	tools	7	true
6	drill	tools	89	true

`order`

```
order(self, order_expr: object) -> DuckDBPyRelation
```

Reorder the relation object by order_expr, which is SQL order text or an expression.

Parameters

- **order_expr** : str, Expression

  The ordering of the rows: SQL text such as `'price DESC'`, or an Expression such as `col('price').desc()`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.order("price DESC")
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
6	drill	tools	89	true
3	jacket	clothing	65	true
4	hammer	tools	42	true
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
5	screwdriver	tools	7	true

`pivot`

```
pivot(self, column: object, values: Iterable[object], aggregate: object) -> DuckDBPyRelation
```

Turn the given values of `column` into columns holding `aggregate`; the values are explicit.

It runs a `PIVOT` with the values given; the columns not named in `column` or `aggregate` are the rows of the result.

Parameters

- **column** : str, Expression

  The column whose values become the new columns.

- **values** : Iterable

  The values of `column` that become columns, as Python values.

- **aggregate** : str, Expression

  The aggregate each new column holds, as SQL text or an Expression.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.select("in_stock, category, price").pivot("category", ["clothing", "tools"], "sum(price)").order("in_
stock")
```

Result

in_stock boolean	clothing int128	tools int128
false	8	NULL
true	85	138

---

project

```
project(self, *args: object, groups: str = '') -> DuckDBPyRelation
```

Alias of `select()`.

---

rename

```
rename(self, **columns: str) -> DuckDBPyRelation
```

Rename columns, given as old=new.

Parameters

- ****columns** : str

  The columns to rename, as `old=new`; a name the relation does not have raises `BinderException`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()
```

```
rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.rename(price="unit_price")
```

Result

id int32	item varchar	category varchar	unit_price int32	in_stock boolean
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
3	jacket	clothing	65	true
4	hammer	tools	42	true
5	screwdriver	tools	7	true
6	drill	tools	89	true

`sample`

```
sample(self, n: int | None = None, *, percent: float | None = None, seed: int | None = None, method: str | None = None) -> DuckDBPyRelation
```

A random subset: n rows, or `percent` of them, with `seed` repeating a draw.

See Samples for the sampling methods. A fixed `seed` gives the same sample on every run only when DuckDB runs on one thread.

Parameters

- **n** : int, default: None

  The number of rows to draw; give either n or `percent`.

- **percent** : float, keyword-only, default: None

  The percentage of the rows to draw.

- **seed** : int, keyword-only, default: None

  The seed of the random draw.

- **method** : str, keyword-only, default: None

  `'reservoir'`, `'bernoulli'` or `'system'`; `'reservoir'` for n and `'bernoulli'` for `percent` when omitted.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
```

```
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

duckdb_conn.sql("SET threads = 1")

rel.sample(3, seed=42).order("id")
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
3	jacket	clothing	65	true
4	hammer	tools	42	true
6	drill	tools	89	true

`select`

```
select(self, *args: object, groups: str = '') -> DuckDBPyRelation
```

Project the relation object by the projection in project_expr.

Aliases: `project`

Parameters

- ***args** : str, Expression

  The columns to keep, as strings or Expressions. A single string is SQL text, which may list several columns or expressions; among several arguments, each string names one column.

- **groups** : str, keyword-only, default: ''

  Comma-separated list of columns to include in the GROUP BY.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.select("item, price")
```

Result

item varchar	price int32
jeans	20
t-shirt	8
jacket	65
hammer	42
screwdriver	7
drill	89

`select_dtypes`

```
select_dtypes(self, types: object) -> DuckDBPyRelation
```

Alias of `select_types()`.

`select_types`

```
select_types(self, types: object) -> DuckDBPyRelation
```

Select columns from the relation, by filtering based on type(s).

Aliases: `select_dtypes`

Parameters

- **types** : list

  The types of the columns to keep, as a list of `DuckDBPyType` values or type names such as `'VARCHAR'`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.select_types([duckdb.sqltypes.VARCHAR])
```

Result

item varchar	category varchar
jeans	clothing
t-shirt	clothing
jacket	clothing
hammer	tools
screwdriver	tools
drill	tools

---

sort

```
sort(self, *args: object) -> DuckDBPyRelation
```

Reorder the relation object by the provided expressions.

Parameters

- ***args** : str, Expression

  The expressions to sort by, as SQL text such as `'price DESC'` or as Expressions.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.sort("category", "price DESC")
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
3	jacket	clothing	65	true
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
6	drill	tools	89	true
4	hammer	tools	42	true
5	screwdriver	tools	7	true

---

union

```
union(self, union_rel: DuckDBPyRelation) -> DuckDBPyRelation
```

Create the set union of this relation object with another relation object in other_rel.

The union is `UNION ALL`, so it keeps duplicate rows. To keep only distinct rows, apply `distinct()`.

Parameters

- **union_rel** : DuckDBPyRelation

  The relation to combine with this one; it must have the same number of columns.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.union(rel).count("*")
```

Result

```
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│           12 │
└──────────────┘
```

---

union_by_name

```
union_by_name(self, other: DuckDBPyRelation, *, all: bool = True) -> DuckDBPyRelation
```

Rows from both relations, matching columns by name; keeps duplicates unless `all` is false.

Parameters

- **other** : DuckDBPyRelation

  The relation to combine with this one; a column that only one of them has is `NULL` in the rows of the other.

- **all** : bool, keyword-only, default: True

  Keep duplicate rows (`UNION ALL BY NAME`); with `False`, keep only distinct rows (`UNION BY NAME`).

Example

```
import duckdb

duckdb_conn = duckdb.connect()

first = duckdb_conn.sql("SELECT 1 AS id, 10 AS x")
second = duckdb_conn.sql("SELECT 20 AS y, 2 AS id")

first.union_by_name(second).order("id")
```

Result

id int32	x int32	y int32
1	10	NULL
2	NULL	20

---

unnest

```
unnest(self, *columns: str) -> DuckDBPyRelation
```

Expand list columns to one row per element.

Parameters

- ***columns** : str

  Names of the list columns to expand; each keeps its name and holds one element per row.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("SELECT 1 AS id, [10, 20] AS amounts UNION ALL SELECT 2, [30]")

rel.unnest("amounts").order("id, amounts")
```

Result

id int32	amounts int32
1	10
1	20
2	30

---

unpivot

```
unpivot(self, *columns: str, name: str = 'name', value: str = 'value') -> DuckDBPyRelation
```

Turn the named columns into rows; `name` and `value` name the two new columns.

It runs an `UNPIVOT`; the columns not named stay as they are, repeated for each new row.

Parameters

- ***columns** : str

  Names of the columns to turn into rows.

- **name** : str, keyword-only, default: 'name'

  Name of the new column that holds the names of the unpivoted columns.

- **value** : str, keyword-only, default: 'value'

  Name of the new column that holds their values.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("SELECT * FROM (VALUES ('north', 10, 20), ('south', 30, 40)) AS t(region, q1, q2)")

rel.unpivot("q1", "q2", name="quarter", value="amount").order("region, quarter")
```

Result

region varchar	quarter varchar	amount int32
north	q1	10
north	q2	20
south	q1	30
south	q2	40

`with_columns`

```
with_columns(self, **columns: object) -> DuckDBPyRelation
```

Add columns, replacing any of the same name in place.

Parameters

- ****columns** : Expression, object

  The columns, as `name=value`, where a value is an Expression, a column name or a Python value; a column of the same name is replaced in place, any other is added at the end.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
```

```
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

from duckdb import col, lit

rel.with_columns(price=col("price") * 2, currency=lit("EUR"))
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean	currency varchar
1	jeans	clothing	40	true	EUR
2	t-shirt	clothing	16	false	EUR
3	jacket	clothing	130	true	EUR
4	hammer	tools	84	true	EUR
5	screwdriver	tools	14	true	EUR
6	drill	tools	178	true	EUR

Functions

This section contains the functions which can be applied to a relation, in order to get a (scalar) result. They build a new query and run nothing.

Name	Description
any_value	Returns the first non-null value from a given expression.
arg_max	Finds the row with the maximum value for a value column and returns the value of that row for an argument column.
arg_min	Finds the row with the minimum value for a value column and returns the value of that row for an argument column.
avg	Computes the average of a given expression.
bit_and	Computes the bitwise AND of all bits present in a given expression.
bit_or	Computes the bitwise OR of all bits present in a given expression.
bit_xor	Computes the bitwise XOR of all bits present in a given expression.
bitstring_agg	Computes a bitstring with bits set for each distinct value in a given expression.
bool_and	Computes the logical AND of all values present in a given expression.
bool_or	Computes the logical OR of all values present in a given expression.
count	Computes the number of elements present in a given expression.
cume_dist	Computes the cumulative distribution within the partition.
dense_rank	Computes the dense rank within the partition.
favg	Computes the average of all values present in a given expression using a more accurate floating point summation (Kahan Sum).
first	Returns the first value of a given expression.
first_value	Computes the first value within the group or partition.

Name	Description
`fsum`	Computes the sum of all values present in a given expression using a more accurate floating point summation (Kahan Sum).
`geomean`	Computes the geometric mean over all values present in a given expression.
`histogram`	Computes the histogram over all values present in a given expression.
`lag`	Computes the lag within the partition.
`last`	Returns the last value of a given expression.
`last_value`	Computes the last value within the group or partition.
`lead`	Computes the lead within the partition.
`list`	Returns a list containing all values present in a given expression.
`max`	Returns the maximum value present in a given expression.
`mean`	Alias of `avg()`.
`median`	Computes the median over all values present in a given expression.
`min`	Returns the minimum value present in a given expression.
`mode`	Computes the mode over all values present in a given expression.
`n_tile`	Divides the partition as equally as possible into num_buckets.
`nth_value`	Computes the nth value within the partition.
`percent_rank`	Computes the relative rank within the partition.
`product`	Returns the product of all values present in a given expression.
`quantile`	Alias of `quantile_disc()`.
`quantile_cont`	Computes the interpolated quantile value for a given expression.
`quantile_disc`	Computes the exact quantile value for a given expression.
`rank`	Computes the rank within the partition.
`rank_dense`	Alias of `dense_rank()`.
`row_number`	Computes the row number within the partition.
`std`	Alias of `stddev_samp()`.
`stddev`	Alias of `stddev_samp()`.
`stddev_pop`	Computes the population standard deviation for a given expression.
`stddev_samp`	Computes the sample standard deviation for a given expression.
`string_agg`	Concatenates the values present in a given expression with a separator.
`sum`	Computes the sum of all values present in a given expression.
`unique`	Returns the distinct values in a column.
`value_counts`	Computes the number of elements present in a given expression, also projecting the original expression.
`var`	Alias of `var_samp()`.
`var_pop`	Computes the population variance for a given expression.
`var_samp`	Computes the sample variance for a given expression.
`variance`	Alias of `var_samp()`.

any_value

```
any_value(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Returns the first non-null value from a given expression.

Parameters

- **expression** : str

  The column or expression from which to retrieve any value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.any_value("item")
```

Result

any_value(item)
varchar
jeans

---

arg_max

```
arg_max(self, arg_column: str, value_column: str, groups: str = '', window_spec: str = '', projected_
columns: str = '') -> DuckDBPyRelation
```

Finds the row with the maximum value for a value column and returns the value of that row for an argument column.

Parameters

- **arg_column** : str

  The column whose value is returned, taken from the row where `value_column` is largest.

- **value_column** : str

  The column whose largest value picks the row.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.arg_max("item", "price", groups="category", projected_columns="category").order("category")
```

Result

```
┌──────────┬──────────────────────┐
│ category │ arg_max(item, price) │
│ varchar  │       varchar        │
├──────────┼──────────────────────┤
│ clothing │ jacket               │
│ tools    │ drill                │
└──────────┴──────────────────────┘
```

`arg_min`

```
arg_min(self, arg_column: str, value_column: str, groups: str = '', window_spec: str = '', projected_
columns: str = '') -> DuckDBPyRelation
```

Finds the row with the minimum value for a value column and returns the value of that row for an argument column.

Parameters

- **arg_column** : str

  The column whose value is returned, taken from the row where `value_column` is smallest.

- **value_column** : str

  The column whose smallest value picks the row.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the GROUP BY; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.arg_min("item", "price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	arg_min(item, price) varchar
clothing	t-shirt
tools	screwdriver

---

avg

```
avg(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the average of a given expression.

Aliases: `mean`

Parameters

- **expression** : str

  The column or expression to calculate the average of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the GROUP BY; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.avg("price")
```

Result

```
┌─────────────┐
│ avg(price)  │
│   double    │
├─────────────┤
│        38.5 │
└─────────────┘
```

`bit_and`

```
bit_and(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the bitwise AND of all bits present in a given expression.

Parameters

- **expression** : str

  The column or expression to perform the bitwise AND aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.bit_and("id", groups="category", projected_columns="category").order("category")
```

Result

category varchar	bit_and(id) int32
clothing	0
tools	4

---

bit_or

```
bit_or(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the bitwise OR of all bits present in a given expression.

Parameters

- **expression** : str

  The column or expression to perform the bitwise OR aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
```

```
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.bit_or("id", groups="category", projected_columns="category").order("category")
```

Result

category varchar	bit_or(id) int32
clothing	3
tools	7

`bit_xor`

```
bit_xor(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the bitwise XOR of all bits present in a given expression.

Parameters

- **expression** : str

  The column or expression to perform the bitwise XOR aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.bit_xor("id", groups="category", projected_columns="category").order("category")
```

Result

category varchar	bit_xor(id) int32
clothing	0
tools	7

`bitstring_agg`

```
bitstring_agg(self, expression: str, min: int | None = None, max: int | None = None, groups: str = '',
window_spec: str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Computes a bitstring with bits set for each distinct value in a given expression.

Parameters

- **expression** : str

  The column or expression to aggregate as a bitstring.

- **min** : int, default: None

  The smallest value the bitstring can hold; give both `min` and `max`, or neither.

- **max** : int, default: None

  The largest value the bitstring can hold; give both `min` and `max`, or neither.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.bitstring_agg("id", min=1, max=6, groups="category", projected_columns="category").order("category")
```

Result

category varchar	bitstring_agg(id, 1, 6) bit
clothing	111000
tools	000111

---

bool_and

```
bool_and(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the logical AND of all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to perform the boolean AND aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.bool_and("in_stock", groups="category", projected_columns="category").order("category")
```

Result

category varchar	bool_and(in_stock) boolean
clothing	false
tools	true

---

bool_or

```
bool_or(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the logical OR of all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to perform the boolean OR aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.bool_or("in_stock", groups="category", projected_columns="category").order("category")
```

Result

category varchar	bool_or(in_stock) boolean
clothing	true
tools	true

---

count

```
count(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the number of elements present in a given expression.

Parameters

- **expression** : str

  The column or expression to count the values of; `'*'` counts the rows.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.count("id")
```

Result

```
┌──────────┐
│ count(id) │
│   int64   │
├──────────┤
│         6 │
└──────────┘
```

`cume_dist`

```
cume_dist(self, window_spec: str, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the cumulative distribution within the partition.

Parameters

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.cume_dist(window_spec="OVER (PARTITION BY category ORDER BY price)", projected_columns="category, item,
price").order("category, price")
```

Result

category varchar	item varchar	price int32	cume_dist() OVER (PARTITION BY category ORDER BY price) double
clothing	t-shirt	8	0.3333333333333333
clothing	jeans	20	0.6666666666666666
clothing	jacket	65	1.0
tools	screwdriver	7	0.3333333333333333
tools	hammer	42	0.6666666666666666
tools	drill	89	1.0

---

dense_rank

```
dense_rank(self, window_spec: str, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the dense rank within the partition.

Aliases: rank_dense

Parameters

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
```

```
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.dense_rank(window_spec="OVER (PARTITION BY category ORDER BY price)", projected_columns="category,
item, price").order("category, price")
```

Result

category varchar	item varchar	price int32	dense_rank() OVER (PARTITION BY category ORDER BY price) int64
clothing	t-shirt	8	1
clothing	jeans	20	2
clothing	jacket	65	3
tools	screwdriver	7	1
tools	hammer	42	2
tools	drill	89	3

---

favg

```
favg(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the average of all values present in a given expression using a more accurate floating point summation (Kahan Sum).

Parameters

- **expression** : str

  The column or expression to calculate the average of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
```

```
""")

rel.favg("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	favg(price) double
clothing	31.0
tools	46.0

`first`

```
first(self, expression: str, groups: str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Returns the first value of a given expression.

Parameters

- **expression** : str

  The column or expression from which to retrieve the first value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the GROUP BY.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.first("item", groups="category", projected_columns="category").order("category")
```

Result

category varchar	"first"(item) varchar
clothing	jeans
tools	hammer

`first_value`

```
first_value(self, expression: str, window_spec: str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Computes the first value within the group or partition.

Parameters

- **expression** : str

  The column or expression from which to retrieve the first value within the window.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.first_value("item", window_spec="OVER (PARTITION BY category ORDER BY price)", projected_
columns="category").distinct().order("category")
```

Result

category varchar	first_value(item) OVER (PARTITION BY category ORDER BY price) varchar
clothing tools	t-shirt screwdriver

`fsum`

```
fsum(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the sum of all values present in a given expression using a more accurate floating point summation (Kahan Sum).

Parameters

- **expression** : str

  The column or expression to calculate the sum of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.fsum("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	fsum(price) double
clothing	93.0
tools	138.0

---

geomean

```
geomean(self, expression: str, groups: str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Computes the geometric mean over all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the geometric mean of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.geomean("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	geomean(price) double
clothing	21.827857661222115
tools	29.68787497217981

`histogram`

```
histogram(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the histogram over all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the histogram of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
```

```
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.histogram("in_stock", groups="category", projected_columns="category").order("category")
```

Result

category varchar	histogram(in_stock) map(boolean, ubigint)
clothing tools	{false=1, true=2} {true=3}

`lag`

```
lag(self, expression: str, window_spec: str, offset: int = 1, default_value: str = 'NULL', ignore_nulls:
bool = False, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the lag within the partition.

Parameters

- **expression** : str

  The column or expression to apply the lag function on.

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **offset** : int, default: 1

  The number of rows to lag behind.

- **default_value** : str, default: 'NULL'

  The value, as SQL text, to return when the lag offset goes out of bounds.

- **ignore_nulls** : bool, default: False

  Whether to ignore NULL values when computing the lag.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
```

```
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.lag("item", window_spec="OVER (ORDER BY price)", projected_columns="item, price").order("price")
```

Result

item varchar	price int32	lag(item, 1, NULL) OVER (ORDER BY price) varchar
screwdriver	7	NULL
t-shirt	8	screwdriver
jeans	20	t-shirt
hammer	42	jeans
jacket	65	hammer
drill	89	jacket

`last`

```
last(self, expression: str, groups: str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Returns the last value of a given expression.

Parameters

- **expression** : str

  The column or expression from which to retrieve the last value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.last("item", groups="category", projected_columns="category").order("category")
```

Result

category varchar	"last"(item) varchar
clothing	jacket
tools	drill

---

last_value

```
last_value(self, expression: str, window_spec: str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Computes the last value within the group or partition.

Parameters

- **expression** : str

  The column or expression from which to retrieve the last value within the window.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.last_value("item", window_spec="OVER (PARTITION BY category ORDER BY price ROWS BETWEEN UNBOUNDED
PRECEDING AND UNBOUNDED FOLLOWING)", projected_columns="category").distinct().order("category")
```

Result

category	last_value(item) OVER (PARTITION BY category ORDER BY price ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDE D FOLLOWING)
varchar	varchar
clothing	jacket

```
│ tools    │ drill
└──────────┴──────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

`lead`

```
lead(self, expression: str, window_spec: str, offset: int = 1, default_value: str = 'NULL', ignore_nulls:
bool = False, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the lead within the partition.

Parameters

- **expression** : str

  The column or expression to apply the lead function on.

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **offset** : int, default: 1

  The number of rows to lead ahead.

- **default_value** : str, default: 'NULL'

  The value, as SQL text, to return when the lead offset goes out of bounds.

- **ignore_nulls** : bool, default: False

  Whether to ignore NULL values when computing the lead.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.lead("item", window_spec="OVER (ORDER BY price)", projected_columns="item, price").order("price")
```

Result

```
┌────────────┬─────────┬──────────────────────────────────────────────────┐
│    item    │  price  │ lead(item, 1, NULL) OVER (ORDER BY price)        │
│  varchar   │  int32  │                     varchar                      │
├────────────┼─────────┼──────────────────────────────────────────────────┤
│ screwdriver│       7 │ t-shirt                                          │
```

t-shirt	8	jeans
jeans	20	hammer
hammer	42	jacket
jacket	65	drill
drill	89	NULL

`list`

```
list(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Returns a list containing all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to aggregate values into a list.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.list("item", groups="category", projected_columns="category").order("category")
```

Result

category varchar	list(item) varchar[]
clothing	[jeans, t-shirt, jacket]
tools	[hammer, screwdriver, drill]

---

max

```
max(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Returns the maximum value present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the maximum value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.max("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	max(price) int32
clothing	65
tools	89

---

mean

```
mean(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Alias of `avg()`.

---

median

```
median(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the median over all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the median value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.median("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	median(price) double
clothing	20.0
tools	42.0

---

min

```
min(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Returns the minimum value present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the min value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the GROUP BY; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.min("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	min(price) int32
clothing	8
tools	7

---

mode

```
mode(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the mode over all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the mode (most frequent value) of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the GROUP BY; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

sales = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES ('clothing', 20), ('clothing', 20), ('clothing', 8), ('tools', 42), ('tools', 7),
('tools', 7))
        AS sales(category, price)
""")

sales.mode("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	"mode"(price) int32
clothing	20
tools	7

---

n_tile

```
n_tile(self, window_spec: str, num_buckets: int, projected_columns: str = '') -> DuckDBPyRelation
```

Divides the partition as equally as possible into num_buckets.

Parameters

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **num_buckets** : int

  The number of buckets to divide the rows into.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
```

```
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.n_tile(window_spec="OVER (PARTITION BY category ORDER BY price)", num_buckets=2, projected_
columns="category, item, price").order("category, price")
```

Result

category varchar	item varchar	price int32	ntile(2) OVER (PARTITION BY category ORDER BY price) int64
clothing	t-shirt	8	1
clothing	jeans	20	1
clothing	jacket	65	2
tools	screwdriver	7	1
tools	hammer	42	1
tools	drill	89	2

---

nth_value

```
nth_value(self, expression: str, window_spec: str, offset: int, ignore_nulls: bool = False, projected_
columns: str = '') -> DuckDBPyRelation
```

Computes the nth value within the partition.

Parameters

- **expression** : str

  The column or expression from which to retrieve the nth value within the window.

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **offset** : int

  The position of the value to retrieve within the window (1-based index).

- **ignore_nulls** : bool, default: False

  Whether to ignore NULL values when computing the nth value.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
```

```
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.nth_value("item", window_spec="OVER (PARTITION BY category ORDER BY price)", offset=2, projected_
columns="category, item, price").order("category, price")
```

Result

category varchar	item varchar	price int32	nth_value(item, 2) OVER (PARTITION BY category ORDER BY price) varchar
clothing	t-shirt	8	NULL
clothing	jeans	20	jeans
clothing	jacket	65	jeans
tools	screwdriver	7	NULL
tools	hammer	42	hammer
tools	drill	89	hammer

`percent_rank`

```
percent_rank(self, window_spec: str, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the relative rank within the partition.

Parameters

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.percent_rank(window_spec="OVER (PARTITION BY category ORDER BY price)", projected_columns="category,
item, price").order("category, price")
```

Result

category varchar	item varchar	price int32	percent_rank() OVER (PARTITION BY category ORDER BY price) double
clothing	t-shirt	8	0.0
clothing	jeans	20	0.5
clothing	jacket	65	1.0
tools	screwdriver	7	0.0
tools	hammer	42	0.5
tools	drill	89	1.0

`product`

```
product(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Returns the product of all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the product of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.product("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	product(price) double

```
┌──────────┬──────────────────┐
│ clothing │          10400.0 │
│ tools    │          26166.0 │
└──────────┴──────────────────┘
```

---

quantile

```
quantile(self, expression: str, q: float | builtins.list[float] = 0.5, groups: str = '', window_spec: str =
'', projected_columns: str = '') -> DuckDBPyRelation
```

Alias of `quantile_disc()`.

---

quantile_cont

```
quantile_cont(self, expression: str, q: float | builtins.list[float] = 0.5, groups: str = '', window_spec:
str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Computes the interpolated quantile value for a given expression.

Parameters

- **expression** : str

  The column or expression to compute the continuous quantile for.

- **q** : float, list[float], default: 0.5

  The quantile to compute, such as 0.5 for the median, or a list of quantiles.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.quantile_cont("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	quantile_cont(price, 0.500000) double
clothing	20.0
tools	42.0

---

quantile_disc

```
quantile_disc(self, expression: str, q: float | builtins.list[float] = 0.5, groups: str = '', window_spec:
str = '', projected_columns: str = '') -> DuckDBPyRelation
```

Computes the exact quantile value for a given expression.

Aliases: `quantile`

Parameters

- **expression** : str

  The column or expression to compute the discrete quantile for.

- **q** : float, list[float], default: 0.5

  The quantile to compute, such as 0.5 for the median, or a list of quantiles.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.quantile_disc("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	quantile_disc(price, 0.500000) int32

clothing	20
tools	42

`rank`

```
rank(self, window_spec: str, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the rank within the partition.

Parameters

- **window_spec** : str

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.rank(window_spec="OVER (PARTITION BY category ORDER BY price)", projected_columns="category, item,
price").order("category, price")
```

Result

category varchar	item varchar	price int32	rank() OVER (PARTITION BY category ORDER BY price) int64
clothing	t-shirt	8	1
clothing	jeans	20	2
clothing	jacket	65	3
tools	screwdriver	7	1
tools	hammer	42	2
tools	drill	89	3

---

rank_dense

```
rank_dense(self, window_spec: str, projected_columns: str = '') -> DuckDBPyRelation
```

Alias of dense_rank().

---

row_number

```
row_number(self, window_spec: str, projected_columns: str = '') -> DuckDBPyRelation
```

Computes the row number within the partition.

Parameters

- **window_spec** : str

  Window specification, written as OVER (PARTITION BY ... ORDER BY ...).

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.row_number(window_spec="OVER (PARTITION BY category ORDER BY price)", projected_columns="category,
item, price").order("category, price")
```

Result

category varchar	item varchar	price int32	row_number() OVER (PARTITION BY category ORDER BY price) int64
clothing	t-shirt	8	1
clothing	jeans	20	2
clothing	jacket	65	3
tools	screwdriver	7	1
tools	hammer	42	2
tools	drill	89	3

---

std

```
std(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Alias of `stddev_samp()`.

---

stddev

```
stddev(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Alias of `stddev_samp()`.

---

stddev_pop

```
stddev_pop(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the population standard deviation for a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the standard deviation for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.stddev_pop("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	stddev_pop(price) double
clothing	24.535688292770594
tools	33.595634637057636

---

stddev_samp

```
stddev_samp(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the sample standard deviation for a given expression.

Aliases: `std`, `stddev`

Parameters

- **expression** : str

  The column or expression to calculate the standard deviation for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.stddev_samp("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	stddev_samp(price) double
clothing	30.04995840263344
tools	41.14608122288197

---

string_agg

```
string_agg(self, expression: str, sep: str = ',', groups: str = '', window_spec: str = '', projected_
columns: str = '') -> DuckDBPyRelation
```

Concatenates the values present in a given expression with a separator.

Parameters

- **expression** : str

  The column or expression to concatenate values from.

- **sep** : str, default: ','

  Separator string to use between concatenated values.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.string_agg("item", sep=", ", groups="category", projected_columns="category").order("category")
```

Result

category varchar	string_agg(item, ', ') varchar
clothing tools	jeans, t-shirt, jacket hammer, screwdriver, drill

---

sum

```
sum(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the sum of all values present in a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the sum for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.sum("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	sum(price) int128
clothing	93
tools	138

`unique`

```
unique(self, unique_aggr: str) -> DuckDBPyRelation
```

Returns the distinct values in a column.

Parameters

- **unique_aggr** : str

  The column to get the distinct values for.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.unique("category").order("category")
```

Result

category varchar
clothing tools

---

value_counts

```
value_counts(self, expression: str, groups: str = '') -> DuckDBPyRelation
```

Computes the number of elements present in a given expression, also projecting the original expression.

Parameters

- **expression** : str

  The column or expression to count values from.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the GROUP BY.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.value_counts("in_stock", groups="in_stock").order("in_stock")
```

Result

in_stock boolean	count(in_stock) int64
false	1
true	5

---

var

```
var(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Alias of `var_samp()`.

---

var_pop

```
var_pop(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the population variance for a given expression.

Parameters

- **expression** : str

  The column or expression to calculate the population variance for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.var_pop("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	var_pop(price) double
clothing	602.0
tools	1128.6666666666667

`var_samp`

```
var_samp(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Computes the sample variance for a given expression.

Aliases: `var`, `variance`

Parameters

- **expression** : str

  The column or expression to calculate the sample variance for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `GROUP BY`; cannot be combined with `window_spec`.

- **window_spec** : str, default: ''

  Window specification, written as `OVER (PARTITION BY ... ORDER BY ...)`; cannot be combined with `groups`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.var_samp("price", groups="category", projected_columns="category").order("category")
```

Result

category varchar	var_samp(price) double
clothing	903.0
tools	1693.0

---

variance

```
variance(self, expression: str, groups: str = '', window_spec: str = '', projected_columns: str = '') ->
DuckDBPyRelation
```

Alias of `var_samp()`.

Output

This section contains the methods which read the relation, store it in a table, a view or a file, or write to the table it reads.

Name	Description
`insert`	Inserts the given values into the relation.
`insert_into`	Inserts the relation object into an existing table named table_name.
`update`	Update the given relation with the provided expressions.
`arrow`	Alias of `to_arrow_reader()`.
`close`	Does nothing: a relation holds no result.
`create`	Creates a new table named table_name with the contents of the relation object.
`create_view`	Creates a view named view_name that refers to the relation object.
`df`	Execute and fetch all rows as a pandas DataFrame.
`fetch_arrow_reader`	Deprecated alias of `to_arrow_reader()`.
`fetch_arrow_table`	Deprecated alias of `to_arrow_table()`.
`fetch_record_batch`	Deprecated alias of `to_arrow_reader()`.
`fetchall`	Execute and fetch all rows as a list of tuples.
`fetchdf`	Alias of `df()`.
`fetchnumpy`	Execute and fetch all rows as a Python dict mapping each column to one numpy arrays.
`iter_rows`	Every row, a batch at a time, over one run of the relation.
`materialize`	Execute once and return a relation over the rows, held in client memory.
`pl`	Execute and fetch all rows as a Polars DataFrame.
`tf`	Fetch a result as dict of TensorFlow Tensors.
`to_arrow_reader`	Execute and return an Arrow Record Batch Reader that yields all rows.
`to_arrow_table`	Execute and fetch all rows as an Arrow Table.
`to_csv`	Alias of `write_csv()`.
`to_df`	Alias of `df()`.
`to_dicts`	Every row, as a dict keyed by column name.
`to_parquet`	Alias of `write_parquet()`.
`to_table`	Alias of `create()`.
`to_view`	Alias of `create_view()`.
`torch`	Fetch a result as dict of PyTorch Tensors.
`write_csv`	Write the relation object to a CSV file in 'file_name'.
`write_parquet`	Write the relation object to a Parquet file in 'file_name'.

`insert`

```
insert(self, values: object) -> None
```

Inserts the given values into the relation.

The relation must read a table, as one from `table()` does. The relation runs again on every read, so reading it afterwards shows the new rows.

Parameters

- **values** : tuple, list, object

  The values of one row in the order of the table's columns: a tuple, a list or, for a table of one column, a single value.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("CREATE TABLE items (item VARCHAR, price INTEGER)")

rel = duckdb_conn.table("items")

rel.insert(("jeans", 20))
rel.insert(["hammer", 42])

rel.order("item")
```

Result

item varchar	price int32
hammer	42
jeans	20

`insert_into`

```
insert_into(self, table_name: str) -> None
```

Inserts the relation object into an existing table named table_name.

The name is table, schema.table or catalog.schema.table, and a part may be double quoted; a name of more than three parts needs SQL (INSERT INTO) for now.

Parameters

- **table_name** : str

  The name of the table: `table`, `schema.table` or `catalog.schema.table`; a name of more parts raises `NotImplementedException`. The table must exist, and the relation must have its columns in the table's column order.

Example

```
import duckdb

duckdb_conn = duckdb.connect()
```

```
duckdb_conn.sql("CREATE TABLE items (item VARCHAR, price INTEGER)")

rel = duckdb_conn.values(("jeans", 20), ("hammer", 42))

rel.insert_into("items")

duckdb_conn.table("items").order("item")
```

Result

item varchar	price int32
hammer	42
jeans	20

---

update

```
update(self, set: object, *, condition: object = None) -> None
```

Update the given relation with the provided expressions.

The relation must read a table, as one from `table()` does. The relation runs again on every read, so reading it afterwards shows the updated rows.

Parameters

- **set** : dict

  The new values, as a dict of column name to an Expression or a Python value. A string value raises `TypeError`: write `col('y')` for the column y, or `lit('y')` for the text.

- **condition** : str, Expression, keyword-only, default: None

  Optional condition to filter which rows to update.

Example

```
import duckdb
from duckdb import col, lit

duckdb_conn = duckdb.connect()

duckdb_conn.sql("CREATE TABLE items AS SELECT * FROM (VALUES (1, 'jeans', 20), (2, 'hammer', 42)) AS t(id, item, price)")

rel = duckdb_conn.table("items")

rel.update(set={"item": lit("claw hammer"), "price": col("price") + 5}, condition=col("id") == 2)

rel.order("id")
```

Result

id int32	item varchar	price int32
1	jeans	20
2	claw hammer	47

---

arrow

```
arrow(self, batch_size: int = 131072) -> Any
```

Alias of `to_arrow_reader()`.

> We recommend using `to_arrow_reader()` instead.

---

close

```
close(self) -> None
```

Does nothing: a relation holds no result.

---

create

```
create(self, table_name: str) -> None
```

Creates a new table named table_name with the contents of the relation object.

The name is table, schema.table or catalog.schema.table, and a part may be double quoted; a name of more than three parts needs SQL (CREATE TABLE AS) for now.

Aliases: `to_table`

Parameters

- **table_name** : str

  The name of the table: `table`, `schema.table` or `catalog.schema.table`; a name of more parts raises `NotImplementedException`. There must not be a table of that name already.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.filter("category = 'tools'").create("tools")

duckdb_conn.table("tools")
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
4	hammer	tools	42	true
5	screwdriver	tools	7	true
6	drill	tools	89	true

`create_view`

```
create_view(self, view_name: str, replace: bool = True) -> DuckDBPyRelation
```

Creates a view named view_name that refers to the relation object.

A view stores SQL text, so a relation over rows that DuckDB holds for it cannot become one: a relation from `values()` or `materialize()` or over a statement that ran at the call, such as an `INSERT`, raises `InvalidInputException`. Use `create()` to store such rows in a table, or `con.register(name, rel)` for a temporary view that lasts as long as the connection.

Aliases: `to_view`

Parameters

- **view_name** : str

  The name of the view to be created.

- **replace** : bool, default: True

  If the view should be created with `CREATE OR REPLACE`. When set to `False`, there must not be a view of that name already.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.filter("price > 30").create_view("expensive_items")

duckdb_conn.view("expensive_items")
```

Result

id int32	item varchar	category varchar	price int32	in_stock boolean
3	jacket	clothing	65	true
4	hammer	tools	42	true
6	drill	tools	89	true

---

df

```
df(self, *, date_as_object: bool = False) -> Any
```

Execute and fetch all rows as a pandas DataFrame.

Aliases: `fetchdf`, `to_df`

Parameters

- **date_as_object** : bool, keyword-only, default: False

  If the date columns should be interpreted as Python date objects.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.df()
```

Result

```
   id         item  category  price  in_stock
0   1        jeans  clothing     20      True
1   2      t-shirt  clothing      8     False
2   3       jacket  clothing     65      True
3   4       hammer     tools     42      True
4   5  screwdriver     tools      7      True
5   6        drill     tools     89      True
```

---

fetch_arrow_reader

```
fetch_arrow_reader(self, batch_size: int = 131072) -> Any
```

Deprecated alias of `to_arrow_reader()`.

> **Deprecated.** `fetch_arrow_reader()` is deprecated. Use `to_arrow_reader()` instead.

---

fetch_arrow_table

```
fetch_arrow_table(self, batch_size: int = 131072) -> Any
```

Deprecated alias of `to_arrow_table()`.

> **Deprecated.** `fetch_arrow_table()` is deprecated. Use `to_arrow_table()` instead.

---

fetch_record_batch

```
fetch_record_batch(self, rows_per_batch: int = 131072) -> Any
```

Deprecated alias of `to_arrow_reader()`.

> **Deprecated.** `fetch_record_batch()` is deprecated. Use `to_arrow_reader()` instead.

---

fetchall

```
fetchall(self) -> builtins.list[tuple[Any, ...]]
```

Execute and fetch all rows as a list of tuples.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.limit(1).fetchall()
```

Result

```
[(1, 'jeans', 'clothing', 20, True)]
```

---

fetchdf

```
fetchdf(self, *, date_as_object: bool = False) -> Any
```

Alias of `df()`.

`fetchnumpy`

```
fetchnumpy(self) -> dict[str, Any]
```

Execute and fetch all rows as a Python dict mapping each column to one numpy arrays.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.fetchnumpy()
```

Result

```
{'id': array([1, 2, 3, 4, 5, 6], dtype=int32), 'item': array(['jeans', 't-shirt', 'jacket', 'hammer',
'screwdriver', 'drill'],
      dtype=object), 'category': array(['clothing', 'clothing', 'clothing', 'tools', 'tools', 'tools'],
      dtype=object), 'price': array([20,  8, 65, 42,  7, 89], dtype=int32), 'in_stock': array([ True,
False,  True,  True,  True,  True])}
```

`iter_rows`

```
iter_rows(self, batch_size: int = 1024) -> Iterator[tuple[Any, ...]]
```

Every row, a batch at a time, over one run of the relation.

The rows are read once into the engine's memory, so every batch comes from the same run and the connection is free between batches; the caller holds one batch at a time rather than the whole result.

To read a relation row by row, as a loop over `fetchone()` or `fetchmany()` did before DuckDB 2.0, iterate over `iter_rows()`.

Parameters

- **batch_size** : int, default: 1024

  The number of rows fetched at a time; it must be above zero.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
```

```
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

for row in rel.iter_rows(batch_size=4):
    print(row)
```

Result

```
(1, 'jeans', 'clothing', 20, True)
(2, 't-shirt', 'clothing', 8, False)
(3, 'jacket', 'clothing', 65, True)
(4, 'hammer', 'tools', 42, True)
(5, 'screwdriver', 'tools', 7, True)
(6, 'drill', 'tools', 89, True)
```

`materialize`

```
materialize(self) -> DuckDBPyRelation
```

Execute once and return a relation over the rows, held in client memory.

The returned relation reads the held rows, so it does not see later changes to the data, and `create_view()` refuses it. `materialize()` takes the place of `execute()`, which relations no longer have since DuckDB 2.0.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("CREATE TABLE items AS SELECT range AS id FROM range(3)")

rel = duckdb_conn.table("items")
held = rel.materialize()

duckdb_conn.sql("INSERT INTO items VALUES (3)")

print(len(rel), len(held))
```

Result

```
4 3
```

`pl`

```
pl(self, batch_size: int = 131072, *, lazy: bool = False) -> Any
```

Execute and fetch all rows as a Polars DataFrame.

Needs the `polars` package.

Parameters

- **batch_size** : int, default: 131072

  The number of rows of each Arrow record batch.

- **lazy** : bool, keyword-only, default: False

  Return a polars `LazyFrame` instead of a `DataFrame`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.pl(batch_size=1)
```

Result

```
shape: (6, 5)
```

id	item	category	price	in_stock
i32	str	str	i32	bool
1	jeans	clothing	20	true
2	t-shirt	clothing	8	false
3	jacket	clothing	65	true
4	hammer	tools	42	true
5	screwdriver	tools	7	true
6	drill	tools	89	true

`tf`

```
tf(self) -> dict[str, Any]
```

Fetch a result as dict of TensorFlow Tensors.

Needs the `tensorflow` package.

`to_arrow_reader`

```
to_arrow_reader(self, batch_size: int = 131072) -> Any
```

Execute and return an Arrow Record Batch Reader that yields all rows.

Until the reader is read to the end or closed, the connection refuses other queries with `ResourceInUseException`. A `with` block closes the reader when it ends, also when the loop stops early. The example sets a small `batch_size` only to show several batches. To read the rows a batch at a time, as `fetch_df_chunk()` did before DuckDB 2.0, read the batches of this reader.

Aliases: arrow, fetch_arrow_reader, fetch_record_batch

Parameters

- **batch_size** : int, default: 131072

  The number of rows of each Arrow record batch.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

with rel.select("item, price").to_arrow_reader(batch_size=2) as reader:
    for batch in reader:
        print(batch.to_pydict())
```

Result

```
{'item': ['jeans', 't-shirt'], 'price': [20, 8]}
{'item': ['jacket', 'hammer'], 'price': [65, 42]}
{'item': ['screwdriver', 'drill'], 'price': [7, 89]}
```

---

to_arrow_table

```
to_arrow_table(self, batch_size: int = 131072) -> Any
```

Execute and fetch all rows as an Arrow Table.

Aliases: fetch_arrow_table

Parameters

- **batch_size** : int, default: 131072

  The number of rows of each Arrow record batch.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
```

```
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.to_arrow_table()
```

Result

```
pyarrow.Table
id: int32
item: string
category: string
price: int32
in_stock: bool
----
id: [[1,2,3,4,5,6]]
item: [["jeans","t-shirt","jacket","hammer","screwdriver","drill"]]
category: [["clothing","clothing","clothing","tools","tools","tools"]]
price: [[20,8,65,42,7,89]]
in_stock: [[true,false,true,true,true,true]]
```

---

to_csv

```
to_csv(self, file_name: str, *, sep: str | None = None, na_rep: str | None = None, header: bool | None =
None, quotechar: str | None = None, escapechar: str | None = None, date_format: str | None = None,
timestamp_format: str | None = None, quoting: str | int | None = None, encoding: str | None = None,
compression: str | None = None, overwrite: bool | None = None, per_thread_output: bool | None = None, use_
tmp_file: bool | None = None, partition_by: builtins.list[str] | None = None, write_partition_columns: bool
| None = None) -> None
```

Alias of `write_csv()`.

---

to_df

```
to_df(self, *, date_as_object: bool = False) -> Any
```

Alias of `df()`.

---

to_dicts

```
to_dicts(self) -> builtins.list[dict[str, Any]]
```

Every row, as a dict keyed by column name.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
```

```
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.limit(2).to_dicts()
```

Result

```
[{'id': 1, 'item': 'jeans', 'category': 'clothing', 'price': 20, 'in_stock': True}, {'id': 2, 'item':
't-shirt', 'category': 'clothing', 'price': 8, 'in_stock': False}]
```

---

to_parquet

```
to_parquet(self, file_name: str, *, compression: str | None = None, field_ids: object = None, row_group_
size_bytes: int | str | None = None, row_group_size: int | None = None, overwrite: bool | None = None, per_
thread_output: bool | None = None, use_tmp_file: bool | None = None, partition_by: builtins.list[str] |
None = None, write_partition_columns: bool | None = None, append: bool | None = None, filename_pattern: str
| None = None, file_size_bytes: str | int | None = None) -> None
```

Alias of `write_parquet()`.

---

to_table

```
to_table(self, table_name: str) -> None
```

Alias of `create()`.

---

to_view

```
to_view(self, view_name: str, replace: bool = True) -> DuckDBPyRelation
```

Alias of `create_view()`.

---

torch

```
torch(self) -> dict[str, Any]
```

Fetch a result as dict of PyTorch Tensors.

Needs the `torch` package.

---

write_csv

```
write_csv(self, file_name: str, *, sep: str | None = None, na_rep: str | None = None, header: bool | None = None, quotechar: str | None = None, escapechar: str | None = None, date_format: str | None = None, timestamp_format: str | None = None, quoting: str | int | None = None, encoding: str | None = None, compression: str | None = None, overwrite: bool | None = None, per_thread_output: bool | None = None, use_tmp_file: bool | None = None, partition_by: builtins.list[str] | None = None, write_partition_columns: bool | None = None) -> None
```

Write the relation object to a CSV file in 'file_name'.

Most options are the option of the same name of the COPY ... TO statement; `sep` is `DELIMITER`, `na_rep` is `NULL`, `quotechar` is `QUOTE`, `escapechar` is `ESCAPE`, `quoting` sets `FORCE_QUOTE` for every column and `overwrite` is `OVERWRITE_OR_IGNORE`.

Aliases: `to_csv`

Parameters

- **file_name** : str

  Path of the CSV file to write; a directory with `partition_by` or `per_thread_output`.

- **sep** : str, keyword-only, default: None

  Delimiter between the columns; a comma when omitted.

- **na_rep** : str, keyword-only, default: None

  Text written for `NULL` values; empty when omitted.

- **header** : bool, keyword-only, default: None

  Whether to write a first line with the column names; written when omitted.

- **quotechar** : str, keyword-only, default: None

  Character used to quote values; `"` when omitted.

- **escapechar** : str, keyword-only, default: None

  Character written before a quote character inside a quoted value; `"` when omitted.

- **date_format** : str, keyword-only, default: None

  Format used to write dates.

- **timestamp_format** : str, keyword-only, default: None

  Format used to write timestamps.

- **quoting** : str, int, keyword-only, default: None

  Quote every value: pass `csv.QUOTE_ALL`, `'ALL'` or `'FORCE'`; any other value is refused. Values are quoted only where needed when omitted.

- **encoding** : str, keyword-only, default: None

  Only UTF-8 is accepted, written `'utf-8'` or `'utf8'`.

- **compression** : str, keyword-only, default: None

  Compression of the file: `'none'`, `'gzip'` or `'zstd'`. Chosen from the file extension when omitted.

- **overwrite** : bool, keyword-only, default: None

  Whether an existing file may be replaced. Omitted or `True`, an existing file is replaced; `False` refuses an existing file with `IOException`. With `partition_by`, only `True` writes into a directory that already holds files.

- **per_thread_output** : bool, keyword-only, default: None

  Write one file per thread into the directory `file_name`, rather than one file.

- **use_tmp_file** : bool, keyword-only, default: None

  Write to a temporary file first, so a cancelled write does not leave a broken file in place of an existing one.

- **partition_by** : list[str], keyword-only, default: None

  Columns to partition the output by, written as Hive partitioned directories under `file_name`.

- **write_partition_columns** : bool, keyword-only, default: None

  Whether to also write the partition columns into the files; only has an effect with `partition_by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.write_csv("items.csv")
```

`write_parquet`

```
write_parquet(self, file_name: str, *, compression: str | None = None, field_ids: object = None, row_group_
size_bytes: int | str | None = None, row_group_size: int | None = None, overwrite: bool | None = None, per_
thread_output: bool | None = None, use_tmp_file: bool | None = None, partition_by: builtins.list[str] |
None = None, write_partition_columns: bool | None = None, append: bool | None = None, filename_pattern: str
| None = None, file_size_bytes: str | int | None = None) -> None
```

Write the relation object to a Parquet file in 'file_name'.

Each option is the option of the same name of the `COPY ... TO` statement, except `overwrite`, which is `OVERWRITE_OR_IGNORE`.

Aliases: `to_parquet`

Parameters

- **file_name** : str

  Path of the Parquet file to write; a directory with `partition_by`, `per_thread_output` or `file_size_bytes`.

- **compression** : str, keyword-only, default: None

  Compression codec: `'uncompressed'`, `'snappy'`, `'gzip'`, `'zstd'`, `'brotli'`, `'lz4'` or `'lz4_raw'`; `'snappy'` when omitted.

- **field_ids** : dict, str, keyword-only, default: None

  The Parquet `field_id` of each column, as a dict of column name to id, or `'auto'` to assign them.

- **row_group_size_bytes** : int, str, keyword-only, default: None

  Target size of each row group, as a number of bytes or a text such as `'2MB'`.

- **row_group_size** : int, keyword-only, default: None

  Target number of rows of each row group; 122880 when omitted.

- **overwrite** : bool, keyword-only, default: None

  Whether an existing file may be replaced. Omitted or `True`, an existing file is replaced; `False` refuses an existing file with `IOException`. With `partition_by`, `True` writes into a directory that already holds files, as `append=True` does.

- **per_thread_output** : bool, keyword-only, default: None

  Write one file per thread into the directory `file_name`, rather than one file.

- **use_tmp_file** : bool, keyword-only, default: None

  Write to a temporary file first, so a cancelled write does not leave a broken file in place of an existing one.

- **partition_by** : list[str], keyword-only, default: None

  Columns to partition the output by, written as Hive partitioned directories under `file_name`.

- **write_partition_columns** : bool, keyword-only, default: None

  Whether to also write the partition columns into the files; only has an effect with `partition_by`.

- **append** : bool, keyword-only, default: None

  When a generated file name already exists, choose another name instead of overwriting the file; only has an effect when several files are written. Combined with `overwrite`, it raises `BinderException`.

- **filename_pattern** : str, keyword-only, default: None

  Pattern of the names of the files written, where `{i}` is replaced by an index and `{uuid}` by a UUID; only has an effect when several files are written.

- **file_size_bytes** : int, str, keyword-only, default: None

  Start a new file in the directory `file_name` when a file grows past this size, given as a number of bytes or a text such as `'1MB'`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
    SELECT *
    FROM (VALUES
        (1, 'jeans', 'clothing', 20, TRUE),
        (2, 't-shirt', 'clothing', 8, FALSE),
        (3, 'jacket', 'clothing', 65, TRUE),
        (4, 'hammer', 'tools', 42, TRUE),
        (5, 'screwdriver', 'tools', 7, TRUE),
        (6, 'drill', 'tools', 89, TRUE)
    ) AS items(id, item, category, price, in_stock)
""")

rel.write_parquet("items.parquet")
```

Python Function API

You can create a DuckDB user-defined function (UDF) from a Python function so it can be used in SQL queries. Similarly to regular functions, they need to have a name, a return type and parameter types.

Creating Functions

To register a Python UDF, use the `create_function` method from a DuckDB connection:

```
import duckdb

def add_one(x: int) -> int:
    return x + 1

con = duckdb.connect()
con.create_function("add_one", add_one)
print(con.sql("SELECT add_one(41)").fetchall())
```

```
[(42,)]
```

UDFs need NumPy. Without the numpy package, `create_function` raises `InvalidInputException: 'numpy' is required for this operation, but it wasn't installed.`

The signature is `create_function(name, function, parameters=None, return_type=None, *, type=..., null_handling=..., exception_handling=..., side_effects=False)`. The last four parameters are keyword only, and a value they do not accept raises a `TypeError`:

1. `name` A string representing the unique name of the UDF within the connection catalog.
2. `function` The Python function you wish to register as a UDF.
3. `parameters` Scalar functions can operate on one or more columns. This parameter takes a list of column types used as input.
4. `return_type` Scalar functions return one element per row. This parameter specifies the return type of the function.
5. `type` (optional): DuckDB supports both native Python types and PyArrow Arrays. By default, `type = 'native'` is assumed, but you can specify `type = 'arrow'` to use PyArrow Arrays. In general, using an Arrow UDF will be much more efficient than native because it will be able to operate in batches.
6. `null_handling` (optional): By default, `NULL` values are automatically handled as `NULL`-in `NULL`-out. Users can specify a desired behavior for `NULL` values by setting `null_handling = 'special'`.
7. `exception_handling` (optional): By default, when an exception is thrown from the Python function, the query fails with an error that names it. Users can disable this behavior, and instead return `NULL`, by setting this parameter to `'return_null'`.
8. `side_effects` (optional): By default, functions are expected to produce the same result for the same input. If the result of a function is impacted by any type of randomness, `side_effects` must be set to `True`.

A type given as a string, such as `"INTEGER"` or the name of a type created with `CREATE TYPE`, is looked up on the connection that registers the function. While an `execute()` result of the connection is still open, `create_function` raises a `ResourceInUseException`.

To unregister a UDF, you can call the `remove_function` method with the UDF name:

```
con.remove_function("add_one")
```

Using the Connection from a Function

A UDF cannot use the connection that runs its query: such a call raises a `ResourceInUseException`, and the query fails with `InvalidInputException: Python exception occurred while executing the UDF: ResourceInUseException: Resource In Use Error: code running inside a query cannot use that query's connection; use a separate connection from cursor()`. A cursor of that connection works:

```
def lookup(x: int) -> int:
    cur = con.cursor()
    try:
        return cur.execute("SELECT 41 + ?", [x]).fetchone()[0]
    finally:
        cur.close()

con.create_function("lookup", lookup)
print(con.sql("SELECT lookup(1)").fetchall())
```

```
[(42,)]
```

Using Partial Functions

DuckDB UDFs can also be created with Python partial functions. The arguments the partial function fixes are not parameters of the UDF:

```
from functools import partial
import duckdb

def add(x: int, y: int) -> int:
    return x + y

with duckdb.connect() as con:
    con.create_function("add_ten", partial(add, 10))
    print(con.sql("SELECT add_ten(5)").fetchall())
```

```
[(15,)]
```

Type Annotation

When the function has type annotation it's often possible to leave out all of the optional parameters. Using `DuckDBPyType` we can implicitly convert many known types to DuckDB's type system. For example:

```
import duckdb

def my_function(x: int) -> str:
    return x

duckdb.create_function("my_func", my_function)
print(duckdb.sql("SELECT my_func(42)"))
```

```
┌─────────────┐
│ my_func(42) │
│   varchar   │
├─────────────┤
│ 42          │
└─────────────┘
```

When `parameters` is left out, the parameter types come from the annotations, and a parameter without an annotation accepts any type. The return type must be annotated or given as `return_type`; otherwise `create_function` raises `InvalidInputException: Could not infer the return type, please set it explicitly.`

NULL Handling

By default when functions receive a `NULL` value, this instantly returns `NULL`, as part of the default `NULL`-handling. When this is not desired, you need to explicitly set this parameter to `"special"`.

```
import duckdb
from duckdb.sqltypes import BIGINT

def dont_intercept_null(x):
    return 5

duckdb.create_function("dont_intercept", dont_intercept_null, [BIGINT], BIGINT)
res = duckdb.sql("SELECT dont_intercept(NULL)").fetchall()
print(res)
```

```
[(None,)]
```

With `null_handling="special"`:

```
import duckdb
from duckdb.sqltypes import BIGINT
```

```
def dont_intercept_null(x):
    return 5

duckdb.create_function("dont_intercept_special", dont_intercept_null, [BIGINT], BIGINT, null_
handling="special")
res = duckdb.sql("SELECT dont_intercept_special(NULL)").fetchall()
print(res)
```

```
[(5,)]
```

> Always use `null_handling="special"` when the function can return NULL.

```
import duckdb
from duckdb.sqltypes import VARCHAR


def return_str_or_none(x: str) -> str | None:
    if not x:
        return None

    return x

duckdb.create_function(
    "return_str_or_none",
    return_str_or_none,
    [VARCHAR],
    VARCHAR,
    null_handling="special"
)
res = duckdb.sql("SELECT return_str_or_none('')").fetchall()
print(res)
```

```
[(None,)]
```

Exception Handling

By default, when an exception is thrown from the Python function, the query fails with a `duckdb.InvalidInputException` that names the Python exception. If you want to disable this behavior, and instead return `NULL`, you'll need to set this parameter to `"return_null"`.

```
import duckdb
from duckdb.sqltypes import BIGINT

def will_throw():
    raise ValueError("ERROR")

duckdb.create_function("throws", will_throw, [], BIGINT)
try:
    res = duckdb.sql("SELECT throws()").fetchall()
except duckdb.InvalidInputException as e:
    print(e)

duckdb.create_function("doesnt_throw", will_throw, [], BIGINT, exception_handling="return_null")
res = duckdb.sql("SELECT doesnt_throw()").fetchall()
print(res)
```

```
Invalid Input Error: Python exception occurred while executing the UDF: ValueError: ERROR
[(None,)]
```

`"return_null"` turns an `Exception` other than `MemoryError` into `NULL`. A `MemoryError` still fails the query, and so does an exception that does not derive from `Exception`, such as `KeyboardInterrupt`.

Side Effects

By default DuckDB will assume the created function is a *pure* function, meaning it will produce the same output when given the same input. If your function does not follow that rule, for example when your function makes use of randomness, then you will need to mark this function as having `side_effects`.

For example, this function will produce a new count for every invocation.

```
def count() -> int:
    old = count.counter;
    count.counter += 1
    return old

count.counter = 0
```

If we create this function without marking it as having side effects, the result will be the following:

```
con = duckdb.connect()
con.create_function("my_counter", count, side_effects=False)
res = con.sql("SELECT my_counter() FROM range(10)").fetchall()
print(res)
```

```
[(0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,)]
```

Which is obviously not the desired result, when we add `side_effects=True`, the result is as we would expect:

```
con.remove_function("my_counter")
count.counter = 0
con.create_function("my_counter", count, side_effects=True)
res = con.sql("SELECT my_counter() FROM range(10)").fetchall()
print(res)
```

```
[(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,)]
```

Python Function Types

Currently, two function types are supported, `native` (default) and `arrow`.

Arrow

If the function is expected to receive arrow arrays, set the `type` parameter to `'arrow'`.

This will let the system know to provide each argument as a `pyarrow.ChunkedArray` of at most 2048 rows to the function, and also expect an array of the same amount of rows to be returned from the function.

In general, using an Arrow UDF will be much more efficient than native because it will be able to operate in batches.

```
import duckdb
import pyarrow as pa
from duckdb.sqltypes import VARCHAR
from pyarrow import compute as pc


def mirror(strings: pa.Array, sep: pa.Array) -> pa.Array:
    assert isinstance(strings, pa.ChunkedArray)
    assert isinstance(sep, pa.ChunkedArray)
    return pc.binary_join_element_wise(strings, pc.ascii_reverse(strings), sep)


duckdb.create_function(
    "mirror",
    mirror,
    [VARCHAR, VARCHAR],
```

```
    return_type=VARCHAR,
    type="arrow",
)

duckdb.sql(
    "CREATE OR REPLACE TABLE strings AS SELECT 'hello' AS str UNION ALL SELECT 'world' AS str;"
)
print(duckdb.sql("SELECT mirror(str, '|') FROM strings;").fetchall())
```

```
[('hello|olleh',), ('world|dlrow',)]
```

Native

When the function type is set to `native` the function will be provided with a single tuple at a time, and expect only a single value to be returned. This can be useful to interact with Python libraries that don't operate on Arrow, such as `faker`:

```
import duckdb

from duckdb.sqltypes import DATE
from faker import Faker

def random_date():
    fake = Faker()
    return fake.date_between()

duckdb.create_function(
    "random_date",
    random_date,
    parameters=[],
    return_type=DATE,
    type="native",
    side_effects=True,
)
res = duckdb.sql("SELECT random_date()").fetchall()
print(res)
```

```
[(datetime.date(2019, 5, 15),)]
```

Types API

The `DuckDBPyType` class, `duckdb.sqltypes.DuckDBPyType`, represents one of the data types. Use `DuckDBPyType` to describe SQL types when declaring function parameters, return types or casts. Python types such as `int` are accepted too: `int` describes `BIGINT`. Converting a data value such as 5 follows different rules; see Python Object Conversion.

Converting from Other Types

Wherever a `DuckDBPyType` is expected, any of the options listed below can be given instead.

Python Built-Ins

The table below shows the mapping of Python Built-in types to DuckDB type.

Built-in types	DuckDB type
bool	BOOLEAN
bytearray	BLOB

Built-in types	DuckDB type
bytes	BLOB
float	DOUBLE
int	BIGINT
str	VARCHAR

NumPy Scalar Types

The table below shows the mapping of NumPy scalar types to DuckDB type. A NumPy dtype object does not convert: `DuckDBPyType(numpy.dtype("int32"))` raises a `NotImplementedException`.

Type	DuckDB type
numpy.bool_	BOOLEAN
numpy.float32	FLOAT
numpy.float64	DOUBLE
numpy.int16	SMALLINT
numpy.int32	INTEGER
numpy.int64	BIGINT
numpy.int8	TINYINT
numpy.uint16	USMALLINT
numpy.uint32	UINTEGER
numpy.uint64	UBIGINT
numpy.uint8	UTINYINT

Type Names

A string is read as a SQL type name, such as `"INTEGER"`, `"INT[]"`, `"STRUCT(a INT)"` or `"MAP(INT, VARCHAR)"`:

```
import duckdb
from duckdb.sqltypes import DuckDBPyType

print(DuckDBPyType("MAP(INT, VARCHAR)"))
```

```
MAP(INTEGER, VARCHAR)
```

A type created with `CREATE TYPE` exists only in its database, so name the connection: `DuckDBPyType("mood", connection=con)` or `con.type("mood")`. Without a connection, the name is looked up on the default connection.

```
con = duckdb.connect()
con.execute("CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')")
print(con.type("mood"))
```

```
ENUM('sad', 'ok', 'happy')
```

Nested Types

`list[child_type]` `list` type objects map to a `LIST` type of the child type, which can itself be nested.

```
import duckdb.sqltypes

print(duckdb.sqltypes.DuckDBPyType(list[int]))

BIGINT[]
```

`dict[key_type, value_type]` `dict` type objects map to a MAP type of the key type and the value type.

```
import duckdb.sqltypes

print(duckdb.sqltypes.DuckDBPyType(dict[str, int]))

MAP(VARCHAR, BIGINT)
```

`{'a': field_one, 'b': field_two, ..., 'n': field_n}` `dict` objects map to a STRUCT composed of the keys and values of the dict.

```
import duckdb.sqltypes

print(duckdb.sqltypes.DuckDBPyType({'a': str, 'b': int}))

STRUCT(a VARCHAR, b BIGINT)
```

`Union[type_1, ... type_n]` `typing.Union` objects, and unions written as `type_1 | type_2`, map to a UNION type of the provided types.

```
import duckdb.sqltypes
from typing import Union

print(duckdb.sqltypes.DuckDBPyType(Union[int, str, bool, bytearray]))
print(duckdb.sqltypes.DuckDBPyType(int | str))

UNION(u1 BIGINT, u2 VARCHAR, u3 BOOLEAN, u4 BLOB)
UNION(u1 BIGINT, u2 VARCHAR)
```

Creation Functions

For the built-in types, you can use the constants defined in `duckdb.sqltypes`:

DuckDB type
BIGINT
BIT
BLOB
BOOLEAN
DATE
DOUBLE
FLOAT
HUGEINT
INTEGER
INTERVAL
SMALLINT
SQLNULL

DuckDB type
TIME_NS
TIME_TZ
TIME
TIMESTAMP_MS
TIMESTAMP_NS
TIMESTAMP_S
TIMESTAMP_TZ
TIMESTAMP
TINYINT
UBIGINT
UHUGEINT
UINTEGER
USMALLINT
UTINYINT
UUID
VARCHAR
VARIANT

For the complex types there are methods available on the `DuckDBPyConnection` object or the `duckdb` module. Anywhere a `DuckDBPyType` is accepted, we will also accept one of the type objects that can implicitly convert to a `DuckDBPyType`.

```
from duckdb.sqltypes import INTEGER, VARCHAR

print(duckdb.list_type(INTEGER))
print(duckdb.array_type(INTEGER, 3))
print(duckdb.struct_type({"a": INTEGER, "b": VARCHAR}))
print(duckdb.map_type(VARCHAR, INTEGER))
print(duckdb.decimal_type(10, 2))
```

```
INTEGER[]
INTEGER[3]
STRUCT(a INTEGER, b VARCHAR)
MAP(VARCHAR, INTEGER)
DECIMAL(10,2)
```

`list_type` Parameters:

- `type: DuckDBPyType`

`array_type` Parameters:

- `type: DuckDBPyType`
- `size: int`

`struct_type|row_type` Parameters:

- `fields: Union[list[DuckDBPyType], dict[str, DuckDBPyType]]`

A list names the fields `v1`, `v2` and so on. An empty dict or list gives an empty `STRUCT`.

`map_type` Parameters:

- `key: DuckDBPyType`
- `value: DuckDBPyType`

`decimal_type` Parameters:

- `width: int`
- `scale: int`

A `DECIMAL` has a width of 1 to 38 and a scale of 0 to the width.

`union_type` Parameters:

- `members: Union[list[DuckDBPyType], dict[str, DuckDBPyType]]`

`string_type` Parameters:

- `collation: str`, by default `""`

`enum_type` `enum_type` is not implemented and raises a `NotImplementedException`. Create an `ENUM` type with SQL and get it with `con.type()`, as shown under Type Names.

Inspecting Types

A `DuckDBPyType` has an `id`, such as `'struct'`. A nested type, an `ENUM` or a `DECIMAL` also has `children`, a list of pairs of a name and a type or a value. The field of a `STRUCT` type is also available by attribute or by key:

```
point = duckdb.struct_type({"x": INTEGER, "y": VARCHAR})
print(point.id)
print(point.children)
print(point.x, point["y"])
```

```
struct
[('x', INTEGER), ('y', VARCHAR)]
INTEGER VARCHAR
```

Expression API

The `Expression` class represents an instance of an expression.

Why Would I Use the Expression API?

Use expressions to build filters and computed columns in Python. They are useful when column names or conditions depend on your code.

```
import duckdb
import pandas as pd
from duckdb import col

sales = pd.DataFrame({"item": ["jeans", "hammer"], "price": [20, 42]})
price_column = "price"
rel = duckdb.from_df(sales)
print(rel.filter(col(price_column) > 30).fetchall())
```

```
[('hammer', 42)]
```

An expression is passed to a method of a relation, such as `select`, `filter`, `aggregate` or `sort`. `str()` of an expression returns its SQL text, with the identifiers quoted and every operation in parentheses:

```
print((col("a") > 1) & col("b"))
```

```
(("a" > 1) AND "b")
```

The `duckdb` module has a short helper for each kind of expression, and this page uses them throughout. The constructors of the earlier expression API still work and build the same expressions, but new code should use the helpers:

Helper	Older constructor
`col(name)`	`ColumnExpression(name)`
`lit(value)`	`ConstantExpression(value)`
`when(condition).then(value)`	`CaseExpression(condition, value)`
`fn(name, *args)`	`FunctionExpression(name, *args)`
`star(exclude=...)`	`StarExpression(exclude=...)`
`sql_expr(text)`	`SQLExpression(text)`
`coalesce(*values)`	`CoalesceOperator(*values)`
a Python function, such as `lambda x: x * 2`	`LambdaExpression(lhs, rhs)`
`.is_null()`, `.is_not_null()`	`.isnull()`, `.isnotnull()`

Values and Operators

Python operators combine expressions with each other and with Python values. The arithmetic operators are `+`, `-`, `*`, `/`, `//`, `%` and `**`; the comparisons are `==`, `!=`, `<`, `<=`, `>` and `>=`; `&`, `|` and `~` are AND, OR and NOT. A Python value in an operation is a constant. This holds for a string too: `col("s") == "x"` compares the column `s` with the text `x`. Use `col("x")` to compare with the column `x`, and `lit("x")` to make a constant explicit:

```
rel = duckdb.sql("SELECT * FROM (VALUES ('x', 'y'), ('y', 'x')) t(s, x)")
print(rel.filter(col("s") == col("x")).fetchall())
print(rel.filter(col("s") == duckdb.lit("x")).fetchall())
```

```
[]
[('x', 'y')]
```

When a plain string is also the name of a column of the relation, as in `rel.filter(col("s") == "x")`, DuckDB uses the string as a value and warns: `UserWarning: 'x' is used as a string value, and it is also a column name; use col('x') for the column, or lit('x') to keep the value and silence this warning.`

An expression has no truth value: `bool()` of an expression, as in `if col("a") == 1:`, raises `TypeError: an expression has no truth value; combine with & | ~, and test with .is_null()`. An operand that is neither an expression nor a value also raises a `TypeError`.

Below is a list of currently supported expressions that can be created through the API.

Column Expression

This expression references a column by name, with `col(name)`.

```
import duckdb
import pandas as pd
```

```
df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})
```

Selecting a single column:

```
a_column = duckdb.col('a')
duckdb.from_df(df).select(a_column).show()
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│       1 │
│       2 │
│       3 │
│       4 │
└─────────┘
```

Selecting multiple columns:

```
col_list = [
        duckdb.col('a') * 10,
        duckdb.col('b').is_null(),
        duckdb.col('c') + 5
    ]
duckdb.from_df(df).select(*col_list).show()
```

```
┌──────────┬──────────────┬─────────┐
│ (a * 10) │ (b IS NULL)  │ (c + 5) │
│  int64   │   boolean    │  int64  │
├──────────┼──────────────┼─────────┤
│       10 │ false        │      47 │
│       20 │ true         │      26 │
│       30 │ false        │      18 │
│       40 │ false        │      19 │
└──────────┴──────────────┴─────────┘
```

A name is never split at a dot: `col("st.a")` is the column named `st.a`. For a qualified column or a field of a STRUCT, give the parts, as in `col("st", "a")`, or index the expression, as in `col("st")["a"]`. Indexing with `[]` also takes an entry of a map by key, or an element of a list by position as in SQL: the first element is at position 1, a negative position counts from the end, and position 0 or a position out of range gives NULL.

```
rel = duckdb.sql("SELECT {'a': 1} AS st, 2 AS \"st.a\"")
print(rel.select(duckdb.col("st.a"), duckdb.col("st", "a"), col("st")["a"]).fetchall())
```

```
[(2, 1, 1)]
```

Star Expression

This expression selects all columns of the input source.

Optionally it's possible to provide an `exclude` list to filter out columns of the table. This `exclude` list holds column names. To leave out a column of one side of a join, such as b of `t`, give it by its parts with the older constructor, as in `StarExpression(exclude=[col("t", "b")])`; `star()` takes plain names only.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
```

```
    'c': [42, 21, 13, 14]
})

all_but_b = duckdb.star(exclude = ['b'])
duckdb.from_df(df).select(all_but_b).show()
```

a	c
int64	int64
1	42
2	21
3	13
4	14

`star(exclude=..., rename=...)` also renames columns, as in `star(exclude="b", rename={"c": "total"})`.

Constant Expression

This expression contains a single value, with `lit(value)`. Operators, `fn()` and the case builder take Python values directly, so `lit()` is needed only where a value must be an expression of its own, as in a select list. Values convert as described in Conversion between DuckDB and Python; a `dict` with string keys is a STRUCT, and a `dict` with other keys is a MAP.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

const = duckdb.lit('hello')
duckdb.from_df(df).select(const).show()
```

'hello'
varchar
hello
hello
hello
hello

Case Expression

This expression is a CASE expression. `when(condition).then(value)` starts it, each further `.when(condition).then(value)` adds a branch, tried in order, and `.otherwise(value)` gives the value where no branch matches. `.end()` closes it without an `otherwise`, which leaves NULL there.

```
import duckdb
import pandas as pd
from duckdb import col, when

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})
```

```
size = (
    when(col("c") > 40).then("large")
    .when(col("c") > 20).then("medium")
    .otherwise("small")
    .alias("size")
)
duckdb.from_df(df).select(col("c"), size).show()
```

```
┌───────┬─────────┐
│   c   │  size   │
│ int64 │ varchar │
├───────┼─────────┤
│    42 │ large   │
│    21 │ medium  │
│    13 │ small   │
│    14 │ small   │
└───────┴─────────┘
```

The older constructor builds the same expression: `CaseExpression(condition, value)` takes the first branch, `.when(condition, value)` adds one and `.otherwise(value)` closes it:

```
from duckdb import CaseExpression

older = CaseExpression(col("c") > 40, "large").when(col("c") > 20, "medium").otherwise("small")
print(older)
```

```
(CASE WHEN ("c" > 40) THEN 'large' WHEN ("c" > 20) THEN 'medium' ELSE 'small' END)
```

Function Expression

This expression contains a function call. It is built from the function name and any number of arguments, with `fn(name, *args)`. An argument that is a plain string is a value, so `fn('upper', 's')` passes the text `s`; use `col('s')` for the column.

```
import duckdb
import pandas as pd
from duckdb import col, fn

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

multiply_by_2 = fn('multiply', col('a'), 2)
duckdb.from_df(df).select(multiply_by_2).show()
```

```
┌─────────────────┐
│ multiply(a, 2)  │
│      int64      │
├─────────────────┤
│               2 │
│               4 │
│               6 │
│               8 │
└─────────────────┘
```

`fn` also passes keyword arguments by name: `fn("round", col("x"), precision=1)` is `round("x", "precision" := 1)`. `coalesce(*values)` gives the first of its arguments that is not NULL. A Python function given as an argument becomes a SQL lambda; it is called once, with expressions, to build the lambda's body:

```
from duckdb import fn

rel = duckdb.sql("SELECT [1, 2, 3] AS l")
print(rel.select(fn("list_transform", col("l"), lambda v: v * 10)).fetchall())
```

```
[([10, 20, 30],)]
```

SQL Expression

This expression contains any valid SQL expression, written as text with `sql_expr(text)`, which splices the text in as it is. The older `SQLExpression(text)` also checks that the text is one expression. Never build either from untrusted input. With the `sales` DataFrame from the top of the page:

```
from duckdb import sql_expr

rel = duckdb.from_df(sales)
print(rel.select(sql_expr("price * 2").alias("double_price")).fetchall())
```

```
[(40,), (84,)]
```

Aggregates and Window Functions

An expression has a method for each common aggregate function, such as `.sum()`, `.mean()`, `.min()`, `.max()`, `.count()`, `.median()`, `.string_agg()` and `.array_agg()`; `fn()` calls any other. `col("g").arg_max(col("x"))` is the value of g in the row where x is largest, and `.arg_min()` takes the row where it is smallest. `count_all()` is `count(*)`. `.where(condition)` makes an aggregate use only the rows where the condition holds. A projection may hold aggregates without a GROUP BY:

```
from duckdb import count_all

rel = duckdb.sql("SELECT * FROM (VALUES ('a', 1), ('a', 2), ('b', 3)) t(g, x)")
print(rel.select(col("x").sum(), col("x").sum().where(col("g") == "a"), count_all()).fetchall())
print(rel.aggregate([col("g"), col("x").max().alias("top")], "g").order("g").fetchall())
```

```
[(6, 3, 3)]
[('a', 2), ('b', 3)]
```

`.over(partition_by=..., order_by=..., rows=..., range=...)` turns an aggregate into a window function. `rows` and `range` bound the window as a pair (start, end) relative to the current row, where None means unbounded; `rows` counts rows and `range` measures the distance in the `order_by` value: `rows=(-2, 0)` is the current row and the two before it. The window functions `row_number()`, `rank()`, `dense_rank()`, `ntile(n)`, `lag(expr)`, `lead(expr)`, `first_value(expr)` and `last_value(expr)` are in the duckdb module, and `.ignore_nulls()` makes the last four skip NULL values:

```
from duckdb import lag, row_number

print(rel.select(
    col("g"),
    col("x"),
    row_number().over(partition_by=col("g"), order_by=col("x")).alias("n"),
    lag(col("x")).over(order_by=col("x")).alias("previous"),
).order("x").fetchall())
```

```
[('a', 1, 1, None), ('a', 2, 2, 1), ('b', 3, 1, 2)]
```

Function Families

`.str()`, `.dt()`, `.list()` and `.json()` give the string, date and time, list and JSON functions as methods that apply the function to the expression: `col("s").str().upper()` is `upper("s")`.

```
rel = duckdb.sql("SELECT 'duck' AS s, DATE '2024-05-01' AS d, [3, 1, 2] AS l")
print(rel.select(
    col("s").str().upper(),
    col("d").dt().year(),
    col("l").list().sort(),
).fetchall())
```

```
[('DUCK', 2024, [1, 2, 3])]
```

Common Operations

The Expression class also contains many operations that can be applied to any Expression type.

Operation	Description
`.alias(name: str)`	Applies an alias to the expression
`.cast(type)`	Applies a cast to the provided type on the expression: a `DuckDBPyType`, a Python type or a SQL type name such as `"VARCHAR"`
`.try_cast(type)`	Like `.cast`, but gives `NULL` where the cast fails
`.isin(*values)`	Creates an `IN` expression against the provided values
`.isnotin(*values)`	Creates a `NOT IN` expression against the provided values
`.between(low, high)`	Checks whether the expression is between `low` and `high`, both included
`.like(pattern)`, `.ilike(pattern)`	Matches the text against a `LIKE` pattern, `ilike` ignoring case
`.is_null()`	Checks whether the expression is `NULL`
`.is_not_null()`	Checks whether the expression is not `NULL`
`.collate(collation: str)`	Compares and sorts by a collation, such as `nocase`
`.concat(*values)`	Concatenates text values
`[key]`	Takes a field of a `STRUCT` by name, an element of a list by position, counting from `1`, or an entry of a map by key

Order Operations

When expressions are provided to `DuckDBPyRelation.order()`, which takes one expression, or to `DuckDBPyRelation.sort()`, which takes several, the following order operations can be applied.

Operation	Description
`.asc()`	Indicates that this expression should be sorted in ascending order
`.desc()`	Indicates that this expression should be sorted in descending order
`.nulls_first()`	Indicates that the nulls in this expression should precede the non-null values
`.nulls_last()`	Indicates that the nulls in this expression should come after the non-null values

Spark API

The DuckDB Spark API implements the PySpark API, allowing you to use the familiar Spark API to interact with DuckDB. All statements are translated to DuckDB's internal plans using our relational API and executed using DuckDB's query engine. The functions it covers are tested against PySpark 4.2.0 and 3.5.9.

> **Warning.** The DuckDB Spark API is experimental and receives bug fixes only: missing features will not be added, and a later release will deprecate it. We recommend SQLFrame, which runs PySpark DataFrame code on DuckDB, as its replacement. SQLFrame does not support DuckDB 2.0 yet: SQLFrame 4.5.0 requires `duckdb<1.6` for its DuckDB engine. Please report bugs in the Spark API on GitHub.

Example

```python
from duckdb.experimental.spark.sql import SparkSession as session
from duckdb.experimental.spark.sql.functions import lit, col

spark = session.builder.getOrCreate()

df = spark.createDataFrame(
    [(34, "Joan"), (45, "Peter"), (23, "John"), (56, "Bob")],
    ["age", "name"],
)
df = df.withColumn("location", lit("Seattle"))
res = df.select(col("age"), col("location")).collect()

print(res)
```

```
[Row(age=34, location='Seattle'), Row(age=45, location='Seattle'), Row(age=23, location='Seattle'),
Row(age=56, location='Seattle')]
```

`createDataFrame` also accepts a pandas DataFrame, which needs pandas to be installed.

Contribution Guidelines

The experimental Spark API accepts bug fixes only. When contributing a fix, please test it with our `pytest` testing framework instead of using temporary files.

Python Client API

Troubleshooting

Troubleshooting

Running EXPLAIN Renders Newlines

In Python, the output of the `EXPLAIN` statement contains hard line breaks (\n):

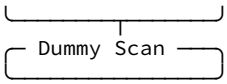
```
In [1]: import duckdb
   ...: duckdb.sql("EXPLAIN SELECT 42 AS x")
```

```
Out[1]:
┌───────────────┬──────────────────────────────────────────────────────────────────────────────────────────
│  explain_key  │                                   explain_value
│    varchar    │                                      varchar
├───────────────┼──────────────────────────────────────────────────────────────────────────────────────────
│ physical_plan │ ┌ Projection ┐\n│ Projections: x │\n│ ~1 row        │\n└──────┬──────┘\n┌ Dummy
Scan ┐\n │
│               │ └───────────────┘\n
└───────────────┴──────────────────────────────────────────────────────────────────────────────────────────
```

To work around this, `print` the output of the `explain()` function:

```
In [2]: print(duckdb.sql("SELECT 42 AS x").explain())
```

```
Out[2]:
┌ Projection ┐
│ Projections: x │
│ ~1 row      │
```



In a Jupyter notebook, `explain()` displays the plan as HTML itself and returns an empty string, so printing it prints nothing.

Please also check out the Jupyter guide for tips on using Jupyter with JupySQL.

Crashes and Errors on Windows

When importing DuckDB on Windows, the Python runtime may crash or return an error upon import or first use:

```
import duckdb

duckdb.sql("...")
```

```
ImportError: DLL load failed while importing duckdb: The specified module could not be found.
```

```
Windows fatal exception: access violation

Current thread 0x0000311c (most recent call first):
  File "<stdin>", line 1 in <module>

Process finished with exit code -1073741819 (0xC0000005)
```

The problem is likely caused by using an outdated Microsoft Visual C++ (MSVC) Redistributable package. The solution is to install the latest MSVC Redistributable package. Alternatively, you can instruct `pip` to compile the package from source as follows:

```
python3 -m pip install duckdb --no-binary duckdb
```

Known Issues

Unfortunately there are some issues that are either beyond our control or are very elusive / hard to track down. Below is a list of these issues that you might have to be aware of, depending on your workflow.

Numpy Import Multithreading

When making use of multi threading and fetching results either directly as Numpy arrays or indirectly through a Pandas DataFrame, it might be necessary to ensure that `numpy.core.multiarray` is imported. If this module has not been imported from the main thread, and a different thread during execution attempts to import it this causes either a deadlock or a crash.

To avoid this, it's recommended to `import numpy.core.multiarray` before starting up threads.

Many Threads per Database

Besides its worker threads, each database starts a pool of asynchronous I/O threads: by default four times the number of CPU cores, at most 256. Every `duckdb.connect()` to an unnamed in-memory database starts a new database, so a process that opens many of them runs many threads. To start fewer, set `async_threads` when connecting:

```
import duckdb

con = duckdb.connect(config={"async_threads": 4})
```

Fetching Time Zone Aware Values Needs `pytz`

Converting a `TIMESTAMP WITH TIME ZONE` value to a Python object, as `fetchone()` and `fetchall()` do, needs the `pytz` package, which is not installed with DuckDB. Without it, the fetch raises an `InvalidInputException` saying `Required module 'pytz' failed to import`. Install `pytz` with `pip install pytz`.

`max_temp_directory_size` Is Ignored in `connect()`

The `max_temp_directory_size` setting has no effect when it is passed in the `config` dictionary of `duckdb.connect()`: `current_setting()` reports the value, but queries write temporary files beyond it. Set it with a `SET` statement after connecting instead:

```
import duckdb

con = duckdb.connect()
con.execute("SET max_temp_directory_size = '10GB'")
```

Arrow Readers Hang on Windows after a Subinterpreter Was Created

On Windows, once the process has created a subinterpreter, a query that scans a `pyarrow.RecordBatchReader` whose batches come from a Python generator can hang.

DataFrame Queries or Boxplots Fail in JupySQL

With JupySQL 0.11.1, run `%sql SET python_scan_all_frames = true` to query notebook DataFrames. This is needed with both native and SQLAlchemy connections. For `%sqlplot boxplot`, use SQLAlchemy. See the Jupyter guide for the complete setup.

Ibis Fails with DuckDB 2.0

Ibis 12.0.0 fails on several operations with DuckDB 2.0, such as `list_tables()`. See the Ibis guide.

`DESCRIBE` Returns an Empty Table in JupySQL

In JupySQL, the `DESCRIBE` statement returns an empty table:

```
%%sql
CREATE OR REPLACE TABLE tbl AS (SELECT 42 AS x);
DESCRIBE tbl;
```

To work around this, wrap it into a subquery:

```
%%sql
CREATE OR REPLACE TABLE tbl AS (SELECT 42 AS x);
FROM (DESCRIBE tbl);
```

Protobuf Error for JupySQL in IPython

Loading the JupySQL extension in IPython fails:

```
In [1]: %load_ext sql

ImportError: cannot import name 'builder' from 'google.protobuf.internal' (unknown location)
```

The solution is to fix the `protobuf` package. This may require uninstalling conflicting packages, e.g.:

```
%pip uninstall tensorflow
%pip install protobuf
```

Migrating Python Code to DuckDB 2.0

This page lists what Python code written for DuckDB 1.x must change to run on DuckDB 2.0. Each item gives the 1.x form, the 2.0 form and the page that explains it. For changes outside the Python client, see Migrating to DuckDB v2.0.

- **Python version.** DuckDB 2.0 requires Python 3.11 or newer; free-threaded builds of CPython are not supported. On Python 3.10, `pip install duckdb` installs a 1.x release. See Installation.
- **Arrow and Polars output of `execute()`.** 1.x: `con.execute(query).fetch_arrow_table()`, `fetch_record_batch()`, `arrow()` and `pl()`. 2.0: these methods are gone from `execute()` results, and `to_arrow_table()` and `to_arrow_reader()` there copy the rows and warn. Use a relation: `con.sql(query).to_arrow_table()`, `con.sql(query).to_arrow_reader()` or `con.sql(query).pl()`. See Arrow and Polars Output.
- **Fetching from a relation.** 1.x: `rel.fetchone()`, `rel.fetchmany(n)`, `rel.execute()` and `rel.fetch_df_chunk()`. 2.0: these methods are gone, because a relation keeps no result between calls and runs its query again on every read. To keep a `fetchone()` or `fetchmany()` loop for a SQL query, create its result with `con.execute(query)` instead of `con.sql(query)`. To iterate over an existing relation, use `for row in rel.iter_rows(): ...`; this keeps the complete result in engine memory. For streamed batches, use `rel.to_arrow_reader()`. Replace `rel.execute()` with `rel.materialize()` when you want to compute and retain the rows.
- **Plain strings in expressions.** 1.x: in `ColumnExpression("s") == "x"` and in `rel.update(set={"a": "x"})`, the string `"x"` named a column. 2.0: in an expression a plain string is a value, so the comparison tests `s` against the text `x` and warns when `x` is also a column name; `update()` refuses a plain string with a `TypeError`. Write `col("x")` for the column and `lit("x")` for the text. See the Expression API and `update`.
- **Dotted column names.** 1.x: `ColumnExpression("t.x")` was the column `x` of `t`. 2.0: a name is never split at a dot, so `"t.x"` is one column name, and a query without a column of that name raises a `BinderException`. Write `col("t", "x")`. See Column Expression.
- **Dicts with keys that are not strings.** 1.x: `con.values({1: "a"})` made a STRUCT with a field named `1`. 2.0: in `values()` and in expressions, such a dict becomes a MAP, here `MAP(INTEGER, VARCHAR)`. Use string keys for a STRUCT. See `dict`.
- **Columns named like new relation methods.** 1.x: `rel.offset` was the column `offset` of `rel`. 2.0: relations have new methods such as `offset`, `sample`, `head`, `rename`, `pivot`, `drop`, `unnest`, `unpivot`, `preview` and `group_by`, so `rel.offset` is the method. Write `rel["offset"]` for the column. See the Relational API.
- **`sql()` of statements that do not read.** 1.x: `con.sql("CREATE TABLE t (i INTEGER)")` returned None. 2.0: it returns a relation over what the statement returned, which a REPL or notebook prints; the statement still runs at the call. Use `con.execute()` for DDL and DML. Printing a relation runs its query, every time. See Basic API Usage.
- **Open `execute()` results.** 1.x: other queries on the connection ran while a result was partly read. 2.0: running another query on the connection raises a `ResourceInUseException` until the result is read to its end or ended with `con.complete()` or `con.abort()`, and the next `execute()` replaces the open result. See Open Results.
- **One connection in several threads.** 1.x: calls from several threads on one connection waited for each other. 2.0: a call while another thread uses the connection raises a `ResourceInUseException`. Give each thread its own `con.cursor()`. See Using Connections in Parallel Python Programs.
- **Failed statements in a transaction.** 1.x: after `con.begin()`, a statement run with `execute()` that failed with a catalog, binder or parser error left the transaction usable. 2.0: any statement run with `execute()` that fails aborts the transaction: `con.commit()` then rolls it back, and any other statement raises a `TransactionException` until `con.rollback()`. See Transactions.
- **Writing files with `overwrite=False`.** 1.x: `rel.write_csv(path, overwrite=False)` and `rel.write_parquet(path, overwrite=False)` replaced an existing file. 2.0: they raise an `IOException`. Leave `overwrite` out to replace the file. See `write_csv` and `write_parquet`.
- **`Decimal` values with a positive exponent.** 1.x: `Decimal("1E+2")` became `1.00` of type `DECIMAL(3,2)`. 2.0: it is `100` of type `DECIMAL(3,0)`. See Conversion between DuckDB and Python.
- **Arrow batch size.** 1.x: the Arrow output of a relation came in batches of 1000000 rows by default. 2.0: the default is 131072 rows. Pass `batch_size` to `to_arrow_table()` or `to_arrow_reader()` for another size. See `to_arrow_reader`.

- **Database files.** 1.x: a database file that DuckDB 1.5 wrote opens in DuckDB 1.5. 2.0: DuckDB 1.5 cannot open a file that DuckDB 2.0 writes in its default format; to keep a file readable by DuckDB 1.5, connect with `config={"storage_compatibility_version": "v1.5.0"}`. See Configuration and Storage Format.

R Client

> Installation To use the DuckDB R client, visit the R installation page.
>
> The latest stable version of the DuckDB R client is {{ latest_version }}.

Installation

duckdb: R Client

The DuckDB R client can be installed using the following command:

```
install.packages("duckdb")
```

Please see the installation page for details.

duckplyr: dplyr Client

DuckDB offers a dplyr-compatible API via the `duckplyr` package. It can be installed using `install.packages("duckplyr")`. For details, see the `duckplyr` documentation.

Reference Manual

The reference manual for the DuckDB R client is available at r.duckdb.org.

Basic Client Usage

The standard DuckDB R client implements the DBI interface for R. If you are not familiar with DBI yet, see the Using DBI page for an introduction.

Startup & Shutdown

To use DuckDB, you must first create a connection object that represents the database. The connection object takes as parameter the database file to read and write from. If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else). The special value `:memory:` (the default) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the R process). If you would like to connect to an existing database in read-only mode, set the `read_only` flag to TRUE. Read-only mode is required if multiple R processes want to access the same database file at the same time.

```
library("duckdb")
# to start an in-memory database
con <- dbConnect(duckdb())
# or
con <- dbConnect(duckdb(), dbdir = ":memory:")
# to use a database file (not shared between processes)
con <- dbConnect(duckdb(), dbdir = "my-db.duckdb", read_only = FALSE)
# to use a database file (shared between processes)
con <- dbConnect(duckdb(), dbdir = "my-db.duckdb", read_only = TRUE)
```

Connections are closed implicitly when they go out of scope or if they are explicitly closed using `dbDisconnect()`. To shut down the database instance associated with the connection, use `duckdb_shutdown()` on the driver object returned by `duckdb()`.

Querying

DuckDB supports the standard DBI methods to send queries and retrieve result sets. `dbExecute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `dbGetQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below is an example.

```
# create a table
dbExecute(con, "CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)")
# insert two items into the table
dbExecute(con, "INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2, 2)")

# retrieve the items again
res <- dbGetQuery(con, "SELECT * FROM items")
print(res)
#     item value count
# 1  jeans  20.0     1
# 2 hammer  42.2     2
```

DuckDB also supports prepared statements in the R client with the `dbExecute` and `dbGetQuery` methods. Here is an example:

```
# prepared statement parameters are given as a list
dbExecute(con, "INSERT INTO items VALUES (?, ?, ?)", list('laptop', 2000, 1))

# if you want to reuse a prepared statement multiple times, use dbSendStatement() and dbBind()
stmt <- dbSendStatement(con, "INSERT INTO items VALUES (?, ?, ?)")
dbBind(stmt, list('iphone', 300, 2))
dbBind(stmt, list('android', 3.5, 1))
dbClearResult(stmt)

# query the database using a prepared statement
res <- dbGetQuery(con, "SELECT item FROM items WHERE value > ?", list(400))
print(res)
#       item
# 1 laptop
```

> **Warning.** Do **not** use prepared statements to insert large amounts of data into DuckDB. See below for better options.

Efficient Transfer

To write a R data frame into DuckDB, use the standard DBI function `dbWriteTable()`. This creates a table in DuckDB and populates it with the data frame contents. For example:

```
dbWriteTable(con, "iris_table", iris)
res <- dbGetQuery(con, "SELECT * FROM iris_table LIMIT 1")
print(res)
#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
```

It is also possible to "register" a R data frame as a virtual table, comparable to a SQL `VIEW`. This *does not actually transfer data* into DuckDB yet. Below is an example:

```
duckdb_register(con, "iris_view", iris)
res <- dbGetQuery(con, "SELECT * FROM iris_view LIMIT 1")
print(res)
#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
```

> DuckDB keeps a reference to the R data frame after registration. This prevents the data frame from being garbage-collected. The reference is cleared when the connection is closed, but can also be cleared manually using the `duckdb_unregister()` method.

Also refer to the data import documentation for more options of efficiently importing data.

dbplyr

DuckDB also plays well with the dbplyr / dplyr packages for programmatic query construction from R. Here is an example:

```
library("duckdb")
library("dplyr")
con <- dbConnect(duckdb())
duckdb_register(con, "flights", nycflights13::flights)

tbl(con, "flights") |>
  group_by(dest) |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

When using dbplyr, CSV and Parquet files can be read using the `dplyr::tbl` function.

```
# Establish a CSV for the sake of this example
write.csv(mtcars, "mtcars.csv")

# Summarize the dataset in DuckDB to avoid reading the entire CSV into R's memory
tbl(con, "mtcars.csv") |>
  group_by(cyl) |>
  summarise(across(disp:wt, .fns = mean)) |>
  collect()

# Establish a set of Parquet files
dbExecute(con, "COPY flights TO 'dataset' (FORMAT parquet, PARTITION_BY (year, month))")

# Summarize the dataset in DuckDB to avoid reading 12 Parquet files into R's memory
tbl(con, "read_parquet('dataset/**/*.parquet', hive_partitioning = true)") |>
  filter(month == "3") |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

Memory Limit

You can use the `memory_limit` configuration option to limit the memory use of DuckDB, e.g.:

```
SET memory_limit = '2GB';
```

Note that this limit is only applied to the memory DuckDB uses and it does not affect the memory use of other R libraries. Therefore, the total memory used by the R process may be higher than the configured `memory_limit`.

Troubleshooting

Warning When Installing on macOS

On macOS, installing DuckDB may result in a warning `unable to load shared object '.../R_X11.so'`:

```
Warning message:
In doTryCatch(return(expr), name, parentenv, handler) :
  unable to load shared object '/Library/Frameworks/R.framework/Resources/modules//R_X11.so':
  dlopen(/Library/Frameworks/R.framework/Resources/modules//R_X11.so, 0x0006): Library not loaded:
/opt/X11/lib/libSM.6.dylib
  Referenced from: <31EADEB5-0A17-3546-9944-9B3747071FE8>
/Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/modules/R_X11.so
  Reason: tried: '/opt/X11/lib/libSM.6.dylib' (no such file) ...
> ')
```

Note that this is just a warning, so the simplest solution is to ignore it. Alternatively, you can install DuckDB from the R-universe:

```
install.packages("duckdb", repos = c("https://duckdb.r-universe.dev", "https://cloud.r-project.org"))
```

You may also install the optional `xquartz` dependency via Homebrew.

Rust

Rust Client

> Installation To use the DuckDB Rust client, visit the Rust installation page.
>
> The latest stable version of the DuckDB Rust client is {{ latest_version }}.

The DuckDB Rust client, `duckdb-rs`, is an ergonomic wrapper over the DuckDB C API that exposes an interface modeled on rusqlite. It supports type-safe queries, bulk loading with the Appender, Apache Arrow interchange, user-defined functions, and building DuckDB extensions in Rust. This page focuses on installation. The other pages in this section cover connecting and each feature in detail.

Installation

The DuckDB Rust client is published on crates.io. The full API is documented on docs.rs. Add it to a project with `cargo`, enabling the `bundled` feature so that DuckDB is compiled from source and no system library is required:

```
cargo add duckdb --features bundled
```

The equivalent `Cargo.toml` entry pins the version. Starting with DuckDB 1.5.0 the crate version has the form `1.<major_minor_patch>.x`. A tilde requirement receives patch releases without changing the bundled DuckDB version:

```
[dependencies]
duckdb = { version = "~{{ latest_package }}", features = ["bundled"] }
```

The `bundled` feature is the simplest way to get started. Without it, the `libduckdb-sys` crate links against a system DuckDB library instead of compiling one, which is covered in Troubleshoot.

To track the latest bindings before they are released on crates.io, depend on the `main` branch or a specific commit from git. The bundled DuckDB version is whatever that commit vendors:

```
[dependencies]
# Latest development version
duckdb = { git = "https://github.com/duckdb/duckdb-rs", branch = "main", features = ["bundled"] }

# A specific commit
duckdb = { git = "https://github.com/duckdb/duckdb-rs", rev = "abc123def", features = ["bundled"] }
```

For the DuckDB 1.4 Andium LTS release line, use the `v1.4-andium` branch.

> **Tip.** The Rust client offers a large set of Cargo feature flags that turn on file-format readers, the Arrow and Polars integrations, connection pooling, and the function and extension interfaces. Enable only the features a project uses to keep build times and binary size down.

Basic API Usage

To use DuckDB, first initialize a `Connection` with `Connection::open_in_memory()` for an in-memory database, or `Connection::open(path)` for a database file. Queries are sent with `execute()` and `execute_batch()`, and results are read by preparing a statement and mapping each row to a Rust value with `query_map()`:

```rust
use duckdb::{params, Connection, Result};

#[derive(Debug)]
struct Person {
    id: i32,
    name: String,
    data: Option<Vec<u8>>,
}

fn main() -> Result<()> {
    let conn = Connection::open_in_memory()?;

    conn.execute_batch(
        r"CREATE SEQUENCE seq;
          CREATE TABLE person (
              id   INTEGER PRIMARY KEY DEFAULT NEXTVAL('seq'),
              name TEXT NOT NULL,
              data BLOB
          );",
    )?;

    let me = Person {
        id: 0,
        name: "Steven".to_string(),
        data: None,
    };
    conn.execute(
        "INSERT INTO person (name, data) VALUES (?, ?)",
        params![me.name, me.data],
    )?;

    let mut stmt = conn.prepare("SELECT id, name, data FROM person")?;
    let person_iter = stmt.query_map([], |row| {
        Ok(Person {
            id: row.get(0)?,
            name: row.get(1)?,
            data: row.get(2)?,
        })
    })?;

    for person in person_iter {
        println!("Found person {:?}", person.unwrap());
    }
    Ok(())
}
```

This example is adapted from the crate's `basic` example. Run Queries covers sending queries and reading results in full.

Feature Flags

The crate is modular: most integrations are gated behind Cargo features that are off by default. Enable them in the dependency declaration, for example `features = ["bundled", "vtab", "appender-arrow"]`. The following is the complete set of the crate's Cargo features, grouped by purpose.

Building DuckDB

Feature	Enables
`bundled`	Compile DuckDB from the bundled source with the `cc` crate, so no system library is needed.

Feature	Enables
`bundled-cmake`	Experimental. Compile the bundled source with DuckDB's upstream CMake build instead of `cc`, which is required for the bundled extensions below. Only available from a `git` checkout, not from crates.io. Implies `bundled` and `parquet`.
`buildtime_bindgen`	Regenerate the C API bindings at build time with `bindgen` instead of using the shipped pregenerated bindings.

File Formats

Both imply `bundled`.

Feature	Enables
`json`	Reading and writing JSON, statically linked.
`parquet`	Reading and writing Parquet, statically linked.

Bundled Extensions

Each statically links the matching bundled extension. All imply `bundled-cmake`, so they need a `git` checkout.

Feature	Enables
`autocomplete`	The autocomplete extension.
`icu`	The ICU extension for locale-aware and time-zone-aware operations, which the plain `bundled` build omits.
`tpch`	The TPC-H benchmark extension.
`tpcds`	The TPC-DS benchmark extension.

Table and Scalar Functions

Feature	Enables
`vtab`	Base support for table functions.
`vtab-arrow`	Apache Arrow integration for table functions, converting between an Arrow `RecordBatch` and DuckDB data chunks. Implies `vtab`.
`vscalar`	Scalar functions. Implies `vtab-arrow`.
`vscalar-arrow`	Arrow-optimized scalar functions, intended for use alongside `vscalar`.
`appender-arrow`	Append Arrow `RecordBatches` through the Appender. Implies `vtab-arrow`.
`loadable-extension`	Experimental. Build a loadable DuckDB extension rather than a client application. See Building a Loadable Extension.

Ecosystem Integrations

Feature	Enables
`polars`	Exchange query results as Polars data frames. See Handle Results.
`r2d2`	A connection pool through the `r2d2` crate. See Connect.
`chrono`	`ToSql` and `FromSql` conversions for `chrono` date and time types.
`rust_decimal`	`ToSql` and `FromSql` conversions for `rust_decimal::Decimal`.
`serde_json`	`ToSql` and `FromSql` conversions for `serde_json::Value`.
`url`	`ToSql` and `FromSql` conversions for `url::Url`.
`uuid`	`ToSql` and `FromSql` conversions for `uuid::Uuid`.

Umbrella Features

Feature	Enables
`vtab-full`	`vtab-arrow` and `appender-arrow`, plus the deprecated `vtab-excel`.
`extensions-full`	`json`, `parquet` and `vtab-full`.
`modern-full`	`chrono`, `serde_json`, `url`, `r2d2`, `uuid`, `polars` and `rust_decimal`.

Deprecated Features

Feature	Enables
`vtab-excel`	A no-op retained for feature compatibility.
`vtab-loadable`	Superseded by `loadable-extension`.

Further Reading

- Connect — opening in-memory and file-backed databases, `Config` options, connection pooling, and thread safety.
- Run Queries — `execute()`, prepared statements, parameter binding, and mapping rows to Rust types.
- Import Data — bulk loading with the Appender and reading directly from Parquet, CSV, and JSON files.
- Handle Results — Apache Arrow and Polars result interchange.
- Write User Defined Functions — user-defined scalar and table functions, and loadable extensions.
- Profile and Monitor — query profiling and interrupting long-running queries.
- Troubleshoot — linking, the ICU extension, and extension-versus-client build issues.
- Clients Overview — the other client APIs DuckDB provides alongside Rust.

Connect

Overview

Every DuckDB feature reachable from Rust starts with a `Connection`. Opening in-memory and file-backed databases, configuring a database instance with `Config`, opening read-only, pooling connections across threads, the crate's thread-safety guarantees, and closing a connection are all covered below.

Opening a Connection

A `Connection` is created with `Connection::open_in_memory()` for an in-memory database, or `Connection::open(path)` for a database file. Both return a `Result`, so the `?` operator propagates any error:

```rust
use duckdb::{Connection, Result};

// In-memory database: nothing is persisted to disk.
let conn = Connection::open_in_memory()?;

// File-backed database: created if it does not exist.
let conn = Connection::open("my_database.duckdb")?;
```

For an in-memory database no data is persisted to disk, so all data is lost when the process exits. A file-backed database is created if the file does not exist. The extension may be `.db`, `.duckdb`, or anything else.

Configuring a Connection

To set DuckDB options when the database instance starts, build a `Config` and open the connection with `Connection::open_with_flags()` (or `Connection::open_in_memory_with_flags()`). `Config` uses a builder style: each method consumes the config and returns it, and each returns a `Result` because DuckDB validates the option:

```rust
use duckdb::{Config, Connection, Result};

let config = Config::default()
    .max_memory("4GB")?
    .threads(4)?;
let conn = Connection::open_with_flags("my_database.duckdb", config)?;
```

`Config` exposes typed methods for the most common options, including `access_mode()`, `max_memory()`, `threads()`, `default_order()`, `default_null_order()`, `enable_external_access()`, `enable_object_cache()`, `custom_user_agent()`, and `allow_unsigned_extensions()`. Any other DuckDB setting can be supplied by name with `with()`:

```rust
let config = Config::default()
    .with("temp_directory", "/path/to/temp/dir/")?
    .with("preserve_insertion_order", "false")?;
```

The full list of settings is on the Configuration page. Many of them can also be changed after connecting with a SET statement or the equivalent PRAGMA.

Read-Only Connections

A database file can be opened in **read-only** mode, which is useful when several processes read the same file at once. Set the access mode on a `Config` with the `AccessMode` enum:

```rust
use duckdb::{AccessMode, Config, Connection, Result};

let config = Config::default().access_mode(AccessMode::ReadOnly)?;
let conn = Connection::open_with_flags("my_database.duckdb", config)?;
```

`AccessMode` has three variants: `Automatic` (the default), `ReadOnly`, and `ReadWrite`.

Thread Safety

A `Connection` is `Send` but not `Sync`: it can be moved to another thread, but it cannot be shared between threads at the same time. The recommended pattern is one connection per thread. To open several connections to the same in-memory or file-backed database, use `Connection::try_clone()`, which opens a new connection to the already-opened database:

```
let conn = Connection::open("my_database.duckdb")?;
let conn2 = conn.try_clone()?; // a second connection to the same database
```

DuckDB runs queries on its own pool of native threads, sized by the `threads` option and shared by all connections to a database instance. See the DuckDB concurrency documentation for how connections and threads interact.

Connection Pooling

For applications that hand a connection to each worker thread, the `r2d2` feature integrates DuckDB with the `r2d2` connection pool. A `DuckdbConnectionManager` creates connections that the pool checks out and returns:

```
use duckdb::{params, DuckdbConnectionManager};

let manager = DuckdbConnectionManager::file("my_database.duckdb")?;
let pool = r2d2::Pool::new(manager)?;

// Each worker checks out its own connection from the pool.
let conn = pool.get()?;
conn.execute("INSERT INTO foo (bar) VALUES (?)", params![1])?;
```

`DuckdbConnectionManager::memory()` builds a manager for an in-memory database, and `DuckdbConnectionManager::file_with_flags()` and `memory_with_flags()` accept a `Config`. Enable the feature in `Cargo.toml` with `features = ["r2d2"]`.

Closing a Connection

A `Connection` closes automatically when it is dropped (goes out of scope), which shuts down the underlying database. To close it explicitly and handle any error, call `close()`:

```
match conn.close() {
    Ok(()) => {}
    Err((_conn, err)) => eprintln!("failed to close connection: {err}"),
}
```

`close()` returns the connection back to the caller on failure, so closing can be retried. In the typical case there is little difference between an explicit `close()` and letting the connection drop, but `close()` gives a chance to observe and handle a shutdown error that `Drop` would otherwise discard.

Further Reading

- Run Queries — using the `Connection` to send queries and read results.
- Configuration — the full list of DuckDB settings that can be passed to `Config`.
- Concurrency — how DuckDB handles multiple connections and threads.
- Troubleshoot — linking and build problems encountered when opening a connection.

Import Data

Overview

The Rust client offers two ways to get data into DuckDB in bulk: the high-performance Appender for inserting rows from Rust, and reading directly from data files such as Parquet, CSV, and JSON. Both are described below.

Appender

The Appender is the fastest way to insert rows generated in Rust. Create one from a `Connection` with `appender()`, passing the target table name, then push rows with `append_row()`. Each row is a parameter list built with the `params!` macro:

```rust
use duckdb::{params, Connection, Result};

fn insert_rows(conn: &Connection) -> Result<()> {
    conn.execute_batch("CREATE TABLE foo (a INTEGER, b INTEGER)")?;

    let mut app = conn.appender("foo")?;
    app.append_row(params![1, 2])?;
    app.append_row(params![3, 4])?;
    app.flush()?;
    Ok(())
}
```

To append many rows from an iterator of parameter lists in one call, use `append_rows()`:

```rust
app.append_rows([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])?;
```

Additional constructors target tables outside the default schema: `appender_to_db()` takes a schema name, and `appender_to_catalog_and_db()` takes a catalog and a schema. To append only a subset of columns, use `appender_with_columns()` (and its schema and catalog variants).

> **Warning.** DuckDB buffers appended rows and checks constraints such as `NOT NULL` and foreign keys only when the buffer is flushed, so `append_row()` may return `Ok` for a row that later fails a constraint check. Always call `flush()` explicitly and check its result before the appender is dropped. Dropping an appender flushes any remaining rows, but errors raised during that implicit flush are discarded, because Rust's `Drop` cannot report them.

The `Appender` is `Sync` but not `Send`, because it borrows the `Connection`. To append from another thread, move the `Connection` to that thread and create the appender there.

Appending Inside a Transaction

For a large load, wrapping the appends in a transaction lets DuckDB commit them as a single unit. Begin a transaction with `Connection::transaction()`, set it to commit on drop, and create the appender from the transaction. The following bulk-inserts ten million rows, adapted from the crate's `appender` example:

```rust
use duckdb::{params, Connection, DropBehavior, Result};

fn main() -> Result<()> {
    let mut db = Connection::open_in_memory()?;
    db.execute_batch(
        "CREATE TABLE test (id INTEGER NOT NULL, area CHAR(6), age TINYINT NOT NULL, active TINYINT NOT
        NULL)",
    )?;

    let row_count = 10_000_000;
    {
        let mut tx = db.transaction()?;
        tx.set_drop_behavior(DropBehavior::Commit);
        let mut app = tx.appender("test")?;
        for i in 0..row_count {
            app.append_row(params![i, "123456", 15, 1])?;
        }
    } // the appender flushes and the transaction commits here, on drop

    let count = db.query_row("SELECT count(*) FROM test", [], |row| row.get::<_, i64>(0))?;
    assert_eq!(count, row_count);
    Ok(())
}
```

A `Transaction` rolls back by default when dropped. Call `set_drop_behavior(DropBehavior::Commit)` to commit on drop instead, or call `tx.commit()` explicitly. The other `DropBehavior` variants are `Rollback` (the default), `Ignore` (leave the transaction open), and `Panic`.

Appending Arrow Data

With the `appender-arrow` feature enabled, an Appender can append an Apache Arrow `RecordBatch` directly, which avoids building a parameter list per row. See Handle Results for the Arrow integration.

Reading Data Files

DuckDB can read many file formats directly in SQL, which is often the simplest way to load external data. The Rust client just sends the query. For Parquet, enable the `parquet` feature and query `read_parquet()`, adapted from the crate's `parquet` example:

```
use duckdb::{Connection, Result};
use duckdb::arrow::record_batch::RecordBatch;

let conn = Connection::open_in_memory()?;
conn.execute_batch("INSTALL parquet; LOAD parquet;")?;

let batches: Vec<RecordBatch> = conn
    .prepare("SELECT * FROM read_parquet(?)")?
    .query_arrow(["data.parquet"])?
    .collect();
```

The same pattern reads CSV with `read_csv()` and JSON with `read_json()` (the latter needs the `json` feature). To load a file into a table rather than query it in place, wrap the reader in a `CREATE TABLE ... AS SELECT` or `COPY` statement sent with `execute()`.

Further Reading

- Appender — the engine-level Appender interface that the Rust `Appender` wraps.
- Data Import — DuckDB's full set of bulk-loading options.
- Handle Results — the Apache Arrow integration used to append and read record batches.
- Run Queries — sending the `CREATE TABLE` and `COPY` statements the file readers build on.

Run Queries

Overview

A `Connection` runs SQL through two families of methods:

- The `execute_*` methods (`execute()` and `execute_batch()`) send statements that do not return rows, such as `INSERT` or DDL, and report the number of affected rows.
- The `query_*` methods (`query()`, `query_map()`, and `query_row()`) run statements that return rows and hand each Row back to be mapped into a Rust value.

Prepared statements work with both. The sections below walk through sending statements, binding parameters, and turning rows into typed Rust values. For opening the `Connection` these run on, see Connect.

Sending Statements

Use `execute()` for a single statement that does not return rows, such as `INSERT` or `UPDATE`. It takes the SQL and a list of parameters, and returns the number of affected rows. Pass an empty slice `[]` when there are no parameters:

```
conn.execute(
    "CREATE TABLE person (id INTEGER, name TEXT, data BLOB)",
    [],
)?;

let rows_changed = conn.execute(
    "INSERT INTO person (id, name) VALUES (1, 'Steven')",
    [],
)?;
```

To run several statements at once, for example a schema-setup script, use `execute_batch()`:

```
conn.execute_batch(
    r"CREATE SEQUENCE seq;
      CREATE TABLE person (
          id   INTEGER PRIMARY KEY DEFAULT NEXTVAL('seq'),
          name TEXT NOT NULL,
          data BLOB
      );",
)?;
```

Binding Parameters

Values are bound to a statement's placeholders rather than formatted into the SQL string. DuckDB uses positional (`?`) and numbered (`?1`, `?2`) placeholders. The `params!` macro packs a heterogeneous list of values into the parameter slice:

```
use duckdb::params;

conn.execute(
    "INSERT INTO person (id, name, data) VALUES (?, ?, ?)",
    params![1, "Steven", None::<Vec<u8>>],
)?;
```

For named placeholders such as `$name`, use the `named_params!` macro:

```
use duckdb::named_params;

conn.query_row(
    "SELECT $age >= 18 AND $name = 'Alice'",
    named_params! {
        "age": min_age,
        "name": name,
    },
    |row| row.get(0),
)?;
```

Any value bound as a parameter implements the `ToSql` trait, and any value read out of a row implements `FromSql`. The crate provides both for the standard Rust numeric, string, byte-slice, and boolean types. `Option<T>` maps to and from SQL `NULL`, so `None` binds as `NULL` and a `NULL` result reads back as `None`.

> **Warning.** Do *not* use prepared statements to insert large amounts of data into DuckDB. See Import Data for the Appender and other faster options.

Prepared Statements

`Connection::prepare()` compiles a statement once so it can be executed repeatedly. The returned `Statement` can be executed with `execute()`, or queried to read rows back.

```
let mut stmt = conn.prepare("INSERT INTO person (id, name) VALUES (?, ?)")?;
stmt.execute(params![1, "Steven"])?;
stmt.execute(params![2, "Jane"])?;
```

Mapping Rows to Rust Values

To read a result set, prepare a `SELECT` and call `query_map()`, passing a closure that builds a value from each Row. Columns are read by 0-based index with `row.get()`, whose return type is inferred from the struct field it is assigned to. `query_map()` yields an iterator of `Result`, which can be collected into a `Vec`:

```
#[derive(Debug)]
struct Person {
    id: i32,
    name: String,
    data: Option<Vec<u8>>,
}

let mut stmt = conn.prepare("SELECT id, name, data FROM person")?;
let people = stmt
    .query_map([], |row| {
        Ok(Person {
            id: row.get(0)?,
            name: row.get(1)?,
            data: row.get(2)?,
        })
    })?
    .collect::<Result<Vec<_>>>()?;

for person in people {
    println!("Found person {person:?}");
}
```

Columns can also be read by name with `row.get("name")`. This example follows the crate's `basic` example.

Reading a Single Row

When a query returns exactly one row, `Connection::query_row()` runs it and applies a closure to that row in one call, without preparing a statement by hand. The `<(T,)>::try_from(row)` helper converts a single-column row into a tuple:

```
let count = conn.query_row(
    "SELECT count(*) FROM person",
    [],
    |row| row.get::<_, i64>(0),
)?;

// A whole row can be converted to a tuple in one step.
let (n,) = conn.query_row("SELECT count(*) FROM person", [], |row| {
    <(i64,)>::try_from(row)
})?;
```

`query_row_and_then()` is the counterpart for a closure that returns a custom error type.

Reading Nested and Composite Types

DuckDB's nested and composite types are read through the `Value` enum, which `row.get()` returns when the target type is `Value`. `Value` has variants such as `Value::List`, `Value::Struct`, `Value::Map`, `Value::Array`, `Value::Union`, `Value::Enum`, and `Value::Decimal`, so a query that selects nested columns can be matched on:

```
use duckdb::types::Value;

let mut stmt = conn.prepare("SELECT [1, 2, 3] AS l, {'a': 1, 'b': 2} AS s")?;
let mut rows = stmt.query([])?;
while let Some(row) = rows.next()? {
    let list: Value = row.get(0)?;
    let strukt: Value = row.get(1)?;
```

```
    println!("{list:?} {strukt:?}");
}
```

`Value`, `ValueRef`, and `Type` are marked non-exhaustive, so a `match` over them should include a wildcard arm to stay forward-compatible as DuckDB adds types. To read whole result sets as columnar batches instead of row by row, see Handle Results.

Further Reading

- Handle Results — reading results as Apache Arrow record batches or Polars data frames instead of row by row.
- Import Data — the Appender, the recommended alternative to prepared statements for bulk inserts.
- Prepared Statements — DuckDB's SQL-level support for the parameterized queries used here.
- Connect — opening the `Connection` that these statements run on.

Handle Results

Overview

Beyond reading a result set row by row, the Rust client can hand a query result to Apache Arrow as a stream of record batches, register an Arrow batch as a queryable table, and return results as Polars data frames. Each of these columnar result-handling options is described below.

Apache Arrow

DuckDB is columnar, and its native way to return a result to Rust in bulk is Apache Arrow. The crate re-exports the `arrow` crate, so no separate Arrow dependency or version alignment is required: reach the Arrow types through `duckdb::arrow`.

Reading a Result as Arrow

Call `query_arrow()` on a prepared statement to get an iterator of `RecordBatch`. Collecting it materializes the whole result, while iterating it reads one batch at a time:

```
use duckdb::{Connection, Result};
use duckdb::arrow::record_batch::RecordBatch;
use duckdb::arrow::util::pretty::print_batches;

let conn = Connection::open_in_memory()?;
let mut stmt = conn.prepare("SELECT * FROM generate_series(1, 5)")?;
let batches: Vec<RecordBatch> = stmt.query_arrow([])?.collect();
print_batches(&batches).unwrap();
```

`get_schema()` on the returned handle reports the Arrow schema DuckDB inferred for the result.

Streaming Arrow Results

`query_arrow()` runs the statement to completion and buffers the whole result on the client side, so a large result is held in memory even while the iterator is advanced one batch at a time. For a result that should be consumed lazily, fetching chunks only as the iterator advances, use `stream_arrow()`, which is otherwise identical:

```
let mut stmt = conn.prepare("SELECT * FROM big_table")?;
for batch in stmt.stream_arrow([])? {
    // process one RecordBatch at a time
    println!("{} rows", batch.num_rows());
}
```

DuckDB may still materialize the result internally for some statements. The streaming iterator panics if fetching or Arrow conversion fails after execution has started.

Querying an Arrow Batch

An Arrow `RecordBatch` produced elsewhere in a Rust program can be registered as a DuckDB table function and queried in SQL. Enable the `vtab-arrow` feature, register the built-in `ArrowVTab` table function on the connection, and pass the batch as a query parameter with `arrow_recordbatch_to_query_params()`. The following is adapted from the crate's `arrow_vtab` example:

```
use duckdb::{Connection, arrow::record_batch::RecordBatch};
use duckdb::vtab::arrow::{arrow_recordbatch_to_query_params, ArrowVTab};

let conn = Connection::open_in_memory()?;
conn.register_table_function::<ArrowVTab>("arrow")?;

let params = arrow_recordbatch_to_query_params(cities_batch);
let batches: Vec<RecordBatch> = conn
    .prepare(
        "SELECT city, population
         FROM arrow(?, ?)
         WHERE coastal AND population >= 500000
         ORDER BY population DESC",
    )?
    .query_arrow(params)?
    .collect();
```

The batch is addressed by the name given to `register_table_function()` (`arrow` here), and DuckDB filters, orders, and aggregates it like any other table. `arrow_recordbatch_to_query_params()` expands a batch into the two parameters the `arrow(?, ?)` function expects.

Polars Data Frames

With the `polars` feature enabled, a query result can be returned as Polars `DataFrames`. Call `query_polars()` on a prepared statement to get an iterator of data frames, one per result chunk:

```
use duckdb::{Connection, Result};
use polars::prelude::DataFrame;

let conn = Connection::open_in_memory()?;
let mut stmt = conn.prepare("SELECT * FROM test")?;
let dfs: Vec<DataFrame> = stmt.query_polars([])?.collect();
```

To combine the chunks into a single `DataFrame`, use `accumulate_dataframes_vertical_unchecked` from `polars_core`. The crate re-exports `polars`, so its types are also reachable through `duckdb::polars`.

Further Reading

- Run Queries — sending the queries whose results this page reads, and reading them row by row.
- Write User Defined Functions — writing table functions, of which the built-in `ArrowVTab` is one.
- Import Data — appending Arrow record batches into a table with the `appender-arrow` feature.

Write User Defined Functions

Overview

The Rust client can register user-defined functions written in Rust: scalar functions with the `vscalar` feature and table functions with the `vtab` feature. Both operate on DuckDB's columnar data chunks, so a single call processes a batch of rows. The sections below build and register each, then point to a template for packaging them as a loadable extension.

Scalar Functions

A scalar function is a type that implements the `VScalar` trait, registered on a connection with `register_scalar_function()`. DuckDB calls the function's `invoke()` once per data chunk, passing a batch of input rows and expecting the corresponding output values written back. `signatures()` declares the argument and return types.

The following defines `add_one(BIGINT) -> BIGINT`, adapted from the crate's `vscalar` example. It reads the single `BIGINT` argument as a slice of `i64` and writes each result into the output vector:

```
use duckdb::{
    Connection, Result,
    core::{DataChunkHandle, LogicalTypeHandle, LogicalTypeId},
    vscalar::{ScalarFunctionSignature, VScalar},
    vtab::arrow::WritableVector,
};

struct AddOne;

impl VScalar for AddOne {
    // No per-function state is needed.
    type State = ();

    fn invoke(
        _: &Self::State,
        input: &mut DataChunkHandle,
        output: &mut dyn WritableVector,
    ) -> Result<(), Box<dyn std::error::Error>> {
        let len = input.len();
        let src = input.flat_vector(0);
        let src = unsafe { src.as_slice_with_len::<i64>(len) };

        let mut out = output.flat_vector();
        let dst = unsafe { out.as_mut_slice_with_len::<i64>(len) };

        for (d, s) in dst.iter_mut().zip(src) {
            *d = s + 1;
        }
        Ok(())
    }

    fn signatures() -> Vec<ScalarFunctionSignature> {
        vec![ScalarFunctionSignature::exact(
            vec![LogicalTypeHandle::from(LogicalTypeId::Bigint)],
            LogicalTypeHandle::from(LogicalTypeId::Bigint),
        )]
    }
}

fn main() -> Result<()> {
    let conn = Connection::open_in_memory()?;
    conn.register_scalar_function::<AddOne>("add_one")?;

    let mut stmt = conn.prepare("SELECT add_one(i) FROM range(5) t(i)")?;
    let values: Vec<i64> = stmt.query_map([], |row| row.get(0))?.collect::<Result<_>>()?;
    assert_eq!(values, vec![1, 2, 3, 4, 5]);
    Ok(())
}
```

Once registered, the function is called from SQL by the name passed to `register_scalar_function()`:

```
SELECT add_one(41);
```

The `associated type State` holds any per-function state shared across calls; use `()` when none is needed. The example reads its input as a typed slice for brevity and assumes non-null input; a production function should also check each value's validity. This feature requires `features = ["vscalar"]`.

Table Functions

A table function is a type that implements the `VTab` trait, registered with `register_table_function()` and queried like any other table. `VTab` splits the work across three callbacks:

- `bind` runs once when the statement is prepared. It declares the result columns with `add_result_column()` and reads the call's arguments with `get_parameter()`, returning read-only bind data.
- `init` runs once before execution and returns per-scan state, such as a cursor.
- `func` produces one chunk of rows per call. It fills the output vectors and sets the chunk length. A length of `0` ends the scan. DuckDB calls it repeatedly until the scan is done, so a large result is delivered over several calls.
- `parameters` declares the function's parameter types.

The crate's `vtab` example defines a `numbers(BIGINT)` function that returns one row per integer in `0..count`, over columns of increasing complexity. Its skeleton, showing the bind data, init state, and per-chunk execution model:

```rust
use duckdb::{
    Connection, Result,
    core::{DataChunkHandle, LogicalTypeHandle, LogicalTypeId},
    vtab::{BindInfo, InitInfo, TableFunctionInfo, VTab},
};
use std::sync::atomic::{AtomicUsize, Ordering};

struct Numbers;

// Computed once from the call's arguments; read-only during execution.
struct NumbersBind {
    count: usize,
}

// Tracks scan progress: func is called once per output chunk.
struct NumbersInit {
    cursor: AtomicUsize,
}

impl VTab for Numbers {
    type BindData = NumbersBind;
    type InitData = NumbersInit;

    fn bind(bind: &BindInfo) -> Result<Self::BindData, Box<dyn std::error::Error>> {
        bind.add_result_column("n", LogicalTypeId::Bigint.into());
        let count = bind.get_parameter(0).to_int64().max(0) as usize;
        Ok(NumbersBind { count })
    }

    fn init(_: &InitInfo) -> Result<Self::InitData, Box<dyn std::error::Error>> {
        Ok(NumbersInit { cursor: AtomicUsize::new(0) })
    }

    fn func(
        func: &TableFunctionInfo<Self>,
        output: &mut DataChunkHandle,
    ) -> Result<(), Box<dyn std::error::Error>> {
        let count = func.get_bind_data().count;
        let cursor = &func.get_init_data().cursor;

        // A chunk holds at most a vector's worth of rows.
        let capacity = output.flat_vector(0).capacity();
        let start = cursor.fetch_add(capacity, Ordering::Relaxed);
        if start >= count {
            output.set_len(0); // no more rows: end the scan
            return Ok(());
        }
        let rows = capacity.min(count - start);
```

```
        let mut v = output.flat_vector(0);
        let slice = unsafe { v.as_mut_slice_with_len::<i64>(rows) };
        for (i, slot) in slice.iter_mut().enumerate() {
            *slot = (start + i) as i64;
        }
        output.set_len(rows);
        Ok(())
    }

    fn parameters() -> Option<Vec<LogicalTypeHandle>> {
        Some(vec![LogicalTypeId::Bigint.into()])
    }
}

fn main() -> Result<()> {
    let conn = Connection::open_in_memory()?;
    conn.register_table_function::<Numbers>("numbers")?;

    let total: i64 = conn.query_row("SELECT count(*) FROM numbers(5000)", [], |row| row.get(0))?;
    assert_eq!(total, 5000);
    Ok(())
}
```

```
SELECT * FROM numbers(6);
```

Because `func` claims its slice of the result with an atomic `fetch_add`, the scan stays correct even when DuckDB runs it on multiple threads. The full example also populates `DOUBLE`, `VARCHAR` (with `NULL` handling via `set_null`), `LIST<BIGINT>`, and `STRUCT` columns, showing how to write each vector kind. The Rust table function interface mirrors DuckDB's C API table functions as closely as possible. This feature requires `features = ["vtab"]`; the `vtab-arrow` feature adds the built-in `ArrowVTab`, covered in Handle Results.

Building a Loadable Extension

The functions above are registered on a `Connection` inside a client application. The same `VScalar` and `VTab` interfaces can instead be packaged as a loadable DuckDB extension that any DuckDB client can `LOAD`. This uses the `loadable-extension` feature and the `duckdb_entrypoint_c_api` macro to declare the extension entry point. The crate's `hello-ext` example registers a `hello(name)` table function this way.

> **Warning.** Enable `loadable-extension` only when building an extension. In a client application, that feature makes calls such as `Connection::open_in_memory()` panic, because they run outside an extension context.

A plain `cargo build` does not produce a loadable file: DuckDB requires a file ending in `.duckdb_extension` with a valid metadata footer. Start from the official extension-template-rs template, whose `make debug` produces a `.duckdb_extension` that loads with `duckdb -unsigned`.

Further Reading

- Handle Results — the built-in `ArrowVTab` table function and the Arrow vector API shared with user-defined functions.
- C API — the table function interface that the Rust `VTab` API mirrors.
- Extensions — DuckDB's extension system that a loadable Rust extension plugs into.
- Run Queries — calling registered functions from SQL.

Profile and Monitor

Overview

The Rust client can observe a running or completed query: it can retrieve per-operator profiling metrics after a query runs, and interrupt a long-running query from another thread. Both are shown below.

Profiling

Profiling must be enabled on a connection before metrics are collected. Enable it with the `enable_profiling` PRAGMA, run a query, then read the result with `get_profiling_info()`:

```
use duckdb::Connection;

let conn = Connection::open_in_memory()?;

// Do not print profiling output to stdout after each query (the default when profiling is enabled).
conn.execute("PRAGMA enable_profiling = 'no_output'", [])?;

conn.execute("SELECT 42", [])?;

let info = conn.get_profiling_info().expect("profiling should be enabled");
println!("profiled children: {}", info.children.len());
```

`get_profiling_info()` returns an `Option<ProfilingInfo>`, which is None when profiling is not enabled. `ProfilingInfo` is a tree whose metrics are grouped and named in lowercase (see the metrics list): the root's children are the root plan operator, with metrics such as `type`, `timing` and `intermediate_rows`, followed by one node per metric group, such as `query` with `total_time`. Read a node's metrics through its `metrics` map and its children through `children`.

Interrupting Queries

A long-running query can be cancelled from another thread. Call `interrupt_handle()` on the connection to get an `Arc<InterruptHandle>`. `InterruptHandle` is both Send and Sync, so you can move it, or the connection, to another thread and call `interrupt()` from wherever is convenient. The interrupted query fails with an error:

```
use duckdb::{Connection, Result};

fn run_query(conn: Connection) -> Result<()> {
    let interrupt_handle = conn.interrupt_handle();

    let join_handle = std::thread::spawn(move || {
        conn.execute("<expensive query>", [])
    });

    // ... later, from this thread, cancel the query:
    interrupt_handle.interrupt();

    let query_result = join_handle.join().unwrap();
    assert!(query_result.is_err());
    Ok(())
}
```

`interrupt()` cancels whichever query is currently running on the connection the handle was obtained from. If the connection has already been dropped, calling `interrupt()` is a no-op.

Further Reading

- Profiling — DuckDB's query profiling output and the `enable_profiling` PRAGMA and the `tracked_metrics` setting.
- Run Queries — running the queries whose metrics this page reads and whose execution it interrupts.
- Connect — the `Connection` that profiling and the interrupt handle operate on.

Troubleshoot

Overview

This page collects common issues encountered when using the DuckDB Rust client, together with their workarounds. If you run into a problem that is not covered here, search the client's issue tracker on GitHub.

Linking against a System Library

The `bundled` feature compiles DuckDB from the source shipped with the crate, so nothing else is needed to build, and it is the simplest choice for most applications. Without `bundled`, the underlying `libduckdb-sys` crate links against a DuckDB library already on the system, which it locates through `pkg-config` or, on the MSVC ABI, Vcpkg. If no library is found, the build fails. There are three ways to build without `bundled`.

Point at a Library You Already Have

Set `DUCKDB_LIB_DIR` to the directory containing the library, and `DUCKDB_INCLUDE_DIR` to the directory containing `duckdb.h` if it is elsewhere. For example, using a prebuilt library from a DuckDB release:

```
wget https://github.com/duckdb/duckdb/releases/download/v2.0.0/duckdb-shared-libs-osx-universal.tar.gz
mkdir libduckdb && tar -xzf duckdb-shared-libs-osx-universal.tar.gz -C libduckdb

export DUCKDB_LIB_DIR=$PWD/libduckdb
export DUCKDB_INCLUDE_DIR=$DUCKDB_LIB_DIR
export DYLD_FALLBACK_LIBRARY_PATH=$DUCKDB_LIB_DIR    # LD_LIBRARY_PATH on Linux
cargo build
```

On Linux, use the matching `duckdb-shared-libs-linux-amd64.tar.gz` or `duckdb-shared-libs-linux-arm64.tar.gz` archive and set `LD_LIBRARY_PATH` instead.

Let the Build Download a Library

Set `DUCKDB_DOWNLOAD_LIB=1` and the build script downloads the prebuilt DuckDB library matching the crate's version from GitHub Releases, caches it under `target/`, and adds it to the linker search path. Leave `DUCKDB_STATIC` unset, because the downloaded archives contain only the dynamic library:

```
DUCKDB_DOWNLOAD_LIB=1 cargo build
```

Use a Package Manager

If DuckDB is installed through `pkg-config` or Vcpkg, the build finds it automatically. `vcpkg` uses static libraries by default; set `VCPKGRS_DYNAMIC=1` to link dynamically instead.

ICU Extension Missing from the Bundled Build

Because crates.io imposes a 10 MB package-size limit, the `bundled` source of earlier crate versions excluded the `icu` extension. In DuckDB 2.0, the `icu` extension no longer depends on the ICU library and implements time zones, calendars and collations natively. If some collation or time-zone-aware date/time operations still fail out of the box with your crate version, install and load the extension at runtime:

```
conn.execute_batch("INSTALL icu; LOAD icu;")?;
```

Alternatively, statically link the extension by enabling the crate's `icu` feature, which pulls it into the build.

`open_in_memory` Panics in an Extension Build

The `loadable-extension` feature builds a DuckDB extension rather than a client application. When it is enabled, entry points such as `Connection::open_in_memory()` panic if they are called outside an extension context, because there is no host DuckDB instance to attach to.

Do not enable `loadable-extension` for a client application. Use it only when building an extension. See Write User Defined Functions for how loadable extensions are built and packaged.

Cross-Compiling with `bundled`

Cross-compiling with the `bundled` feature compiles DuckDB's C++ source for the target, so it requires a working C++ cross-compiler for that target and is supported only on a best-effort basis. When cross-compilation is difficult to set up, link against a prebuilt library for the target with `DUCKDB_LIB_DIR` instead, as described in Linking against a System Library.

Further Reading

- Rust Client — installation, the `bundled` feature, and the full list of Cargo feature flags.
- Connect — opening connections, whose failures are often build or linking problems.
- Write User Defined Functions — the `loadable-extension` feature and how to package an extension correctly.

Wasm

DuckDB Wasm Client

> Installation To use the DuckDB Wasm client, visit the `duckdb-wasm` GitHub repository.
>
> The latest stable version of the DuckDB WebAssembly client is {{ latest_version }}.

DuckDB-Wasm is a version of DuckDB that has been compiled to WebAssembly, the portable binary format that browsers can run at near-native speed. This lets a full DuckDB engine run entirely inside any browser on any device, with no server to send queries to and no data leaving the user's machine.

It offers a layered API to fit different needs:

- embedded as a JavaScript + WebAssembly library
- used as a Web shell
- built from source

This page covers installation and basic usage; the other pages in this section cover instantiating the client, importing data, running queries, loading extensions, and deploying it.

Try It Yourself

The shell below is a full DuckDB engine running in your browser — type a SQL query and run it right here:

> **Note.** The shell above is itself built with DuckDB-Wasm. It is the `@duckdb/duckdb-wasm-shell` package, one of the layered APIs listed above, running on top of the same DuckDB-Wasm library this page describes. Every query you type executes client-side in WebAssembly, with no server round-trip, so it is a live demonstration of what your own application gets when it embeds DuckDB-Wasm.

Installation

DuckDB-Wasm is published to npm as `@duckdb/duckdb-wasm`. Install it with your package manager:

```
npm install @duckdb/duckdb-wasm
```

This installs the current stable release (the `latest` tag). To try the newest, unreleased build instead, install the `next` tag with `npm install @duckdb/duckdb-wasm@next`; it tracks the `main` branch and is intended for testing rather than production.

Alternatively, load a prebuilt bundle directly from a CDN such as jsDelivr without installing anything; see Instantiate for the bundle-selection options. To try it interactively with no setup, use the DuckDB-Wasm Web shell.

Basic API Usage

The example below instantiates DuckDB-Wasm from a CDN bundle, opens a connection, runs a query, and reads the result. Instantiate covers the other ways to load the bundle, such as with webpack, Vite, or self-hosted files.

Refer to the externally hosted DuckDB-Wasm API Reference for details on every class and method. The DuckDB-Wasm launch blog post is another good introduction.

```ts
import * as duckdb from '@duckdb/duckdb-wasm';

// Instantiate DuckDB-Wasm from a jsDelivr bundle
const bundle = await duckdb.selectBundle(duckdb.getJsDelivrBundles());

// Worker scripts must be same-origin, so wrap the CDN worker in a Blob URL
const worker_url = URL.createObjectURL(
    new Blob([`importScripts("${bundle.mainWorker}");`], { type: 'text/javascript' })
);
const worker = new Worker(worker_url);
const db = new duckdb.AsyncDuckDB(new duckdb.ConsoleLogger(), worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
URL.revokeObjectURL(worker_url);

// Open a connection and run a query
const conn = await db.connect();
const result = await conn.query('SELECT 42 AS answer');
console.log(result.toArray()[0].answer); // prints 42

// Release resources
await conn.close();
await db.terminate();
await worker.terminate();
```

How DuckDB-Wasm Works

DuckDB-Wasm uses Apache Arrow as its data protocol for both data import and query results. Arrow is a columnar, database-friendly format that the `apache-arrow` npm package implements in the browser, letting DuckDB-Wasm exchange data efficiently and interoperate with other JavaScript data tools without reimplementing SQL type logic in JavaScript.

DuckDB-Wasm is built on a virtual filesystem that treats local files, remote HTTP(S) servers and in-memory buffers uniformly.

Because DuckDB understands file formats such as Parquet, it reads only the bytes a query needs rather than downloading an entire file: `SELECT count(*) FROM 'file.parquet'` can be answered from the file metadata alone, and selective filters or `LIMIT ... OFFSET` clauses let entire row groups be skipped. This makes it practical to query large Parquet files hosted on a remote server directly from the browser.

> **Note.** This partial-read behavior depends on which httpfs path serves the request. The JavaScript httpfs implementation fetches only the required byte ranges, but reading through the built-in httpfs extension may currently download the whole file instead (see duckdb-wasm issue #2153).

Limitations

- By default, the WebAssembly client only uses a single thread. Multithreading is available but still experimental.
- The WebAssembly client has a limited amount of memory available. WebAssembly limits the amount of available memory to 4 GB and browsers may impose even stricter limits.

See Troubleshoot for workarounds when you run into these limits.

Further Reading

- Instantiate — the bundle-selection patterns for jsDelivr, webpack, Vite, and statically served files.
- Import Data — registering files and inserting Apache Arrow, CSV, JSON, and Parquet data.
- Run Queries — materialized and streaming queries, prepared statements, and exporting results.
- Load Extensions — how extension loading differs from native DuckDB and which extensions are available.
- Deploy — the components a deployment serves and the security considerations involved.
- Troubleshoot — common issues and their workarounds, including CORS errors, memory limits, and threading.
- Clients Overview — the other client APIs DuckDB provides alongside Wasm.

Instantiate

Overview

DuckDB-Wasm can be instantiated in several ways, depending on how your application bundles and serves its assets. Each approach resolves the `mainModule` (the WebAssembly file) and the `mainWorker` (the worker script) for the browser's capabilities, then hands them to `AsyncDuckDB`.

This page shows the patterns for a jsDelivr CDN, webpack, Vite and statically served files. Once instantiated, the db object is used to import data and run queries.

Bundle Selection

DuckDB-Wasm ships several WebAssembly modules compiled for different browser feature sets, because post-MVP WebAssembly features reach browsers at different speeds and each one can bring a flat performance improvement. The `selectBundle` function runs dynamic browser checks and picks the fastest bundle the current browser supports:

- `mvp` — the WebAssembly 1.0 (MVP) baseline, supported everywhere
- `eh` — adds Wasm-level exception handling, which improves performance; DuckDB and DuckDB-Wasm are written in C++ and use exceptions to propagate errors, so native exception handling avoids emulating them through JavaScript
- `coi` — adds threading for parallel query execution; it requires the page to be cross-origin isolated and must be opted into explicitly (see Threading)

The examples below all call `selectBundle` so the browser receives the fastest bundle it can run. You can also inspect the selected bundle and feature set from the web shell with the `.features` command.

`cdn(jsdelivr)`

The simplest way to load DuckDB-Wasm is straight from a CDN, with no build step or bundler configuration. `getJsDelivrBundles()` returns the set of bundles hosted on jsDelivr, and `selectBundle` picks the one that matches the browser. Because a Web Worker script must be same-origin, the CDN worker URL is wrapped in a `Blob` that `importScripts` it, and the temporary object URL is revoked once the worker has started.

```
import * as duckdb from '@duckdb/duckdb-wasm';

const JSDELIVR_BUNDLES = duckdb.getJsDelivrBundles();

// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(JSDELIVR_BUNDLES);

const worker_url = URL.createObjectURL(
  new Blob([`importScripts("${bundle.mainWorker}");`], {type: 'text/javascript'})
);

// Instantiate the asynchronous version of DuckDB-Wasm
const worker = new Worker(worker_url);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
URL.revokeObjectURL(worker_url);
```

`webpack`

When your application is built with webpack, let the bundler resolve and emit DuckDB-Wasm's assets instead of fetching them from a CDN. Import each `.wasm` module directly, and reference the worker scripts with `new URL(..., import.meta.url)` so that webpack fingerprints them and rewrites the paths to the emitted files. The bundles are declared manually because their final locations are known only after the build, and `selectBundle` then chooses between the `mvp` and `eh` variants at runtime.

```
import * as duckdb from '@duckdb/duckdb-wasm';
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb-mvp.wasm';
import duckdb_wasm_next from '@duckdb/duckdb-wasm/dist/duckdb-eh.wasm';
const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: duckdb_wasm,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-mvp.worker.js',
import.meta.url).toString(),
    },
    eh: {
        mainModule: duckdb_wasm_next,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-eh.worker.js',
import.meta.url).toString(),
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronous version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

`vite`

Vite resolves assets a little differently: appending the `?url` suffix to an import tells Vite to return the asset's final URL rather than its contents. Import both the `.wasm` modules and the worker scripts this way, assemble them into the manual bundle definition, and let `selectBundle` select the appropriate variant for the browser at runtime.

```
import * as duckdb from '@duckdb/duckdb-wasm';
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb-mvp.wasm?url';
import mvp_worker from '@duckdb/duckdb-wasm/dist/duckdb-browser-mvp.worker.js?url';
import duckdb_wasm_eh from '@duckdb/duckdb-wasm/dist/duckdb-eh.wasm?url';
import eh_worker from '@duckdb/duckdb-wasm/dist/duckdb-browser-eh.worker.js?url';

const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: duckdb_wasm,
        mainWorker: mvp_worker,
    },
    eh: {
        mainModule: duckdb_wasm_eh,
        mainWorker: eh_worker,
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronous version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

Statically Served

If you would rather not depend on a CDN or a bundler, you can host the DuckDB-Wasm files yourself. Manually download the distribution files from https://cdn.jsdelivr.net/npm/@duckdb/duckdb-wasm/dist/, serve them from your own origin, and point the bundle paths at those locations. This keeps every asset on your own server, which suits offline, air-gapped, or strict content-security-policy deployments. Update the placeholder paths below to match where you serve the files.

```
import * as duckdb from '@duckdb/duckdb-wasm';
```

```
const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: 'change/me/../duckdb-mvp.wasm',
        mainWorker: 'change/me/../duckdb-browser-mvp.worker.js',
    },
    eh: {
        mainModule: 'change/me/../duckdb-eh.wasm',
        mainWorker: 'change/me/../duckdb-browser-eh.worker.js',
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronous version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

Configuration

`instantiate()` starts DuckDB-Wasm with default settings. To change how the database behaves, call `open()` on the db object before opening a connection, passing a configuration object:

```
import * as duckdb from '@duckdb/duckdb-wasm';

await db.instantiate(bundle.mainModule, bundle.pthreadWorker);

await db.open({
    path: ':memory:',
    query: {
        castBigIntToDouble: true,
        castDecimalToDouble: true,
        castDurationToTime64: true,
        castTimestampToDate: true,
        queryPollingInterval: 1000,
    },
});

const conn = await db.connect();
```

Commonly used fields of the configuration object:

- `path`: the database file to open. Defaults to an in-memory database (`:memory:`). Use an `opfs://` path to persist data to the browser's Origin Private File System (see Persistence with OPFS).
- `accessMode`: `duckdb.DuckDBAccessMode.READ_ONLY` or `duckdb.DuckDBAccessMode.READ_WRITE`.
- `allowUnsignedExtensions`: allow loading extensions that are not signed (see Load Extensions).
- `maximumThreads`: the number of threads to use. This takes effect only with the threaded `coi` bundle on a cross-origin-isolated page (see Threading).
- `query`: controls how query results are converted from Arrow into JavaScript values.
- `opfs`: controls how `opfs://` files other than the database file are registered. Set `fileHandling` to `'auto'` to register files referenced in SQL automatically; the default is manual registration (see Persistence with OPFS).

The `query` object tunes the Arrow-to-JavaScript type mapping:

- `castBigIntToDouble`: return 64-bit integers as JavaScript numbers instead of `BigInt` values. This is convenient, but large integers can lose precision.
- `castDecimalToDouble`: return `DECIMAL` values as floating-point numbers instead of DuckDB's exact decimal representation.
- `castTimestampToDate`: return `TIMESTAMP` values as JavaScript `Date` objects.
- `castDurationToTime64`: return duration values as 64-bit (microsecond) time values.
- `queryPollingInterval`: the interval, in milliseconds, at which streaming queries poll for results.

Threading

By default DuckDB-Wasm runs on a single thread. The threaded `coi` bundle runs queries across multiple threads, but it has two requirements: the page must be cross-origin isolated, and you must add the `coi` bundle to the set you pass to `selectBundle`, because `getJsDelivrBundles()` returns only the `mvp` and `eh` bundles. The `coi` bundle also needs a third artifact, the pthread worker, alongside the usual module and worker:

```
import * as duckdb from '@duckdb/duckdb-wasm';

const DIST = 'https://cdn.jsdelivr.net/npm/@duckdb/duckdb-wasm/dist/';

// getJsDelivrBundles() returns only mvp and eh, so add coi explicitly
const bundles: duckdb.DuckDBBundles = {
    ...duckdb.getJsDelivrBundles(),
    coi: {
        mainModule: `${DIST}duckdb-coi.wasm`,
        mainWorker: `${DIST}duckdb-browser-coi.worker.js`,
        pthreadWorker: `${DIST}duckdb-browser-coi.pthread.worker.js`,
    },
};

// selectBundle picks coi only on a cross-origin-isolated page whose browser
// supports Wasm exceptions, SIMD, and threads; otherwise it falls back to eh or mvp
const bundle = await duckdb.selectBundle(bundles);

// Worker scripts must be same-origin, so wrap the CDN worker in a Blob URL
const worker_url = URL.createObjectURL(
    new Blob([`importScripts("${bundle.mainWorker}");`], { type: 'text/javascript' })
);
const worker = new Worker(worker_url);
const db = new duckdb.AsyncDuckDB(new duckdb.ConsoleLogger(), worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
URL.revokeObjectURL(worker_url);

// Configure the thread count (only effective on the coi bundle)
await db.open({ maximumThreads: 4 });
const conn = await db.connect();
```

Because `selectBundle` falls back to the `eh` or `mvp` bundle when the `coi` requirements are not met, the same code runs everywhere: on a page that is not cross-origin isolated it simply runs single-threaded, and `bundle.pthreadWorker` is `null`. Two things to keep in mind when deploying the threaded bundle:

- The page must be served with the `Cross-Origin-Opener-Policy: same-origin` and `Cross-Origin-Embedder-Policy: require-corp` headers, which is what makes the browser expose `SharedArrayBuffer` and report the page as cross-origin isolated. See Cross-Origin Isolation.
- Extensions must be built for the threaded platform. The `coi` bundle uses the `wasm_threads` extension platform, so extensions published only for `wasm_mvp` or `wasm_eh` will not load into it.

Persistence with OPFS

By default a DuckDB-Wasm database lives in memory and is lost when the page closes. To persist data across page reloads and sessions, open the database from the browser's Origin Private File System (OPFS) by giving `open()` a `path` with the `opfs://` scheme:

```
import * as duckdb from '@duckdb/duckdb-wasm';

await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
await db.open({
    path: 'opfs://duckdb.db',
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
});
```

```
const conn = await db.connect();
await conn.query(` CREATE TABLE t AS SELECT * FROM range(10) AS r(i)`);

// Flush changes to OPFS so they survive a reload
await conn.query(` CHECKPOINT`);
```

Reopening the same `opfs://` path in a later session restores the tables.

You can also read and write other `opfs://` files directly from SQL. By default, DuckDB-Wasm uses manual registration: every OPFS file other than the database file has to be registered with `db.registerOPFSFileName()` before a statement uses it, and dropped with `db.dropFile()` afterwards:

```
await db.open({
    path: 'opfs://duckdb.db',
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
});

// Register the file, use it from SQL, then release its handle
await db.registerOPFSFileName('opfs://export.parquet');
await conn.query(` COPY (SELECT * FROM t) TO 'opfs://export.parquet'`);
await db.dropFile('opfs://export.parquet');
```

Alternatively, set `opfs.fileHandling` to `'auto'`. DuckDB-Wasm then scans each statement for single-quoted `'opfs://...'` literals, registers those files before execution (creating them and any missing directories if needed) and drops their handles afterwards. This option only takes effect when the database itself was opened from an `opfs://` path:

```
await db.open({
    path: 'opfs://duckdb.db',
    accessMode: duckdb.DuckDBAccessMode.READ_WRITE,
    opfs: { fileHandling: 'auto' },
});

// Read a Parquet file stored in OPFS, and write a CSV back to OPFS
await conn.query(` CREATE TABLE t AS SELECT * FROM 'opfs://data.parquet'`);
await conn.query(` COPY (SELECT * FROM t) TO 'opfs://export.csv'`);
```

Automatic registration is convenient for one-off reads. Manual registration requires more code but avoids re-acquiring an OPFS access handle on every statement, which adds up for applications that run many small queries.

> **Note.** OPFS access relies on synchronous file access handles, which browsers expose only inside a Web Worker; DuckDB-Wasm's asynchronous API already runs in one. Keep these limitations in mind:
>
> - Call `CHECKPOINT` to flush writes to disk.
> - A file can be held by only one handle at a time. Drop registered files with `db.dropFile()` (or `db.dropFiles()`) before another connection or database instance opens them.
> - Moving files into or out of OPFS is not yet supported. Renaming files from SQL only works between two OPFS files that are both already registered.

Further Reading

- Import Data — registering files and inserting data into the instantiated database.
- Run Queries — executing queries against the db object created here.
- Deploy — serving the library, worker, and WebAssembly components that these bundles reference.
- DuckDB Wasm Client — the layered API and the examples the snippets above are drawn from.

Import Data

Overview

DuckDB-Wasm has multiple ways to import data, depending on the format of the data. Importing happens in two steps.

1. The data file is registered in a virtual file system using one of the register functions (registerFileBuffer, registerFileHandle, registerFileText, registerFileURL). These are called on the database object (db), because a registered file is visible to every connection.
2. The data is loaded into DuckDB using one of the insert functions (insertArrowFromIPCStream, insertArrowTable, insertCSVFromPath, insertJSONFromPath), which are called on a connection (conn). Alternatively, the Parquet, JSON, or Wasm-flavored httpfs extensions can read a registered — or remote — file directly from a FROM SQL query, and INSERT statements can load values inline.

The examples below all use a connection named conn, opened from the db object created when you instantiate DuckDB-Wasm.

Opening and Closing a Connection

Every insert and query runs on a connection. Open one with `connect()`, and close it when you are done to release memory:

```
// Create a new connection
const conn = await db.connect();

// ... import data

// Close the connection to release memory
await conn.close();
```

Apache Arrow

Apache Arrow is DuckDB-Wasm's native data protocol, so Arrow data can be inserted directly. Pass an existing `arrow.Table` to `insertArrowTable()`:

> **Warning.** Use the same major version of the `apache-arrow` package that DuckDB-Wasm depends on (currently `^17`). A newer major version causes Arrow inserts such as `insertArrowTable()` and `insertArrowFromIPCStream()` to fail — often silently, leaving the target table uncreated. Check DuckDB-Wasm's `apache-arrow` dependency and pin your `apache-arrow` install to match.

```
import { tableFromArrays } from 'apache-arrow';

const arrowTable = tableFromArrays({
  id: [1, 2, 3],
  name: ['John', 'Jane', 'Jack'],
  age: [20, 21, 22],
});

await conn.insertArrowTable(arrowTable, { name: 'arrow_table' });
```

To insert data that arrives as a raw Arrow IPC stream — for example, from a `fetch()` response — read each chunk and pass it to `insertArrowFromIPCStream()`, then write the end-of-stream (EOS) marker to signal that the table is complete:

```
// EOS signal according to the Arrow IPC streaming format
// See https://arrow.apache.org/docs/format/Columnar.html#ipc-streaming-format
const EOS = new Uint8Array([255, 255, 255, 255, 0, 0, 0, 0]);

const streamResponse = await fetch(` someapi`);
const streamReader = streamResponse.body.getReader();
const streamInserts = [];
while (true) {
    const { value, done } = await streamReader.read();
    if (done) break;
    streamInserts.push(conn.insertArrowFromIPCStream(value, { name: 'streamed' }));
}

// Write the EOS marker
streamInserts.push(conn.insertArrowFromIPCStream(EOS, { name: 'streamed' }));

await Promise.all(streamInserts);
```

CSV

Register the CSV text as a file, then load it with `insertCSVFromPath()`. The insert options describe the target table and, when auto-detection is disabled, the CSV dialect and column types:

```
import { Int32, Utf8 } from 'apache-arrow';

const csvContent = '1|foo\n2|bar\n';
await db.registerFileText('data.csv', csvContent);

await conn.insertCSVFromPath('data.csv', {
    schema: 'main',
    name: 'foo',
    detect: false,
    header: false,
    delimiter: '|',
    columns: {
        col1: new Int32(),
        col2: new Utf8(),
    },
});
```

JSON

`insertJSONFromPath()` accepts both row-major and column-major JSON. Register the document first, then insert it:

```
// Row-major format
const jsonRowContent = [
    { "col1": 1, "col2": "foo" },
    { "col1": 2, "col2": "bar" },
];
await db.registerFileText('rows.json', JSON.stringify(jsonRowContent));
await conn.insertJSONFromPath('rows.json', { name: 'rows' });

// Column-major format
const jsonColContent = {
    "col1": [1, 2],
    "col2": ["foo", "bar"]
};
await db.registerFileText('columns.json', JSON.stringify(jsonColContent));
await conn.insertJSONFromPath('columns.json', { name: 'columns' });
```

To import JSON fetched from an API, register the response bytes as a file buffer:

```
const streamResponse = await fetch(` someapi/content.json`);
await db.registerFileBuffer('file.json', new Uint8Array(await streamResponse.arrayBuffer()));
await conn.insertJSONFromPath('file.json', { name: 'JSONContent' });
```

Parquet

A Parquet file is registered rather than loaded through a dedicated insert function, then read with SQL. Register a local file picked by the user with `registerFileHandle()`, a remote file with `registerFileURL()`, or bytes you have already fetched with `registerFileBuffer()`:

```
// Local file chosen by the user
const pickedFile: File = letUserPickFile();
await db.registerFileHandle('local.parquet', pickedFile, DuckDBDataProtocol.BROWSER_FILEREADER, true);

// Remote file, read lazily over HTTP
await db.registerFileURL('remote.parquet', 'https://origin/remote.parquet', DuckDBDataProtocol.HTTP,
false);
```

```
// Bytes already fetched into memory
const res = await fetch('https://origin/remote.parquet');
await db.registerFileBuffer('buffer.parquet', new Uint8Array(await res.arrayBuffer()));
```

Once a file is registered, query it by its name:

```
await conn.query(` CREATE TABLE local AS SELECT * FROM 'local.parquet'`);
```

httpfs (Wasm-Flavored)

With the Wasm-flavored httpfs extension, you can skip registration and read a remote file directly by putting its URL in the SQL text:

```
await conn.query(`
    CREATE TABLE direct AS
        SELECT * FROM 'https://origin/remote.parquet'
`);
```

> **Tip.** Reading a remote file makes the browser issue the request, so it is subject to the browser's CORS policy. If a query over an S3 or other remote file fails with a network error, see Troubleshoot.

Insert Statement

Finally, `INSERT` statements load values directly, without registering a file — useful for small amounts of data or for appending to an existing table:

```
await conn.query(`
    INSERT INTO existing_table
    VALUES (1, 'foo'), (2, 'bar')`);
```

Further Reading

- Run Queries — querying the data imported here and exporting results.
- Load Extensions — the Parquet, JSON, and Wasm-flavored httpfs extensions used to read files directly in SQL.
- `INSERT` Statement — DuckDB's SQL-level `INSERT`, an alternative to the insert functions above.
- Instantiate — creating the db and connection these imports run on.
- Troubleshoot — CORS network errors when reading remote files.

Run Queries

Overview

DuckDB-Wasm provides functions for querying data. Queries are run sequentially:

1. First, create a connection by calling connect.
2. Then, run queries by calling query to materialize the whole result, or send to fetch result chunks lazily.

This page covers running queries, prepared statements, converting results to JSON, and exporting to Parquet.

Query Execution

Use `query()` when you want the entire result at once, or `send()` to iterate over Apache Arrow record batches as they are produced — useful for large results that you would rather stream than hold in memory all at once:

```
// Create a new connection
const conn = await db.connect();

// Either materialize the query result
await conn.query<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`);
// ..., or fetch the result chunks lazily
for await (const batch of await conn.send<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`)) {
    // ...
}

// Close the connection to release memory
await conn.close();
```

Prepared Statements

Prepared statements let you parse a query once and run it repeatedly with different parameter values, binding each value in place of a `?` placeholder. Both `query()` (materialized) and `send()` (streamed) are available on a prepared statement:

```
// Create a new connection
const conn = await db.connect();
// Prepare the query
const stmt = await conn.prepare(` SELECT v + ? FROM generate_series(0, 10_000) t(v);`);
// ... and run it with materialized results
await stmt.query(234);
// ... or as result chunks
for await (const batch of await stmt.send(234)) {
    // ...
}
// Close the statement to release memory
await stmt.close();
// Closing the connection releases its statements as well
await conn.close();
```

Arrow Table to JSON

Query results are returned as Arrow tables. To convert a result into plain JavaScript objects, map over the table's rows and call `toJSON()` on each:

```
// Create a new connection
const conn = await db.connect();

// Query the data
const arrowResult = await conn.query<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`);

// Convert the Arrow table to an array of JSON objects
const result = arrowResult.toArray().map((row) => row.toJSON());

// Close the connection to release memory
await conn.close();
```

Export Parquet

To export a query result, write it to a Parquet file in the virtual file system with `COPY`, then copy that file's bytes into a buffer you can download or process:

```
// Create a new connection
const conn = await db.connect();

// Export the query result to a Parquet file in the virtual file system
await conn.query(` COPY (SELECT * FROM tbl) TO 'result-snappy.parquet' (FORMAT parquet);`);

// Copy the file contents into a buffer
const parquetBuffer = await db.copyFileToBuffer('result-snappy.parquet');

// Generate a download link
const link = URL.createObjectURL(new Blob([parquetBuffer]));

// Close the connection to release memory
await conn.close();
```

Further Reading

- Import Data — loading the data that these queries read.
- Load Extensions — the Parquet extension used by the export example and other query-time extensions.
- Prepared Statements — DuckDB's SQL-level support for the parameterized queries shown here.
- Instantiate — creating the db that connections are opened on.
- Troubleshoot — streaming results with `send()` to stay within the memory limit.

Load Extensions

Overview

DuckDB-Wasm's (dynamic) extension loading is modeled after regular DuckDB's extension loading, with a few relevant differences due to the difference in platform. A growing subset of core, community, and external extensions is supported.

This page explains the extension format, how `INSTALL` and `LOAD` behave, autoloading, which extensions are available, signing, and how extensions are fetched and served.

Format

Extensions in DuckDB are binaries to be dynamically loaded via `dlopen`. A cryptographical signature is appended to the binary.

An extension in DuckDB-Wasm is a regular Wasm file to be dynamically loaded via Emscripten's `dlopen`. A cryptographical signature is appended to the Wasm file as a WebAssembly custom section called `duckdb_signature`. This ensures the file remains a valid WebAssembly file.

> Currently, this custom section is required to be the last one, but this requirement may be relaxed in the future.

`INSTALL` and `LOAD`

The `INSTALL` semantic in native embeddings of DuckDB is to fetch, decompress from `gzip` and store data in local disk. The `LOAD` semantic in native embeddings of DuckDB is to (optionally) perform signature checks *and* dynamic load the binary with the main DuckDB binary.

In DuckDB-Wasm, `INSTALL` performs no durable installation, because there is no cross-session storage; it only records where an extension will be fetched from (for example, with `INSTALL <name> FROM community`), deferring the actual fetch to `LOAD`. The `LOAD` operation will fetch (and decompress on the fly), perform signature checks *and* dynamically load via the Emscripten implementation of `dlopen`.

Autoloading

Autoloading, i.e., the possibility for DuckDB to add extension functionality on-the-fly, is enabled by default in DuckDB-Wasm.

Available Extensions

A growing subset of extensions is supported for DuckDB-Wasm, spanning core extensions, community extensions, and external extensions. Most are fetched on the fly when they are autoloaded or explicitly loaded, rather than being bundled into the DuckDB-Wasm binary. For example:

```
-- Explicitly load a core extension
LOAD icu;

-- Load spatial for geospatial support
INSTALL spatial;
LOAD spatial;

-- Install a community extension, then load it
INSTALL h3 FROM community;
LOAD h3;

-- Install an external extension from a repository
INSTALL sqlite_scanner FROM 'https://extensions.duckdb.org';
LOAD sqlite_scanner;
```

The core extensions commonly used with DuckDB-Wasm include:

Extension name	Description	Aliases
autocomplete	Adds support for autocomplete in the shell	
excel	Adds support for Excel-like format strings	
fts	Adds support for Full-Text Search indexes	
icu	Adds support for time zones, calendars and collations	
inet	Adds support for IP-related data types and functions	
json	Adds support for JSON operations	
parquet	Adds support for reading and writing Parquet files	
spatial	Adds support for geospatial data types and functions	
sqlite	Adds support for reading SQLite database files	sqlite, sqlite3
tpcds	Adds TPC-DS data generation and query support	
tpch	Adds TPC-H data generation and query support	

WebAssembly is essentially an additional platform, and there may be platform-specific limitations that prevent some extensions from matching their native capabilities or that make them behave differently. Relevant differences for DuckDB-hosted extensions are documented below.

HTTPFS

The native HTTPFS extension is not compiled into DuckDB-Wasm, because HTTPS capabilities have to go through an additional layer, the browser, which adds both differences and some restrictions relative to native. Instead, DuckDB-Wasm ships a separate JavaScript implementation of the same functionality. Running `LOAD httpfs;` opts into this re-implemented HTTP stack, which is interchangeable with the native extension for most purposes but does not support all use cases, as it must follow the security rules imposed by the browser.

In particular:

- Requests are always upgraded to HTTPS.

- Because of the browser's CORS policy, any request for data must target a site that allows (using CORS headers) the site hosting the DuckDB-Wasm instance to access that data.

The MDN website is a great resource for more information regarding CORS. If a query over a remote file fails with a network error, see Troubleshoot.

Extension Signing

As with regular DuckDB extensions, DuckDB-Wasm extensions are by default checked on `LOAD` to verify the signature and confirm the extension has not been tampered with. Extension signature verification can be disabled via a configuration option.

Signing is a property of the binary itself, so copying a DuckDB extension (say to serve it from a different location) will still keep a valid signature (e.g., for local development).

Fetching DuckDB-Wasm Extensions

Official DuckDB extensions are served at `extensions.duckdb.org`, which is used as the extension repository unless `custom_extension_repository` is set. When installing extensions, a relevant URL will be built that will look like `extensions.duckdb.org/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.gz`.

DuckDB-Wasm extension are fetched only on load, and the URL will look like: `extensions.duckdb.org/duckdb-wasm/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.wasm`.

Note that an additional `duckdb-wasm` is added to the folder structure, and the file is served as a `.wasm` file.

DuckDB-Wasm extensions are served pre-compressed using Brotli compression. While fetched from a browser, extensions will be transparently uncompressed. If you want to fetch the `duckdb-wasm` extension manually, you can use `curl --compressed extensions.duckdb.org/<...>/icu.duckdb_extension.wasm`.

Serving Extensions from a Third-Party Repository

As with regular DuckDB, if you use `SET custom_extension_repository = 'https://some.url.com'`{:.language-sql .highlight}, subsequent loads will be attempted at `https://some.url.com/duckdb-wasm/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.wasm`.

Note that `GET` requests for the extensions must be CORS enabled for a browser to allow the connection; see Troubleshoot if a load fails.

Tooling

Both DuckDB-Wasm and its extensions have been compiled using the latest packaged Emscripten toolchain.

Further Reading

- Deploy — serving and mirroring extensions from a custom endpoint.
- Core Extensions — the core extensions available across all DuckDB clients.
- Community Extensions — third-party extensions installable with `INSTALL … FROM community`.
- Import Data — using the Parquet, JSON, and Wasm-flavored httpfs extensions to read files.
- Troubleshoot — CORS errors when loading extensions from a custom repository.

Deploy

Overview

Deploying DuckDB-Wasm means serving its components so that the browser can fetch and instantiate them. This page describes each component a deployment needs to serve, how the worker and WebAssembly variants relate, how extensions are served and mirrored, and the security considerations involved.

A DuckDB-Wasm deployment needs to access the following components:

- **Main Library Component.** Distributed as TypeScript and compiled to JavaScript code. For further details, see below.
- **JS Worker Component.** Compiled to JavaScript code, possibly instantiated multiple times for threaded environments. For further details, see below.
- **Wasm Worker Component.** Compiled as a WebAssembly file and instantiated by the browser. For further details, see below.
- **DuckDB Extensions.** Any relevant extensions the deployment loads. For further details, see below.

Main Library Component

This component is distributed as TypeScript or CommonJS JavaScript in the npm `@duckdb/duckdb-wasm` package. You can deploy it in one of three ways:

- Bundled together with your application
- Served from a same-origin (sub-)domain and included at runtime
- Served from a third-party CDN such as jsDelivr

It cannot be served as-is: it requires transpilation, because it needs to know the location of the follow-up files — the worker and WebAssembly components — in order to work. The exact steps depend on your setup; see the examples in the repository. For instance, the shell.duckdb.org deployment transpiles the main library together with the shell code (the first approach above), while the `bare-browser` example shows a minimal setup.

JS Worker Component

This is distributed as a JavaScript file in three different flavors, `mvp`, `eh` and `coi`, and needs to be served as-is. The main library component needs to be informed of the actual location.

The three variants target three different WebAssembly feature sets:

- `mvp` targets the WebAssembly 1.0 (MVP) spec
- `eh` targets WebAssembly with Wasm-level exception handling added, which improves performance
- `coi` targets WebAssembly with exception handling and threading, which enables parallel query execution; it requires the deployment to be cross-origin isolated (see Cross-Origin Isolation)

You can serve all three and let the library feature-detect the best one with `selectBundle`, or serve a single variant and instruct the DuckDB-Wasm library on which one to use. The served artifacts are named after the flavor, for example `duckdb-browser-coi.worker.js`. The `coi` flavor corresponds to the `wasm_threads` extension platform.

Wasm Worker Component

This is the DuckDB engine itself, compiled to WebAssembly and instantiated by the browser. Like the JS Worker component, it comes in the same three flavors, each paired with the JS worker of the same flavor:

- `mvp` — for example `duckdb-mvp.wasm`, loaded by the `mvp` JS worker
- `eh` — for example `duckdb-eh.wasm`, loaded by the `eh` JS worker
- `coi` — for example `duckdb-coi.wasm`, loaded by the `coi` JS worker

Each JS worker loads the WebAssembly module of its own flavor, so serve the module (or modules) that correspond to the JS workers you deploy. If you serve all three JS workers and let `selectBundle` feature-detect the best one, serve all three modules as well.

When serving these files, keep the following in mind:

- Serve them as-is. Unlike the main library component, they require no transpilation or bundling.
- Serve them with the `application/wasm` content type so the browser can compile them efficiently.
- Host them anywhere reachable from the main document — the same origin or an arbitrary [sub-]domain. The main library component is told each module's location through the bundle definition (the `mainModule` path), so the modules do not need to sit next to the JavaScript.

DuckDB Extensions

DuckDB extensions for DuckDB-Wasm, similar for the native cases, are served signed at the default extension endpoint: `https://extensions.duckdb.org`. If you are deploying duckdb-wasm you can consider mirroring relevant extensions at a different endpoint, possibly allowing for air-tight deployments on internal networks.

```
SET custom_extension_repository = '<https://some.endpoint.org/path/to/repository>';
```

Changes the default extension repository from the public `https://extensions.duckdb.org` to the one specified. Note that extensions are still signed, so the best path is downloading and serving the extensions with a similar structure to the original repository. See the additional notes on Creating a Custom Repository.

Community extensions are served at https://community-extensions.duckdb.org, and they are signed with a different key, so they can be disabled with a one way SQL statement such as:

```
SET allow_community_extensions = false;
```

This will allow loading **only** of core duckdb extensions. Note that the failure is at `LOAD` time, not at `INSTALL` time.

Please review the Extension Distribution page for general information about extensions.

Cross-Origin Isolation

The `coi` bundle runs multiple threads, which relies on `SharedArrayBuffer`. Browsers only expose `SharedArrayBuffer` to pages that are cross-origin isolated. To serve the threaded bundle, the top-level document must be delivered with the following HTTP headers:

```
Cross-Origin-Embedder-Policy: require-corp
Cross-Origin-Opener-Policy: same-origin
```

These headers isolate the document from other cross-origin documents and require any cross-origin resource it embeds to explicitly opt in. Because many third-party endpoints do not yet send the headers needed to be embedded under these policies, most deployments run on non-isolated pages, where DuckDB-Wasm falls back to the single-threaded `mvp` or `eh` bundle.

For more background on why these headers are required and how they unlock `SharedArrayBuffer`, see the following resources:

- WebAssembly threads on web.dev
- Enabling `SharedArrayBuffer` on the Chrome developer blog

Security Considerations

> **Warning.** Deploying DuckDB-Wasm with access to your own data means whoever has access to SQL can access the data that DuckDB-Wasm can access. Also, DuckDB-Wasm in the default setting can access remote endpoints, so it can have a visible effect on the external world even from within the sandbox.

Further Reading

- Instantiate — how the library, worker, and WebAssembly components served here are wired together at runtime.
- Load Extensions — how extensions are fetched, signed, and served from a custom repository.
- Extension Distribution — general information about extension repositories and creating a custom one.
- DuckDB Wasm Client — the layered API and example deployments this page builds on.
- Troubleshoot — cross-origin isolation for threading and other deployment-related issues.

Troubleshoot

Overview

This page collects common issues encountered when using DuckDB-Wasm, together with their workarounds. If you run into a problem that is not covered here, search the `duckdb-wasm` issue tracker on GitHub.

Network Error When Querying Remote Files

When DuckDB-Wasm reads a remote file — through the Wasm-flavored httpfs extension or `registerFileURL` — the browser issues the request, so it is subject to the browser's CORS policy and every request is upgraded to HTTPS.

The server hosting the data must allow (using CORS headers) the origin that hosts the DuckDB-Wasm instance to read it. If the server does not send the required headers, the query fails with a network error such as:

```
Failed to execute 'send' on 'XMLHttpRequest'
```

For example, when querying files from Amazon S3, configure the bucket's CORS policy to allow `GET` and `HEAD` requests:

```
[
    {
        "AllowedHeaders": ["*"],
        "AllowedMethods": ["GET", "HEAD"],
        "AllowedOrigins": ["*"],
        "ExposeHeaders": [],
        "MaxAgeSeconds": 3000
    }
]
```

This array is a server-side bucket configuration (for Amazon S3, the bucket's CORS policy), not a value you pass to DuckDB-Wasm. There is no db or connection option and no `DuckDBGlobalFileInfo` field for it, because the browser, not DuckDB-Wasm, enforces CORS. Apply it where the data is hosted, then read the file as usual.

Beyond CORS, reading remote files has several other known limitations:

- Because every request is upgraded to HTTPS, an `http://` URL can be rewritten incorrectly (for example to `https://http//…`), which breaks reads against local or non-TLS endpoints (see duckdb-wasm issue #2118).
- DuckDB relies on HTTP range requests to read only part of a file. Range requests can fail on Firefox (see issue #1932) and are not detected correctly for some S3 pre-signed URLs (see issue #2228), which forces the whole file to be downloaded or makes the read fail.
- Cached reads of files in DuckDB's native format over HTTP can be corrupted (see issue #1957).
- HTTP authentication headers are not supported, so endpoints that require them cannot be read directly (see issue #1967).

Extension Fails to Load from a Custom Repository

DuckDB-Wasm fetches each extension over the network when it is loaded. If you serve extensions from a custom repository with `SET custom_extension_repository = '<https://some.url.com>'`{:.language-sql .highlight}, the `GET` requests for the extension files must be CORS enabled for the browser to allow the connection, exactly as for remote data files above.

Extensions remain signed regardless of where they are served, so copying an extension to a different location keeps its signature valid.

Out of Memory

The WebAssembly client has a limited amount of memory available: WebAssembly limits the available memory to 4 GB and browsers may impose even stricter limits. Queries over large datasets can therefore run out of memory where native DuckDB would not. To stay within the limit:

- Stream large results with `send()` instead of materializing them all at once with `query()`.
- Read only the columns and rows you need. Because DuckDB reads Parquet files lazily, projections, filters, and `LIMIT` clauses let entire row groups be skipped rather than loaded into memory.

See Limitations for the other constraints of the WebAssembly client.

Threading Is Not Available

By default, the WebAssembly client uses a single thread; multithreading is available through the `coi` bundle but is still experimental.

The threaded bundle relies on `SharedArrayBuffer`, which browsers only expose to pages that are cross-origin isolated. If the top-level document is not served with the `Cross-Origin-Embedder-Policy` and `Cross-Origin-Opener-Policy` headers, DuckDB-Wasm falls back to the single-threaded `mvp` or `eh` bundle.

See Cross-Origin Isolation for the required headers, and Threading for how to select and configure the `coi` bundle.

Further Reading

- DuckDB Wasm Client — installation, basic usage, and the limitations behind several issues above.
- Import Data — registering and reading remote files that are subject to the CORS rules above.
- Load Extensions — how extensions are fetched, signed, and served, including the httpfs CORS behavior.
- Deploy — serving components and configuring cross-origin isolation for threading.

Tertiary Clients

Tertiary Clients

The following table lists the **tertiary clients** of DuckDB. Tertiary clients come without any support guarantees.

Client API	Maintainer
Common Lisp	ak-coram
Crystal	amauryt
Elixir (Duckdbex)	AlexR2D2
Elixir (QuackDB)	Danila Poyarkov
Erlang	MM Zeeman
Haskell	Tritlo
Julia	The DuckDB team
Perl	Giuseppe Di Terlizzi
PHP	satur-io
PHP (PDO)	Thomas Bley
Pyodide	The DuckDB team
Raku	bduggan
Ruby	suketa
Scala	Salar Rahmanian
Swift	The DuckDB team
Zig	karlseguin

Julia Client

The DuckDB Julia package provides a high-performance front-end for DuckDB. Much like SQLite, DuckDB runs in-process within the Julia client, and provides a `DBInterface` front-end.

The package also supports multi-threaded execution. It uses Julia threads/tasks for this purpose. If you wish to run queries in parallel, you must launch Julia with multi-threading support (by e.g., setting the `JULIA_NUM_THREADS` environment variable).

Installation

Install DuckDB as follows:

```
using Pkg
Pkg.add("DuckDB")
```

Alternatively, enter the package manager using the] key, and issue the following command:

```
pkg> add DuckDB
```

Basics

```julia
using DuckDB

# create a new in-memory database
con = DBInterface.connect(DuckDB.DB, ":memory:")

# create a table
DBInterface.execute(con, "CREATE TABLE integers (i INTEGER)")

# insert data by executing a prepared statement
stmt = DBInterface.prepare(con, "INSERT INTO integers VALUES (?)")
DBInterface.execute(stmt, [42])

# query the database
results = DBInterface.execute(con, "SELECT 42 a")
print(results)
```

Some SQL statements, such as PIVOT and IMPORT DATABASE are executed as multiple prepared statements and will error when using `DuckDB.execute()`. Instead they can be run with `DuckDB.query()` instead of `DuckDB.execute()` and will always return a materialized result.

Scanning DataFrames

The DuckDB Julia package also provides support for querying Julia DataFrames. Note that the DataFrames are directly read by DuckDB – they are not inserted or copied into the database itself.

If you wish to load data from a DataFrame into a DuckDB table you can run a `CREATE TABLE ... AS` or `INSERT INTO` query.

```julia
using DuckDB
using DataFrames

# create a new in-memory database
con = DBInterface.connect(DuckDB.DB)

# create a DataFrame
df = DataFrame(a = [1, 2, 3], b = [42, 84, 42])

# register it as a view in the database
DuckDB.register_data_frame(con, df, "my_df")

# run a SQL query over the DataFrame
results = DBInterface.execute(con, "SELECT * FROM my_df")
print(results)
```

Appender API

The DuckDB Julia package also supports the Appender API, which is much faster than using prepared statements or individual `INSERT INTO` statements. Appends are made in row-wise format. For every column, an `append()` call should be made, after which the row should be finished by calling `end_row()`. After all rows have been appended, `close()` should be used to finalize the Appender and clean up the resulting memory.

```julia
using DuckDB, DataFrames, Dates
db = DuckDB.DB()
# create a table
DBInterface.execute(db,
    "CREATE OR REPLACE TABLE data (id INTEGER PRIMARY KEY, value FLOAT, timestamp TIMESTAMP, date DATE)")
# create data to insert
len = 100
df = DataFrames.DataFrame(
        id = collect(1:len),
```

```
        value = rand(len),
        timestamp = Dates.now() + Dates.Second.(1:len),
        date = Dates.today() + Dates.Day.(1:len)
    )
# append data by row
appender = DuckDB.Appender(db, "data")
for i in eachrow(df)
    for j in i
        DuckDB.append(appender, j)
    end
    DuckDB.end_row(appender)
end
# close the appender after all rows
DuckDB.close(appender)
```

Concurrency

Within a Julia process, tasks are able to concurrently read and write to the database, as long as each task maintains its own connection to the database. In the example below, a single task is spawned to periodically read the database and many tasks are spawned to write to the database using both INSERT statements as well as the Appender API.

```
using Dates, DataFrames, DuckDB
db = DuckDB.DB()
DBInterface.connect(db)
DBInterface.execute(db, "CREATE OR REPLACE TABLE data (date TIMESTAMP, id INTEGER)")

function run_reader(db)
    # create a DuckDB connection specifically for this task
    conn = DBInterface.connect(db)
    while true
        println(DBInterface.execute(conn,
                "SELECT id, count(date) AS count, max(date) AS max_date
                FROM data GROUP BY id ORDER BY id") |> DataFrames.DataFrame)
        Threads.sleep(1)
    end
    DBInterface.close(conn)
end
# spawn one reader task
Threads.@spawn run_reader(db)

function run_inserter(db, id)
    # create a DuckDB connection specifically for this task
    conn = DBInterface.connect(db)
    for i in 1:1000
        Threads.sleep(0.01)
        DuckDB.execute(conn, "INSERT INTO data VALUES (current_timestamp, ?)"; id);
    end
    DBInterface.close(conn)
end
# spawn many insert tasks
for i in 1:100
    Threads.@spawn run_inserter(db, 1)
end

function run_appender(db, id)
    # create a DuckDB connection specifically for this task
    appender = DuckDB.Appender(db, "data")
    for i in 1:1000
        Threads.sleep(0.01)
        row = (Dates.now(Dates.UTC), id)
        for j in row
            DuckDB.append(appender, j);
        end
```

```
        DuckDB.end_row(appender);
    end
    DuckDB.close(appender);
end
# spawn many appender tasks
for i in 1:100
    Threads.@spawn run_appender(db, 2)
end
```

Original Julia Connector

Credits to kimmolinna for the original DuckDB Julia connector.

PHP Client

> The DuckDB PHP client is a tertiary client and is maintained by a third-party.

Client API for PHP, focused on performance. The DuckDB PHP client uses the official C API internally through FFI, achieving good benchmark results. This library is more than just a wrapper for the C API; it introduces custom, PHP-friendly methods to simplify working with DuckDB. It is compatible with Linux, Windows and macOS, requiring PHP version 8.3 or higher.

Full documentation is available at https://duckdb-php.readthedocs.io/.

Automatic Install (Recommended for Newcomers)

```
composer require satur.io/duckdb-auto
```

You will need to allow `satur.io/duckdb-auto` to execute code to use this installation method, check installation for more details.

Quick Start

```
DuckDB::sql("SELECT 'quack' as my_column")->print();
```

```
--------------------
| my_column        |
--------------------
| quack            |
--------------------
```

The function we used here, `DuckDB::sql()`, performs the query in a new in-memory database which is destroyed after retrieving the result.

This is not the most common use case, let's see how to get a persistent connection.

Connection

```
$duckDB = DuckDB::create('duck.db'); // or DuckDB::create() for in-memory database

$duckDB->query('CREATE TABLE test (i INTEGER, b BOOL, f FLOAT);');
$duckDB->query('INSERT INTO test VALUES (3, true, 1.1), (5, true, 1.2), (3, false, 1.1), (3, null, 1.2);');

$duckDB->query('SELECT * FROM test')->print();
```

As you probably guessed, `DuckDB::create()` creates a new connection to the specified database, or creates a new one if it doesn't exist yet and then establishes the connection.

After that, we can use the function `query` to perform the requests.

> Notice the difference between the static method `sql` and the non-static method `query`. While the first one always creates and destroys a new in-memory database, the second one uses a previously established connection and should be the preferred option in most cases.

In addition, the library also provides prepared statements for binding parameters to our query.

Prepared Statements

```
$duckDB = DuckDB::create();

$duckDB->query('CREATE TABLE test (i INTEGER, b BOOL, f FLOAT);');
$duckDB->query('INSERT INTO test VALUES (3, true, 1.1), (5, true, 1.2), (3, false, 1.1), (3, null, 1.2);');

$boolPreparedStatement = $duckDB->preparedStatement('SELECT * FROM test WHERE b = $1');
$boolPreparedStatement->bindParam(1, true);
$result = $boolPreparedStatement->execute();
$result->print();

$intPreparedStatement = $duckDB->preparedStatement('SELECT * FROM test WHERE i = ?');
$intPreparedStatement->bindParam(1, 3);
$result = $intPreparedStatement->execute();
$result->print();
```

Appenders

Appenders are the preferred method to load data in DuckDB. See Appender page for more information.

```
$duckDB = DuckDB::create();
$result = $duckDB->query('CREATE TABLE people (id INTEGER, name VARCHAR);');

$appender = $duckDB->appender('people');

for ($i = 0; $i < 100; ++$i) {
    $appender->append(rand(1, 100000));
    $appender->append('string-'.rand(1, 100));
    $appender->endRow();
}

$appender->flush();
```

DuckDB-Powerful

DuckDB provides some amazing features. For example, you can query remote files directly.

Let's use an aggregate function to calculate the average of a column for a Parquet remote file:

```
DuckDB::sql(
    'SELECT "Reporting Year", avg("Gas Produced, MCF") as "AVG Gas Produced"
    FROM "https://github.com/plotly/datasets/raw/refs/heads/master/oil-and-gas.parquet"
    WHERE "Reporting Year" BETWEEN 1985 AND 1990
    GROUP BY "Reporting Year";'
)->print();
```

```
---------------------------------------
| Reporting Year   | AVG Gas Produce |
---------------------------------------
| 1985             | 2461.4047344111 |
| 1986             | 6060.8575605681 |
| 1987             | 5047.5813074014 |
| 1988             | 4763.4090541633 |
| 1989             | 4175.2989758837 |
```

```
| 1990             | 3706.9404742437 |
-------------------------------------
```

Or summarize a remote CSV:

```
DuckDB::sql('SUMMARIZE TABLE "https://blobs.duckdb.org/data/Star_Trek-Season_1.csv";')->print();

-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
----------------
| column_name      | column_type      | min             | max             | approx_unique    | avg
| std              | q25              | q50             | q75             | count            | null_percentage |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
----------------
| season_num       | BIGINT           | 1               | 1               | 1                | 1.0
| 0.0              | 1                | 1               | 1               | 30               | 0               |
| episode_num      | BIGINT           | 0               | 29              | 29               | 14.5
| 8.8034084308295  | 7                | 14              | 22              | 30               | 0               |
| aired_date       | DATE             | 1965-02-28      | 1967-04-13      | 35               |
|                  | 1966-10-20       | 1966-12-22      | 1967-02-16      | 30               | 0               |
| cnt_kirk_hookup  | BIGINT           | 0               | 2               | 3                | 0.3333333333333
| 0.6064784348631  | 0                | 0               | 1               | 30               | 0               |
...
```

Requirements

- Linux, macOS or Windows.
- x64 platform.
- PHP >= 8.3.
- ext-ffi.

Recommended

- ext-bcmath – Needed for big integers (> PHP_INT_MAX).
- ext-zend-opcache – For better performance.

Type Support

From version 1.2.0 on the library supports all DuckDB file types.

DuckDB Type	SQL Type	PHP Type
DUCKDB_TYPE_BOOLEAN	BOOLEAN	bool
DUCKDB_TYPE_TINYINT	TINYINT	int
DUCKDB_TYPE_SMALLINT	SMALLINT	int
DUCKDB_TYPE_INTEGER	INTEGER	int
DUCKDB_TYPE_BIGINT	BIGINT	int
DUCKDB_TYPE_UTINYINT	UTINYINT	int
DUCKDB_TYPE_USMALLINT	USMALLINT	int
DUCKDB_TYPE_UINTEGER	UINTEGER	int
DUCKDB_TYPE_UBIGINT	UBIGINT	Saturio\DuckDB\Type\Math\LongInteger
DUCKDB_TYPE_FLOAT	FLOAT	float

DuckDB Type	SQL Type	PHP Type
DUCKDB_TYPE_DOUBLE	DOUBLE	float
DUCKDB_TYPE_TIMESTAMP	TIMESTAMP	Saturio\DuckDB\Type\Timestamp
DUCKDB_TYPE_DATE	DATE	Saturio\DuckDB\Type\Date
DUCKDB_TYPE_TIME	TIME	Saturio\DuckDB\Type\Time
DUCKDB_TYPE_INTERVAL	INTERVAL	Saturio\DuckDB\Type\Interval
DUCKDB_TYPE_HUGEINT	HUGEINT	Saturio\DuckDB\Type\Math\LongInteger
DUCKDB_TYPE_UHUGEINT	UHUGEINT	Saturio\DuckDB\Type\Math\LongInteger
DUCKDB_TYPE_VARCHAR	VARCHAR	string
DUCKDB_TYPE_BLOB	BLOB	Saturio\DuckDB\Type\Blob
DUCKDB_TYPE_TIMESTAMP_S	TIMESTAMP_S	Saturio\DuckDB\Type\Timestamp
DUCKDB_TYPE_TIMESTAMP_MS	TIMESTAMP_MS	Saturio\DuckDB\Type\Timestamp
DUCKDB_TYPE_TIMESTAMP_NS	TIMESTAMP_NS	Saturio\DuckDB\Type\Timestamp
DUCKDB_TYPE_UUID	UUID	Saturio\DuckDB\Type\UUID
DUCKDB_TYPE_TIME_TZ	TIMETZ	Saturio\DuckDB\Type\Time
DUCKDB_TYPE_TIMESTAMP_TZ	TIMESTAMPTZ	Saturio\DuckDB\Type\Timestamp
DUCKDB_TYPE_DECIMAL	DECIMAL	float
DUCKDB_TYPE_ENUM	ENUM	string
DUCKDB_TYPE_LIST	LIST	array
DUCKDB_TYPE_STRUCT	STRUCT	array
DUCKDB_TYPE_ARRAY	ARRAY	array
DUCKDB_TYPE_MAP	MAP	array
DUCKDB_TYPE_UNION	UNION	mixed
DUCKDB_TYPE_BIT	BIT	string
DUCKDB_TYPE_BIGNUM	BIGNUM	string
DUCKDB_TYPE_SQLNULL	NULL	null

PHP Client (PDO)

> The PHP DuckDB PDO extension is a tertiary client and is maintained by a third-party.
>
> To use pdo_duckdb, follow the instructions below.
>
> The latest version of pdo_duckdb supports DuckDB 2.0.0.

pdo_duckdb is a native DuckDB database driver for the PHP Data Objects (PDO) interface.

Any application or framework compatible with PDO can directly use pdo_duckdb.\ It is thread safe and fully tested with php-cli, php-fpm, mod_php, FrankenPHP (PHP-ZTS), TrueAsync and Swoole.

As a native PHP extension, it is implemented in C/C++ and does not require PHP FFI or preloading.\ The release packages contain pre-compiled binaries for all supported platforms and DuckDB is directly included.\ DuckDB extensions work the same way as they do in DuckDB CLI.

Supported PHP versions: 8.2 or newer

Operating systems: Ubuntu 22+, Debian 12+, Fedora 42+, openSUSE 16+, AmazonLinux 2023+, Arch, Alma, Rocky, Alpine, Wolfi OS, Windows Server 2022+ (x64) and macOS 15+ (arm64)

All DuckDB types are supported:\ Text, Numeric, Date, Time, Interval, JSON, Array, Struct, Map, List, Enum, Variant, Geometry, Union, Bit-string, Blob and Boolean

GitHub: pdo-duckdb-php

Packagist: pdo-duckdb-php

Installation and setup

pdo_duckdb is fully compatible with the PHP Installer for Extensions (PIE) and can be installed with:

```
# DuckDB 1.x
pie install thomas-0816/pdo-duckdb-php:^1
# DuckDB 2.x
pie install thomas-0816/pdo-duckdb-php:^2-beta
```

Testing:

```
php -m | grep duckdb
php -r 'print_r((new PDO("duckdb::memory:"))->query("SELECT 42 as n")->fetch(PDO::FETCH_ASSOC));'

# Array
# (
#     [n] => 42
# )
```

Installation and load on demand

```
# DuckDB 1.x
pie install --skip-enable-extension thomas-0816/pdo-duckdb-php:^1
# DuckDB 2.x
pie install --skip-enable-extension thomas-0816/pdo-duckdb-php:^2-beta
```

Testing:

```
php -d extension=pdo_duckdb -m | grep duckdb
php -d extension=pdo_duckdb -r 'print_r((new PDO("duckdb::memory:"))->query("SELECT 42 as
n")->fetch(PDO::FETCH_ASSOC));'
```

Integration into Laravel & Symfony

Laravel Eloquent driver for DuckDB: laravel-duckdb

Doctrine DBAL for DuckDB: doctrine-dbal-duckdb

Examples

Basic usage

```
<?php

$duckDb = new PDO('duckdb::memory:');
$duckDb->exec('CREATE TABLE table1 (id INTEGER, amount DECIMAL(10, 2), description VARCHAR)');

$statement = $duckDb->prepare('INSERT INTO table1 VALUES (?, ?, ?)');
$statement->execute([1, 42.21, 'Hello DuckDB! 🦆']);
```

```
$statement = $duckDb->query('SELECT * FROM table1');
print_r($statement->fetchAll(PDO::FETCH_ASSOC));

# Array
# (
#     [0] => Array
#         (
#             [id] => 1
#             [amount] => 42.21
#             [description] => Hello DuckDB! 🦆
#         )
# )
```

Open databases from disk or in-memory

```
<?php

$db = new PDO('duckdb::memory:'); // open in-memory database

$db = new PDO('duckdb:/tmp/test.db'); // open database file from disk

// open database file as read-only
$db = new PDO('duckdb:/tmp/test.db', null, null, [PDO::DUCKDB_ATTR_CONFIG => ['access_mode' => 'read_
only']]);
```

Read and write Parquet files

```
<?php

$db = new PDO('duckdb::memory:');
$db->exec('CREATE TABLE table1 (id INTEGER, text VARCHAR USING COMPRESSION zstd, data JSON)');

$statement = $db->prepare('INSERT INTO table1 VALUES (?, ?, ?)');
$statement->execute([1, 'Hello DuckDB 🦆', ['foo' => 'bar', 'baz' => 42]]);

$db->exec("COPY (SELECT * FROM table1) TO '/tmp/table1.parquet' (COMPRESSION zstd)");

foreach ($db->query("SELECT * FROM '/tmp/table1.parquet'", PDO::FETCH_ASSOC) as $row) {
    print_r($row);
}

# Array
# (
#     [id] => 1
#     [text] => Hello DuckDB 🦆
#     [data] => Array
#         (
#             [foo] => bar
#             [baz] => 42
#         )
# )
```

Read CSV files

```
<?php

$list = [
    ['aaa', 'bbb', 'ccc'],
    ['123', '456', '789'],
    ['aaa', 'bbb', 'ccc']
];
$fp = fopen('/tmp/test.csv', 'w');
```

```php
foreach ($list as $fields) {
    fputcsv($fp, $fields, ',', '"', '');
}
fclose($fp);

$db = new PDO('duckdb::memory:');
$statement = $db->query("SELECT * FROM '/tmp/test.csv'");
print_r($statement->fetchAll(PDO::FETCH_ASSOC));

# Array
# (
#     [0] => Array
#         (
#             [aaa] => 123
#             [bbb] => 456
#             [ccc] => 789
#         )
#     [1] => Array
#         (
#             [aaa] => aaa
#             [bbb] => bbb
#             [ccc] => ccc
#         )
# )
```

Read JSON files

```php
<?php

file_put_contents('/tmp/logs.json', json_encode(['log' => 'log text']) . PHP_EOL, FILE_APPEND);
file_put_contents('/tmp/logs.json', json_encode(['log' => 'log text 2']) . PHP_EOL, FILE_APPEND);

$db = new PDO('duckdb::memory:');
$statement = $db->query("SELECT * FROM '/tmp/logs.json'");
print_r($statement->fetchAll(PDO::FETCH_ASSOC));

# Array
# (
#     [0] => Array
#         (
#             [log] => log text
#         )
#     [1] => Array
#         (
#             [log] => log text 2
#         )
# )

$db->exec("COPY (SELECT * FROM '/tmp/logs.json') TO '/tmp/logs_json.parquet' (COMPRESSION zstd)");
```

Use structured columns with a fixed schema

```php
<?php

# s is array{v: string, i: int, a: string[], d: float}

$db = new PDO('duckdb::memory:');
$db->exec('CREATE TABLE table1 (s STRUCT(v VARCHAR, i INTEGER, a VARCHAR[], d DECIMAL))');

$statement = $db->prepare('INSERT INTO table1 VALUES (?)');
$statement->execute([['v' => 'foo', 'i' => 21, 'a' => ['b', 'c'], 'd' => 42.21]]);

$statement = $db->query('SELECT * FROM table1');
```

```php
print_r($statement->fetch(PDO::FETCH_ASSOC));

# Array
# (
#     [s] => Array
#         (
#             [v] => foo
#             [i] => 21
#             [a] => Array
#                 (
#                     [0] => b
#                     [1] => c
#                 )
#             [d] => 42.21
#         )
# )
```

Auto increment columns

```php
<?php

$db = new PDO('duckdb::memory:');
$db->exec('CREATE SEQUENCE table1_id');
$db->exec("CREATE TABLE table1 (id INTEGER PRIMARY KEY DEFAULT nextval('table1_id'))");
$statement = $db->query('INSERT INTO table1 VALUES (default) RETURNING *');
print_r($statement->fetch(PDO::FETCH_ASSOC));

# Array
# (
#     [id] => 1
# )
```

More examples can be found in the readme.

Swift Client

DuckDB has a Swift client. See the announcement post for details.

Instantiating DuckDB

DuckDB supports both in-memory and persistent databases. To work with an in-memory database, run:

```swift
let database = try Database(store: .inMemory)
```

To work with a persistent database, run:

```swift
let database = try Database(store: .file(at: "test.db"))
```

Queries can be issued through a database connection.

```swift
let connection = try database.connect()
```

DuckDB supports multiple connections per database.

Application Example

The rest of the page is based on the example of our announcement post, which uses raw data from NASA's Exoplanet Archive loaded directly into DuckDB.

Creating an Application-Specific Type

We first create an application-specific type that we'll use to house our database and connection and through which we'll eventually define our app-specific queries.

```
import DuckDB

final class ExoplanetStore {

  let database: Database
  let connection: Connection

  init(database: Database, connection: Connection) {
    self.database = database
    self.connection = connection
  }
}
```

Loading a CSV File

We load the data from NASA's Exoplanet Archive:

```
wget https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv -O downloaded_exoplanets.csv
```

Once we have our CSV downloaded locally, we can use the following SQL command to load it as a new table to DuckDB:

```
CREATE TABLE exoplanets AS
    SELECT * FROM read_csv('downloaded_exoplanets.csv');
```

Let's package this up as a new asynchronous factory method on our `ExoplanetStore` type:

```
import DuckDB
import Foundation

final class ExoplanetStore {

  // Factory method to create and prepare a new ExoplanetStore
  static func create() async throws -> ExoplanetStore {

  // Create our database and connection as described above
    let database = try Database(store: .inMemory)
    let connection = try database.connect()

  // Download the CSV from the exoplanet archive
  let (csvFileURL, _) = try await URLSession.shared.download(
    from: URL(string: "https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv")!)

  // Issue our first query to DuckDB
  try connection.execute("""
      CREATE TABLE exoplanets AS
          SELECT * FROM read_csv('\(csvFileURL.path)');
  """)

  // Create our pre-populated ExoplanetStore instance
    return ExoplanetStore(
    database: database,
      connection: connection
  )
  }

  // Let's make the initializer we defined previously
  // private. This prevents anyone accidentally instantiating
```

```
  // the store without having pre-loaded our Exoplanet CSV
  // into the database
  private init(database: Database, connection: Connection) {
  ...
  }
}
```

Querying the Database

The following example queries DuckDB from within Swift via an async function. This means the callee won't be blocked while the query is executing. We'll then cast the result columns to Swift native types using DuckDB's `ResultSet` `cast(to:)` family of methods, before finally wrapping them up in a `DataFrame` from the TabularData framework.

```
...

import TabularData

extension ExoplanetStore {

  // Retrieves the number of exoplanets discovered by year
  func groupedByDiscoveryYear() async throws -> DataFrame {

  // Issue the query we described above
    let result = try connection.query("""
      SELECT disc_year, count(disc_year) AS Count
        FROM exoplanets
        GROUP BY disc_year
        ORDER BY disc_year
      """)

    // Cast our DuckDB columns to their native Swift
    // equivalent types
    let discoveryYearColumn = result[0].cast(to: Int.self)
    let countColumn = result[1].cast(to: Int.self)

    // Use our DuckDB columns to instantiate TabularData
    // columns and populate a TabularData DataFrame
    return DataFrame(columns: [
      TabularData.Column(discoveryYearColumn).eraseToAnyColumn(),
      TabularData.Column(countColumn).eraseToAnyColumn(),
    ])
  }
}
```

Complete Project

For the complete example project, clone the DuckDB Swift repository and open up the runnable app project located in `Examples/SwiftUI/ExoplanetExplorer.xcodeproj`.

SQL

SQL Introduction

This page provides an overview of how to perform simple operations in SQL. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. This tutorial is adapted from the PostgreSQL tutorial.

> DuckDB's SQL dialect closely follows the conventions of the PostgreSQL dialect. The few exceptions to this are listed on the PostgreSQL compatibility page.

In the examples that follow, we assume that you have installed the DuckDB Command Line Interface (CLI) shell. See the installation page for information on how to install the CLI.

> **Tip.** If you are looking for a comprehensive to SQL introduction, check the slide decks of the "Tabular Database Systems" course:
>
> - The Structured Query Language (SQL)
> - More SQL (Subqueries + Embedded SQL)
> - SQL: Grouping + Aggregation and Functional Dependencies

Concepts

DuckDB is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. A relation is essentially a mathematical term for a table.

Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Tables themselves are stored inside schemas, and a collection of schemas constitutes the entire database that you can access.

Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city    VARCHAR,
    temp_lo INTEGER, -- minimum temperature on a day
    temp_hi INTEGER, -- maximum temperature on a day
    prcp    FLOAT,
    date    DATE
);
```

You can enter this into the shell with the line breaks. The command is not terminated until the semicolon.

White space (i.e., spaces, tabs and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dash characters (`--`) introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case-insensitive about keywords and identifiers. When returning identifiers, their original cases are preserved.

In the SQL command, we first specify the type of command that we want to perform: `CREATE TABLE`. After that follows the parameters for the command. First, the table name, `weather`, is given. Then the column names and column types follow.

`city VARCHAR` specifies that the table has a column called `city` that is of type VARCHAR. VARCHAR specifies a data type that can store text of arbitrary length. The temperature fields are stored in an `INTEGER` type, a type that stores integer numbers (i.e., whole numbers without a decimal point). `FLOAT` columns store single precision floating-point numbers (i.e., numbers with a decimal point). `DATE` stores a date (i.e., year, month, day combination). `DATE` only stores the specific day, not a time associated with that day.

DuckDB supports the standard SQL types `INTEGER`, `SMALLINT`, `FLOAT`, `DOUBLE`, `DECIMAL`, `CHAR(n)`, `VARCHAR(n)`, `DATE`, `TIME` and `TIMESTAMP`.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name VARCHAR,
    lat  DECIMAL,
    lon  DECIMAL
);
```

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE <tablename>;
```

Populating a Table with Rows

The insert statement is used to populate a table with rows:

```
INSERT INTO weather
VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Constants that are not numeric values (e.g., text and dates) must be surrounded by single quotes (`''`), as in the example. Input dates for the date type must be formatted as `'YYYY-MM-DD'`.

We can insert into the `cities` table in the same manner.

```
INSERT INTO cities
VALUES ('San Francisco', -194.0, 53.0);
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the `prcp` is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

> **Tip.** Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

Alternatively, you can use the COPY statement. This is faster for large amounts of data because the `COPY` command is optimized for bulk loading while allowing less flexibility than `INSERT`. An example with `weather.csv` would be:

```
COPY weather
FROM 'weather.csv';
```

Where the file name for the source file must be available on the machine running the process. There are many other ways of loading data into DuckDB, see the corresponding documentation section for more information.

Querying a Table

To retrieve data from a table, the table is queried. A SQL `SELECT` statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT *
FROM weather;
```

Here `*` is a shorthand for “all columns”. So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date
FROM weather;
```

The output should be:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29
Hayward	37	54	NULL	1994-11-29

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi + temp_lo) / 2 AS temp_avg, date
FROM weather;
```

This should give:

city	temp_avg	date
San Francisco	48.0	1994-11-27
San Francisco	50.0	1994-11-29
Hayward	45.5	1994-11-29

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be "qualified" by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT *
FROM weather
WHERE city = 'San Francisco'
  AND prcp > 0.0;
```

Result:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27

You can request that the results of a query be returned in sorted order:

```
SELECT *
FROM weather
ORDER BY city;
```

city	temp_lo	temp_hi	prcp	date
Hayward	37	54	NULL	1994-11-29
San Francisco	43	57	0.0	1994-11-29
San Francisco	46	50	0.25	1994-11-27

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do:

```
SELECT *
FROM weather
ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
FROM weather;
```

city
San Francisco
Hayward

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together:

```
SELECT DISTINCT city
FROM weather
ORDER BY city;
```

Joins between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the `weather` table with the name column of all rows in the `cities` table, and select the pairs of rows where these values match.

This would be accomplished by the following query:

```
SELECT *
FROM weather, cities
WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	name	lat	lon
San Francisco	46	50	0.25	1994-11-27	San Francisco	-194.000	53.000
San Francisco	43	57	0.0	1994-11-29	San Francisco	-194.000	53.000

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the `weather` table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns from the `weather` and `cities` tables are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, lon, lat
FROM weather, cities
WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	lon	lat
San Francisco	46	50	0.25	1994-11-27	53.000	-194.000

city	temp_lo	temp_hi	prcp	date	lon	lat
San Francisco	43	57	0.0	1994-11-29	53.000	-194.000

Since the columns all had different names, the parser automatically found which table they belong to. If there were duplicate column names in the two tables you'd need to qualify the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.lon, cities.lat
FROM weather, cities
WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
FROM weather
INNER JOIN cities ON weather.city = cities.name;
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching cities row(s). If no matching row is found we want some “empty values” to be substituted for the `cities` table's columns. This kind of query is called an outer join. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT *
FROM weather
LEFT OUTER JOIN cities ON weather.city = cities.name;
```

city	temp_lo	temp_hi	prcp	date	name	lat	lon
San Francisco	46	50	0.25	1994-11-27	San Francisco	-194.000	53.000
San Francisco	43	57	0.0	1994-11-29	San Francisco	-194.000	53.000
Hayward	37	54	NULL	1994-11-29	NULL	NULL	NULL

This query is called a left outer join because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

Aggregate Functions

Like most other relational database products, DuckDB supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with:

```
SELECT max(temp_lo)
FROM weather;
```

max(temp_lo)
46

If we wanted to know what city (or cities) that reading occurred in, we might try:

```
SELECT city
FROM weather
WHERE temp_lo = max(temp_lo);
```

But this will not work since the aggregate max cannot be used in the WHERE clause:

```
Binder Error:
WHERE clause cannot contain aggregates!
```

This restriction exists because the WHERE clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed. However, as is often the case the query can be restated to accomplish the desired result, here by using a subquery:

```
SELECT city
FROM weather
WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

city
San Francisco

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with GROUP BY clauses. For example, we can get the maximum low temperature observed in each city with:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city;
```

city	max(temp_lo)
San Francisco	46
Hayward	37

Which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using HAVING:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city
HAVING max(temp_lo) < 40;
```

city	max(temp_lo)
Hayward	37

which gives us the same results for only the cities that have all temp_lo values below 40. Finally, if we only care about cities whose names begin with S, we can use the LIKE operator:

```
SELECT city, max(temp_lo)
FROM weather
WHERE city LIKE 'S%'            -- (1)
GROUP BY city
HAVING max(temp_lo) < 40;
```

More information about the LIKE operator can be found in the pattern matching page.

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions.

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

```
UPDATE weather
SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT *
FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0.0	1994-11-29
Hayward	35	52	NULL	1994-11-29

Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather
WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT *
FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0.0	1994-11-29

One should be cautious when issuing statements of the following form:

```
DELETE FROM <table_name>;
```

> **Warning.** Without a qualification, `DELETE` will remove all rows from the given table, leaving it empty. The system will not request confirmation before doing this.

Statements

Statements Overview

ANALYZE Statement

The ANALYZE statement recomputes the statistics on DuckDB's tables.

Usage

The statistics recomputed by the ANALYZE statement are only used for join order optimization. It is therefore recommended to recompute these statistics for improved join orders, especially after performing large updates (inserts and/or deletes).

To recompute the statistics, run:

```
ANALYZE;
```

ALTER DATABASE Statement

> **Warning.** The `ALTER DATABASE` statement is not supported in DuckDB 2.0.0: `ALTER DATABASE <old_name> RENAME TO <new_name>` fails with `Parser Error: syntax error at or near "RENAME"`. To rename an attached database, DETACH it and ATTACH it again with a different alias.

Syntax

RENAME DATABASE

The following scenarios are not supported when renaming a DuckDB database with `ALTER DATABASE`:

- Renaming a database to certain reserved keywords such as `system` or `temp`.
- Renaming a database to the same name of a database attached in memory.

Rename a database from `old_name` to `new_name`:

```
ALTER DATABASE old_name RENAME TO new_name;
```

Check if a database exists and rename it:

```
ALTER DATABASE IF EXISTS non_existent RENAME TO something_else;
```

ALTER TABLE Statement

The `ALTER TABLE` statement changes the schema of an existing table in the catalog.

Examples

```
CREATE TABLE integers (i INTEGER, j INTEGER);
```

Add a new column with name k to the table `integers`, it will be filled with the default value `NULL`:

```
ALTER TABLE integers
ADD COLUMN k INTEGER;
```

Add a new column with name `l` to the table integers, it will be filled with the default value 10:

```
ALTER TABLE integers
ADD COLUMN l INTEGER DEFAULT 10;
```

Drop the column k from the table integers:

```
ALTER TABLE integers
DROP k;
```

Change the type of the column `i` to the type VARCHAR using a standard cast:

```
ALTER TABLE integers
ALTER i TYPE VARCHAR;
```

Change the type of the column `i` to the type VARCHAR, using the specified expression to convert the data for each row:

```
ALTER TABLE integers
ALTER i SET DATA TYPE VARCHAR USING concat(i, '_', j);
```

Set the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i SET DEFAULT 10;
```

Drop the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i DROP DEFAULT;
```

Make a column not nullable:

```
ALTER TABLE integers
ALTER COLUMN i SET NOT NULL;
```

Drop the not-`NULL` constraint:

```
ALTER TABLE integers
ALTER COLUMN i DROP NOT NULL;
```

Rename a table:

```
ALTER TABLE integers
RENAME TO integers_old;
```

Rename a column of a table:

```
ALTER TABLE integers
RENAME i TO ii;
```

Add a primary key to a column of a table:

```
ALTER TABLE integers
ADD PRIMARY KEY (i);
```

Add a unique constraint to a column of a table:

```
ALTER TABLE integers
ADD UNIQUE (j);
```

Syntax

`ALTER TABLE` changes the schema of an existing table. All the changes made by `ALTER TABLE` fully respect the transactional semantics, i.e., they will not be visible to other transactions until committed, and can be fully reverted through a rollback.

`RENAME TABLE`

Rename a table:

```
ALTER TABLE integers
RENAME TO integers_old;
```

The `RENAME TO` clause renames an entire table, changing its name in the schema. Note that any views that rely on the table are **not** automatically updated.

> New As of DuckDB 2.0, `ALTER TABLE <view> RENAME TO <new_name>` can also rename a view, for compatibility with PostgreSQL. The equivalent `ALTER VIEW <view> RENAME TO <new_name>` continues to work as well.

`RENAME COLUMN`

To rename a column of a table, use the `RENAME` or `RENAME COLUMN` clauses:

```
ALTER TABLE integers
RENAME COLUMN i TO j;
```

```
ALTER TABLE integers
RENAME i TO j;
```

The `RENAME [COLUMN]` clause renames a single column within a table. Any constraints that rely on this name (e.g., `CHECK` constraints) are automatically updated. However, note that any views that rely on this column name are **not** automatically updated.

`ADD COLUMN`

To add a column of a table, use the `ADD` or `ADD COLUMN` clauses.

E.g., to add a new column with name `k` to the table `integers`, it will be filled with the default value `NULL`:

```
ALTER TABLE integers
ADD COLUMN k INTEGER;
```

Or:

```
ALTER TABLE integers
ADD k INTEGER;
```

Add a new column with name `l` to the table integers, it will be filled with the default value 10:

```
ALTER TABLE integers
ADD COLUMN l INTEGER DEFAULT 10;
```

The `ADD [COLUMN]` clause can be used to add a new column of a specified type to a table. The new column will be filled with the specified default value, or `NULL` if none is specified.

DROP COLUMN

To drop a column of a table, use the DROP or DROP COLUMN clause:

E.g., to drop the column k from the table `integers`:

```
ALTER TABLE integers
DROP COLUMN k;
```

Or:

```
ALTER TABLE integers
DROP k;
```

The DROP [COLUMN] clause can be used to remove a column from a table. Note that columns can only be removed if they do not have any indexes that rely on them. This includes any indexes created as part of a PRIMARY KEY or UNIQUE constraint. Columns that are part of multi-column check constraints cannot be dropped either. If you attempt to drop a column with an index on it, DuckDB will return the following error message:

```
Catalog Error:
Cannot drop this column: an index depends on it!
```

[SET [DATA]] TYPE

Change the type of the column `i` to the type VARCHAR using a standard cast:

```
ALTER TABLE integers
ALTER i TYPE VARCHAR;
```

> Instead of ALTER <column_name> TYPE <type>{:.language-sql .highlight}, you can also use the equivalent ALTER <column_name> SET TYPE <type>{:.language-sql .highlight} and the ALTER <column_name> SET DATA TYPE <type>{:.language-sql .highlight} clauses.

Change the type of the column `i` to the type VARCHAR, using the specified expression to convert the data for each row:

```
ALTER TABLE integers
ALTER i SET DATA TYPE VARCHAR USING concat(i, '_', j);
```

The [SET [DATA]] TYPE clause changes the type of a column in a table. Any data present in the column is converted according to the provided expression in the USING clause, or, if the USING clause is absent, cast to the new data type. Note that columns can only have their type changed if they do not have any indexes that rely on them and are not part of any CHECK constraints.

Handling Structs

There are two options to change the sub-schema of a STRUCT-typed column.

ALTER TABLE with `struct_insert` You can use ALTER TABLE with the `struct_insert` function. For example:

```
CREATE TABLE tbl (col STRUCT(i INTEGER));
ALTER TABLE tbl
ALTER col TYPE USING struct_insert(col, a := 42, b := NULL::VARCHAR);
```

ALTER TABLE with ADD COLUMN / DROP COLUMN / RENAME COLUMN Starting with DuckDB 1.3.0, ALTER TABLE supports the ADD COLUMN, DROP COLUMN and RENAME COLUMN clauses to update the sub-schema of a STRUCT.

SET / DROP DEFAULT

Set the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i SET DEFAULT 10;
```

Drop the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i DROP DEFAULT;
```

The `SET/DROP DEFAULT` clause modifies the `DEFAULT` value of an existing column. Note that this does not modify any existing data in the column. Dropping the default is equivalent to setting the default value to NULL.

> **Warning.** At the moment, DuckDB does not allow most `ALTER TABLE` operations (e.g., renaming the table, renaming, dropping or changing the type of columns, or setting or dropping `NOT NULL`) on a table that has an index. Adding columns and setting or dropping column defaults are allowed. For other changes, drop the index, alter the table, and then recreate the index. Otherwise, you will get a `Dependency Error`.

ADD PRIMARY KEY / ADD UNIQUE

The `ADD [CONSTRAINT <name>] PRIMARY KEY` and `ADD [CONSTRAINT <name>] UNIQUE` clauses add a primary key or unique constraint to an existing table.

Add a primary key to a column of a table:

```
ALTER TABLE integers
ADD PRIMARY KEY (i);
```

Add a primary key to multiple columns of a table:

```
ALTER TABLE integers
ADD PRIMARY KEY (i, j);
```

Add a unique constraint to a column of a table:

```
ALTER TABLE integers
ADD UNIQUE (j);
```

The constraint can also be introduced with the `CONSTRAINT <name>` keyword:

```
ALTER TABLE integers
ADD CONSTRAINT j_unique UNIQUE (j);
```

```
ALTER TABLE integers
ADD CONSTRAINT i_pk PRIMARY KEY (i);
```

If the existing data violates the constraint, the statement fails:

```
Constraint Error:
Data contains duplicates on indexed column(s)
```

SET / RESET (Table Options)

> This feature was introduced in DuckDB 1.5.

Modify table options after table creation.

Set table options:

```
ALTER TABLE my_table
SET ('option_name' = 'value');
```

Reset table options to their defaults:

```
ALTER TABLE my_table
RESET ('option_name');
```

The SET clause assigns values to table options as key-value pairs. The RESET clause removes options or restores them to their default values, depending on the catalog implementation.

Multiple options can be set or reset in a single statement:

```
ALTER TABLE my_table
SET ('option1' = 'value1', 'option2' = 'value2');

ALTER TABLE my_table
RESET ('option1', 'option2');
```

ADD / DROP CONSTRAINT

ADD [CONSTRAINT <name>] is supported for PRIMARY KEY and UNIQUE constraints, see above.

> Adding CHECK and FOREIGN KEY constraints with ALTER TABLE and the DROP CONSTRAINT clause are not yet supported in DuckDB.

Limitations

ALTER COLUMN fails if values of conflicting types have occurred in the table at any point, even if they have been deleted:

```
CREATE TABLE tbl (col VARCHAR);

INSERT INTO tbl
VALUES ('asdf'), ('42');

DELETE FROM tbl
WHERE col = 'asdf';

ALTER TABLE tbl
ALTER COLUMN col TYPE INTEGER;
```

```
Conversion Error:
Could not convert string 'asdf' to INT32
```

Currently, this is expected behavior. As a workaround, you can create a copy of the table:

```
CREATE OR REPLACE TABLE tbl AS FROM tbl;
```

ALTER VIEW Statement

The ALTER VIEW statement changes the schema of an existing view in the catalog.

Examples

Rename a view:

```
ALTER VIEW view1 RENAME TO view2;
```

ALTER VIEW changes the schema of an existing view. All the changes made by ALTER VIEW fully respect the transactional semantics, i.e., they will not be visible to other transactions until committed, and can be fully reverted through a rollback. Note that other views that rely on the view are **not** automatically updated.

ATTACH and DETACH Statements

DuckDB allows attaching to and detaching from database files.

Examples

Attach the database `file.db` with the alias inferred from the name (`file`):

```
ATTACH 'file.db';
```

Attach the database `file.db` with an explicit alias (`file_db`):

```
ATTACH 'file.db' AS file_db;
```

Attach the database `file.db` in read only mode:

```
ATTACH 'file.db' (READ_ONLY);
```

Attach the database `file.db` with a block size of 16 kB:

```
ATTACH 'file.db' (BLOCK_SIZE 16_384);
```

Attach the database `file.db` with a row group size of 2048 rows:

```
ATTACH 'file.db' (ROW_GROUP_SIZE 2048);
```

Attach the database `file.db` with WAL writes disabled for improved performance:

```
ATTACH 'file.db' (RECOVERY_MODE no_wal_writes);
```

Attach a SQLite database for reading and writing (see the `sqlite` extension for more information):

```
ATTACH 'sqlite_file.db' AS sqlite_db (TYPE sqlite);
```

Attach the database `file.db` if inferred database alias `file` does not yet exist:

```
ATTACH IF NOT EXISTS 'file.db';
```

Attach the database `file.db` if explicit database alias `file_db` does not yet exist:

```
ATTACH IF NOT EXISTS 'file.db' AS file_db;
```

Attach the database `file2.db` as alias `file_db` detaching and replacing the existing alias if it exists:

```
ATTACH OR REPLACE 'file2.db' AS file_db;
```

Create a table in the attached database with alias `file`:

```
CREATE TABLE file.new_table (i INTEGER);
```

Detach the database with alias `file`:

```
DETACH file;
```

Show a list of all attached databases:

```
SHOW DATABASES;
```

Change the default database that is used to the database `file`:

```
USE file;
```

ATTACH

The ATTACH statement adds a new database file to the catalog that can be read from and written to. Note that attachment definitions are not persisted between sessions: when a new session is launched, you have to re-attach to all databases.

ATTACH Syntax

ATTACH allows DuckDB to operate on multiple database files, and allows for transfer of data between different database files.

ATTACH supports HTTP and S3 endpoints. For these, it creates a read-only connection by default. Therefore, the following two commands are equivalent:

```
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db;
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db (READ_ONLY);
```

Similarly, the following two commands connecting to S3 are equivalent:

```
ATTACH 's3://<blobs-duckdb>/databases/stations.duckdb' AS stations_db;
ATTACH 's3://<blobs-duckdb>/databases/stations.duckdb' AS stations_db (READ_ONLY);
```

Explicit Storage Versions

DuckDB 1.2.0 introduced the `STORAGE_VERSION` option, which allows explicitly specifying the storage version. This setting specifies the minimum DuckDB version that should be able to read the database file. When database files are written with this option, the resulting files cannot be opened by older DuckDB versions than the specified version. They can be read by the specified version and all newer versions of DuckDB.

Since DuckDB 2.0.0, new database files use the latest storage version by default (equivalent to `STORAGE_VERSION 'latest'`), so they cannot be read by older DuckDB versions. Therefore, the main use of this option is to create database files that remain readable by older DuckDB versions:

```
ATTACH 'file.db' (STORAGE_VERSION 'v1.5.0');
```

You can check the storage version of attached databases using the `tags` column of the `duckdb_databases()` function:

```
SELECT database_name, tags
FROM duckdb_databases();
```

For more information on upgrading to DuckDB 2.0.0, see the migration guide. For more details on storage versions, see the "Storage" page.

Database Encryption

DuckDB supports database encryption. By default, it uses AES encryption with a key length of 256 bits using the recommended GCM mode. The encryption covers the main database file, the write-ahead-log (WAL) file, and even temporary files. To attach to an encrypted database, use the `ATTACH` statement with an `ENCRYPTION_KEY`.

```
ATTACH 'encrypted.db' AS enc_db (ENCRYPTION_KEY 'quack_quack');
```

To encrypt data, DuckDB can use either the built-in `mbedtls` library or the OpenSSL library from the `httpfs` extension. Note that the OpenSSL versions are much faster due to hardware acceleration, so make sure to load the `httpfs` for good encryption performance:

```
LOAD httpfs;
ATTACH 'encrypted.db' AS enc_db (ENCRYPTION_KEY 'quack_quack'); -- will be faster thanks to httpfs
```

To change the AES mode to CBC or CTR, use the `ENCRYPTION_CIPHER` option:

```
ATTACH 'encrypted.db' AS enc_db (ENCRYPTION_KEY 'quack_quack', ENCRYPTION_CIPHER 'CBC');
ATTACH 'encrypted.db' AS enc_db (ENCRYPTION_KEY 'quack_quack', ENCRYPTION_CIPHER 'CTR');
```

Database encryption implies using storage version 1.4.0 or later.

Expressions in Paths and Options

> New Using expressions for the database path was introduced in DuckDB 2.0. In earlier versions, the path had to be a string literal.

The database path and the option values can be arbitrary scalar expressions rather than string literals. DuckDB evaluates these expressions while binding the ATTACH statement, so you can build them from variables, environment variables or string concatenation. This keeps credentials and environment-specific identifiers, such as account numbers, out of your SQL scripts.

Read the database path from a variable:

```
SET VARIABLE db_path = 'file.db';
ATTACH getvariable('db_path') AS file_db;
```

Build the path by concatenating a variable, for example an account number read from the environment:

```
SET VARIABLE account_id = '123456789012';
ATTACH 'account_' || getvariable('account_id') || '.db' AS account_db;
```

Read the path from an environment variable using `getenv()`, which is available in the CLI client only:

```
ATTACH getenv('DATABASE_PATH') AS env_db;
```

The path expression must not evaluate to `NULL`.

Options

Zero or more copy options may be provided within parentheses following the ATTACH statement. Parameter values can be passed in with or without wrapping in single quotes. Arbitrary expressions may be used for parameter values.

Name	Description	Type	Default value
READ_ONLY	Attach the database in read-only mode. Use READ_WRITE (the default) to attach in read-write mode.	BOOLEAN	false
COMPRESS	Whether the database is compressed. Only applicable for in-memory databases.	VARCHAR	false
TYPE	The type of the attached database, e.g., DUCKDB, SQLITE, POSTGRES, MYSQL. If omitted, the type is deduced from the path. See Database Type.	VARCHAR	DUCKDB
DEFAULT_TABLE	The table that is queried when the attached database is referenced directly by its catalog name (e.g., FROM <db>).	VARCHAR	-
BLOCK_SIZE	The block size of a new database file. Must be a power of two and within [16384, 262144]. Cannot be set for existing files.	UBIGINT	262144
ROW_GROUP_SIZE	The row group size of a new database file.	UBIGINT	122880
STORAGE_VERSION	The version of the storage used.	VARCHAR	latest
ENCRYPTION_KEY	The encryption key used for encrypting the database.	VARCHAR	-
ENCRYPTION_CIPHER	The encryption cipher used for encrypting the database (CBC, CTR or GCM).	VARCHAR	-
RECOVERY_MODE	Recovery mode for the database. no_wal_writes disables WAL writes, improving performance at the cost of crash recovery.	VARCHAR	-

> The options above apply when attaching DuckDB and SQLite database files. Attaching a database managed by another system ac-

> cepts additional type-specific options, such as SECRET and SCHEMA. See the relevant extension page for the full list: PostgreSQL, MySQL, and Iceberg.

Database Type

The TYPE option specifies which storage engine backs the attached database. The type name is case-insensitive. Native DuckDB storage uses DUCKDB (the default). Other types are provided by extensions, for example SQLITE (via the `sqlite` extension), POSTGRES (via the `postgres` extension), and MYSQL (via the `mysql` extension). If the extension that provides the type is not already loaded, it is autoloaded.

When TYPE is not specified, DuckDB deduces the type from the path in two steps:

1. If the path begins with a recognized scheme prefix followed by a colon, such as `sqlite:file.db`, `postgres:dbname=postgres`, or `mysql:db=mysql`, the scheme determines the type. The prefix must be at least two characters long and must not be followed by `//`, so URLs such as `https://...` are not treated as type prefixes.
2. Otherwise, DuckDB inspects the file contents. A SQLite header selects the SQLITE type, and a DuckDB database file uses the native DUCKDB type.

> **Warning.** Passing TYPE DUCKDB explicitly fixes the type to native DuckDB storage and disables the path-based deduction described above. If you attach a path that carries a scheme prefix for another engine but also pass TYPE DUCKDB, the prefix is ignored and the statement does not have the intended effect.

Attaching Data Files Directly

A Parquet, CSV, or JSON file can be attached directly, not only a DuckDB database file. This creates a database that exposes the file through two views: one named `file` and one named after the database alias.

```
ATTACH 'people.parquet' AS people;
SELECT * FROM people.file;
SELECT * FROM people.people;
```

Both views are defined as `SELECT * FROM '<path>'`, so the data is read from the file each time a view is queried. Because the views wrap the file rather than a stored table, they are read-only. To change the data, write a new file.

DETACH

The DETACH statement allows previously attached database files to be closed and detached, releasing any locks held on the database file.

Note that it is not possible to detach from the default database: if you would like to do so, issue the USE statement to change the default database to another one. For example, if you are connected to a persistent database, you may change to an in-memory database by issuing:

```
ATTACH ':memory:' AS memory_db;
USE memory_db;
```

> **Warning.** Closing the connection, e.g., invoking the `close()` function in Python, does not release the locks held on the database files as the file handles are held by the main DuckDB instance (in Python's case, the `duckdb` module).

DETACH Syntax

Name Qualification

The fully qualified name of catalog objects contains the *catalog*, the *schema* and the *name* of the object. For example:

Attach the database new_db:

```
ATTACH 'new_db.db';
```

Create the schema `my_schema` in the database `new_db`:

```
CREATE SCHEMA new_db.my_schema;
```

Create the table `my_table` in the schema `my_schema`:

```
CREATE TABLE new_db.my_schema.my_table (col INTEGER);
```

Refer to the column `col` inside the table `my_table`:

```
SELECT new_db.my_schema.my_table.col FROM new_db.my_schema.my_table;
```

Note that often the fully qualified name is not required. When a name is not fully qualified, the system looks for which entries to reference using the *catalog search path*. The default catalog search path includes the system catalog, the temporary catalog and the initially attached database together with the `main` schema.

Also note the rules on identifiers and database names in particular.

Default Database and Schema

When a table is created without any qualifications, the table is created in the default schema of the default database. The default database is the database that is launched when the system is created – and the default schema is `main`.

Create the table `my_table` in the default database:

```
CREATE TABLE my_table (col INTEGER);
```

Changing the Default Database and Schema

The default database and schema can be changed using the `USE` command.

Set the default database schema to `new_db.main`:

```
USE new_db;
```

Set the default database schema to `new_db.my_schema`:

```
USE new_db.my_schema;
```

Resolving Conflicts

When providing only a single qualification, the system can interpret this as *either* a catalog *or* a schema, as long as there are no conflicts. For example:

```
ATTACH 'new_db.db';
CREATE SCHEMA my_schema;
```

Creates the table `new_db.main.tbl`:

```
CREATE TABLE new_db.tbl (i INTEGER);
```

Creates the table `default_db.my_schema.tbl`:

```
CREATE TABLE my_schema.tbl (i INTEGER);
```

If we create a conflict (i.e., we have both a schema and a catalog with the same name) the system requests that a fully qualified path is used instead:

```
CREATE SCHEMA new_db;
CREATE TABLE new_db.tbl (i INTEGER);
```

```
Binder Error:
Ambiguous reference to catalog or schema "new_db" - use a fully qualified path like "memory.new_db"
```

Changing the Catalog Search Path

The catalog search path can be adjusted by setting the `search_path` configuration option, which uses a comma-separated list of values that will be on the search path. The following example demonstrates searching in two databases:

```
ATTACH ':memory:' AS db1;
ATTACH ':memory:' AS db2;
CREATE table db1.tbl1 (i INTEGER);
CREATE table db2.tbl2 (j INTEGER);
```

Reference the tables using their fully qualified name:

```
SELECT * FROM db1.tbl1;
SELECT * FROM db2.tbl2;
```

Or set the search path and reference the tables using their name:

```
SET search_path = 'db1,db2';
SELECT * FROM tbl1;
SELECT * FROM tbl2;
```

Transactional Semantics

When running queries on multiple databases, the system opens separate transactions per database. The transactions are started *lazily* by default – when a given database is referenced for the first time in a query, a transaction for that database will be started. `SET immediate_transaction_mode = true{:.language-sql .highlight}` can be toggled to change this behavior to eagerly start transactions in all attached databases instead.

While multiple transactions can be active at a time – the system only supports *writing* to a single attached database in a single transaction. If you try to write to multiple attached databases in a single transaction the following error will be thrown:

```
Attempting to write to database "db2" in a transaction that has already modified database "db1" -
a single transaction can only write to a single attached database.
```

The reason for this restriction is that the system does not maintain atomicity for transactions across attached databases. Transactions are only atomic *within* each database file. By restricting the global transaction to write to only a single database file the atomicity guarantees are maintained.

CALL Statement

The `CALL` statement invokes the given table function and returns the results.

> Thanks to the FROM-first syntax and the fact that procedures in DuckDB are implemented as table functions, you can use `FROM` instead of `CALL`.

Examples

Invoke the `duckdb_functions` table function:

```
CALL duckdb_functions();
```

Invoke the `pragma_table_info` table function:

```
CALL pragma_table_info('pg_am');
```

Select only the functions where the name starts with `ST_`:

```
SELECT function_name, parameters, parameter_types, return_type
FROM duckdb_functions()
WHERE function_name LIKE 'ST_%';
```

Syntax

CHECKPOINT Statement

The CHECKPOINT statement synchronizes data in the write-ahead log (WAL) to the database data file.

Examples

Synchronize data in the default database:

```
CHECKPOINT;
```

Synchronize data in the specified database:

```
CHECKPOINT file_db;
```

Synchronize data and prevent new transactions from starting:

```
FORCE CHECKPOINT;
```

> In earlier DuckDB versions, FORCE CHECKPOINT aborted any in-progress transactions. From v1.4, it waits until it can grab the checkpoint lock.

Checkpointing In-Memory Tables

Starting with v1.4.0, in-memory tables support checkpointing. This has two key benefits:

- In-memory tables also support compression. This is disabled by default – you can turn it on using:

  ```
  ATTACH ':memory:' AS memory_compressed (COMPRESS);
  USE memory_compressed;
  ```

- Checkpointing triggers vacuuming deleted rows, allowing space to be reclaimed after deletes/truncation.

Syntax

Checkpoint operations happen automatically based on the WAL size (see Configuration). This statement is for manual checkpoint actions.

Behavior

The default CHECKPOINT command will fail if there are any running transactions. Including FORCE will wait until it can grab the checkpoint lock and then execute the checkpoint operation. In DuckDB versions before v1.4, FORCE CHECKPOINT instead aborted any in-progress transactions.

Also see the related PRAGMA option for further behavior modification.

Reclaiming Space

When performing a checkpoint (automatic or otherwise), the space occupied by deleted rows is partially reclaimed. Note that this does not remove all deleted rows, but rather merges row groups that have a significant amount of deletes together. In the current implementation this requires ~25% of rows to be deleted in adjacent row groups.

Since v1.4.0, checkpointing also applies to in-memory databases (see Checkpointing In-Memory Tables), where it likewise vacuums deleted rows and reclaims space.

> **Warning.** The VACUUM statement does *not* trigger vacuuming deletes and hence does not reclaim space.

COMMENT ON Statement

The `COMMENT ON` statement allows adding metadata to catalog entries (tables, columns, etc.). It follows the PostgreSQL syntax.

Examples

Create a comment on a `TABLE`:

```
COMMENT ON TABLE test_table IS 'very nice table';
```

Create a comment on a `COLUMN`:

```
COMMENT ON COLUMN test_table.test_table_column IS 'very nice column';
```

Create a comment on a `VIEW`:

```
COMMENT ON VIEW test_view IS 'very nice view';
```

Create a comment on an `INDEX`:

```
COMMENT ON INDEX test_index IS 'very nice index';
```

Create a comment on a `SEQUENCE`:

```
COMMENT ON SEQUENCE test_sequence IS 'very nice sequence';
```

Create a comment on a `TYPE`:

```
COMMENT ON TYPE test_type IS 'very nice type';
```

Create a comment on a `MACRO`:

```
COMMENT ON MACRO test_macro IS 'very nice macro';
```

Create a comment on a `MACRO TABLE`:

```
COMMENT ON MACRO TABLE test_table_macro IS 'very nice table macro';
```

To unset a comment, set it to `NULL`, e.g.:

```
COMMENT ON TABLE test_table IS NULL;
```

Reading Comments

Comments can be read by querying the `comment` column of the respective metadata functions:

List comments on `TABLE`s:

```
SELECT comment FROM duckdb_tables();
```

List comments on `COLUMN`s:

```
SELECT comment FROM duckdb_columns();
```

List comments on `VIEW`s:

```
SELECT comment FROM duckdb_views();
```

List comments on `INDEX`s:

```
SELECT comment FROM duckdb_indexes();
```

List comments on SEQUENCEs:

```
SELECT comment FROM duckdb_sequences();
```

List comments on TYPEs:

```
SELECT comment FROM duckdb_types();
```

List comments on MACROs:

```
SELECT comment FROM duckdb_functions();
```

List comments on MACRO TABLEs:

```
SELECT comment FROM duckdb_functions();
```

Limitations

The COMMENT ON statement currently has the following limitations:

- It is not possible to comment on schemas or databases.

Syntax

CONNECT and DISCONNECT Statements

> New Support for the CONNECT and DISCONNECT statements was introduced in DuckDB 2.0.

The CONNECT statement routes the current session to a remote database, so that subsequent queries execute on that remote and their results stream back. DISCONNECT returns the session to the local database.

CONNECT is central to using DuckDB as a server: a DuckDB instance can serve its databases over the network, and any other DuckDB can attach to it and route queries there.

Examples

Attach a remote DuckDB server over the Quack protocol, route queries to it, and disconnect again:

```
ATTACH 'quack:server.example.com' AS qk (TOKEN '<my_token>');

CONNECT qk;
SELECT count(*) FROM events; -- executes on the server, results stream back
DISCONNECT;
```

Connect directly to a remote database using a connection string. For databases other than DuckDB, the connection string must start with the type of the database, e.g., postgres:, followed by a connection string for that database (here, a libpq connection string):

```
CONNECT 'postgres:dbname=mydb host=localhost';
SELECT count(*) FROM orders; -- runs on the PostgreSQL server
DISCONNECT;
```

Syntax

```
CONNECT { <attached_database> | '<connection_string>' };
DISCONNECT;
```

CONNECT accepts either the alias of a database attached with ATTACH, or a connection string for a remote database. While a connection is active, queries that do not explicitly qualify a different database execute on the remote.

DISCONNECT ends the active connection and returns the session to the local database. Issuing DISCONNECT without an active connection returns an error.

Supported Databases

CONNECT works with databases that support remote execution, including:

- DuckDB servers reached over the Quack protocol.
- PostgreSQL and MySQL servers. For these, the remote pushdown optimizer ships SQL directly to the server instead of pulling tables over the wire.

In-process local databases, such as the default in-memory database or a DuckDB database file, do not support CONNECT.

COPY Statement

Examples

Read a CSV file into the `lineitem` table, using auto-detected CSV options:

```
COPY lineitem FROM 'lineitem.csv';
```

Read a CSV file into the `lineitem` table, using manually specified CSV options:

```
COPY lineitem FROM 'lineitem.csv' (DELIMITER '|');
```

Read a Parquet file into the `lineitem` table:

```
COPY lineitem FROM 'lineitem.pq' (FORMAT parquet);
```

Read a JSON file into the `lineitem` table, using auto-detected options:

```
COPY lineitem FROM 'lineitem.json' (FORMAT json, AUTO_DETECT true);
```

Read a CSV file into the `lineitem` table, using double quotes:

```
COPY lineitem FROM "lineitem.csv";
```

Read a CSV file into the `lineitem` table, omitting quotes:

```
COPY lineitem FROM lineitem.csv;
```

Write a table to a CSV file:

```
COPY lineitem TO 'lineitem.csv' (FORMAT csv, DELIMITER '|', HEADER);
```

Write a table to a CSV file, using double quotes:

```
COPY lineitem TO "lineitem.csv";
```

Write a table to a CSV file, omitting quotes:

```
COPY lineitem TO lineitem.csv;
```

Write the result of a query to a Parquet file:

```
COPY (SELECT l_orderkey, l_partkey FROM lineitem) TO 'lineitem.parquet' (COMPRESSION zstd);
```

Copy the entire content of database `db1` to database `db2`:

```
COPY FROM DATABASE db1 TO db2;
```

Copy only the schema (catalog elements) but not any data:

```
COPY FROM DATABASE db1 TO db2 (SCHEMA);
```

Overview

COPY moves data between DuckDB and external files. COPY ... FROM imports data into DuckDB from an external file. COPY ... TO writes data from DuckDB to an external file. The COPY command can be used for CSV, PARQUET and JSON files.

COPY ... FROM

COPY ... FROM imports data from an external file into an existing table. The data is appended to whatever data is in the table already. The amount of columns inside the file must match the amount of columns in the table `tbl`, and the contents of the columns must be convertible to the column types of the table. In case this is not possible, an error will be thrown.

If a list of columns is specified, COPY will only copy the data in the specified columns from the file. If there are any columns in the table that are not in the column list, COPY ... FROM will insert the default values for those columns.

Copy the contents of a comma-separated file `test.csv` without a header into the table `test`:

```
COPY test FROM 'test.csv';
```

Copy the contents of a comma-separated file with a header into the `category` table:

```
COPY category FROM 'categories.csv' (HEADER);
```

Copy the contents of `lineitem.tbl` into the `lineitem` table, where the contents are delimited by a pipe character (`|`):

```
COPY lineitem FROM 'lineitem.tbl' (DELIMITER '|');
```

Copy the contents of `lineitem.tbl` into the `lineitem` table, where the delimiter, quote character, and presence of a header are automatically detected:

```
COPY lineitem FROM 'lineitem.tbl' (AUTO_DETECT true);
```

Read the contents of a comma-separated file `names.csv` into the `name` column of the `category` table. Any other columns of this table are filled with their default value:

```
COPY category(name) FROM 'names.csv';
```

Read the contents of a Parquet file `lineitem.parquet` into the `lineitem` table:

```
COPY lineitem FROM 'lineitem.parquet' (FORMAT parquet);
```

Read the contents of a newline-delimited JSON file `lineitem.ndjson` into the `lineitem` table:

```
COPY lineitem FROM 'lineitem.ndjson' (FORMAT json);
```

Read the contents of a JSON file `lineitem.json` into the `lineitem` table:

```
COPY lineitem FROM 'lineitem.json' (FORMAT json, ARRAY true);
```

An expression may be used as the source of a COPY ... FROM command if it is placed within parentheses.

Read the contents of a file whose path is stored in a variable into the `lineitem` table:

```
SET VARIABLE source_file = 'lineitem.json';
COPY lineitem FROM (getvariable('source_file'));
```

Read the contents of a file provided as parameter of a prepared statement into the `lineitem` table:

```
PREPARE v1 AS COPY lineitem FROM ($1);
EXECUTE v1('lineitem.json');
```

Syntax

> To ensure compatibility with PostgreSQL, DuckDB accepts `COPY ... FROM` statements that do not fully comply with the railroad diagram shown here. For example, the following is a valid statement:
>
> ```
> COPY tbl FROM 'tbl.csv' WITH DELIMITER '|' CSV HEADER;
> ```

`COPY ... TO`

`COPY ... TO` exports data from DuckDB to an external CSV, Parquet, JSON or BLOB file. It has mostly the same set of options as `COPY ... FROM`, however, in the case of `COPY ... TO` the options specify how the file should be written to disk. Any file created by `COPY ... TO` can be copied back into the database by using `COPY ... FROM` with a similar set of options.

The `COPY ... TO` function can be called specifying either a table name, or a query. When a table name is specified, the contents of the entire table will be written into the resulting file. When a query is specified, the query is executed and the result of the query is written to the resulting file.

> `COPY ... TO` overwrites the destination file if it already exists, without raising an error. There is no need to delete the file or add overwrite guard code beforehand. By default (controlled by the `USE_TMP_FILE` option), DuckDB writes to a temporary file first and only replaces an existing destination once the write completes successfully, so an interrupted write does not corrupt the previous file. Reading from and writing to the same file in a single statement is not guaranteed to be safe, however, and should be avoided.

> New Support for `COPY ...` blocks in CTEs was introduced in DuckDB 2.0. For other data-modifying statements in CTEs, see Data-Modifying CTEs.

`COPY ... TO` can also be used as a CTE body. The CTE returns the number of rows written. `COPY ... FROM` cannot be used as a CTE body.

```
WITH copied(rows_written) AS (
    COPY lineitem TO 'lineitem.csv' (FORMAT CSV, HEADER)
)
SELECT rows_written FROM copied;
```

Copy the contents of the `lineitem` table to a CSV file with a header:

```
COPY lineitem TO 'lineitem.csv';
```

Copy the contents of the `lineitem` table to the file `lineitem.tbl`, where the columns are delimited by a pipe character (`|`), including a header line:

```
COPY lineitem TO 'lineitem.tbl' (DELIMITER '|');
```

Use tab separators to create a TSV file without a header:

```
COPY lineitem TO 'lineitem.tsv' (DELIMITER '\t', HEADER false);
```

Copy the `l_orderkey` column of the `lineitem` table to the file `orderkey.tbl`:

```
COPY lineitem(l_orderkey) TO 'orderkey.tbl' (DELIMITER '|');
```

Copy the result of a query to the file `query.csv`, including a header with column names:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.csv' (DELIMITER ',');
```

Copy the result of a query to the Parquet file `query.parquet`:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.parquet' (FORMAT parquet);
```

Copy the result of a query to the newline-delimited JSON file `query.ndjson`:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.ndjson' (FORMAT json);
```

Copy the result of a query to the JSON file `query.json`:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.json' (FORMAT json, ARRAY true);
```

Return the files and their column statistics that were written as part of the COPY statement:

```
COPY (SELECT l_orderkey, l_comment FROM lineitem) TO 'lineitem_part.parquet' (RETURN_STATS);
```

filename	count	file_size_bytes	footer_size_bytes	column_statistics	partition_keys
lineitem_part.parquet	600572	8579141	1445	{'"l_comment"'={column_size_bytes=7642227, max=zzle. slyly, min=' Tiresias above the blit', null_count=0}, '"l_orderkey"'={column_size_bytes=935457, max=600000, min=1, null_count=0}}	NULL

Note: for nested columns (e.g., structs) the column statistics are defined for each part. For example, if we have a column `name STRUCT(field1 INTEGER, field2 INTEGER)` the column statistics will have stats for `name.field1` and `name.field2`.

An expression may be used as the target of a `COPY ... TO` command if it is placed within parentheses.

Copy the result of a query to a file whose path is stored in a variable:

```
SET VARIABLE target_file = 'target_file.parquet';
COPY (SELECT 'hello world') TO (getvariable('target_file'));
```

Copy to a file provided as parameter of a prepared statement:

```
PREPARE v1 AS COPY (SELECT 42 AS i) to $1;
EXECUTE v1('file.csv');
```

Expressions may be used for options as well. Copy to a file using a format stored in a variable:

```
SET VARIABLE my_format = 'parquet';
COPY (SELECT 42 AS i) TO 'file' (FORMAT getvariable('my_format'));
```

`COPY ... TO` Options

Zero or more copy options may be provided as a part of the copy operation. The `WITH` specifier is optional, but if any options are specified, the parentheses are required. Parameter values can be passed in with or without wrapping in single quotes. Arbitrary expressions may be used for parameter values.

Any option that is a Boolean can be enabled or disabled in multiple ways. You can write `true`, `ON`, or `1` to enable the option, and `false`, `OFF`, or `0` to disable it. The `BOOLEAN` value can also be omitted, e.g., by only passing `(HEADER)`, in which case `true` is assumed.

With few exceptions, the below options are applicable to all formats written with `COPY`.

Name	Description	Type	Default
FORMAT	Specifies the copy function to use. The default is selected from the file extension (e.g., `.parquet` results in a Parquet file being written/read). If the file extension is unknown CSV is selected. Vanilla DuckDB provides CSV, PARQUET and JSON but additional copy functions can be added by extensions.	VARCHAR	auto
USE_TMP_FILE	Whether or not to write to a temporary file first if the original file exists (`target.csv.tmp`). This prevents overwriting an existing file with a broken file in case the writing is cancelled.	BOOL	auto

Name	Description	Type	Default
OVERWRITE_OR_IGNORE	Whether or not to allow overwriting files if they already exist. Only has an effect when used with options that write multiple files, such as PARTITION_BY, PER_THREAD_OUTPUT or FILE_SIZE_BYTES.	BOOL	false
OVERWRITE	When true, all existing files inside targeted directories will be removed (not supported on remote filesystems). Only has an effect when used with options that write multiple files, such as PARTITION_BY, PER_THREAD_OUTPUT or FILE_SIZE_BYTES.	BOOL	false
APPEND	When true, in the event a filename pattern is generated that already exists, the path will be regenerated to ensure no existing files are overwritten. Only has an effect when used with options that write multiple files, such as PARTITION_BY, PER_THREAD_OUTPUT or FILE_SIZE_BYTES.	BOOL	false
FILENAME_PATTERN	Set a pattern to use for the filename, can optionally contain {uuid} / {uuidv4} or {uuidv7} to be filled in with a generated UUID (v4 or v7, respectively), and {i}, which is replaced by an incrementing index. Only has an effect when used with options that write multiple files, such as PARTITION_BY, PER_THREAD_OUTPUT or FILE_SIZE_BYTES.	VARCHAR	auto
FILE_EXTENSION	Set the file extension that should be assigned to the generated file(s).	VARCHAR	auto
PER_THREAD_OUTPUT	When true, the COPY command generates one file per thread, rather than one file in total. This allows for faster parallel writing.	BOOL	false
FILE_SIZE_BYTES	If this parameter is set, the COPY process creates a directory which will contain the exported files. If a file exceeds the set limit (specified as bytes such as 1000 or in human-readable format such as 1k), the process creates a new file in the directory. This parameter works in combination with PER_THREAD_OUTPUT. Note that the size is used as an approximation, and files can be occasionally slightly over the limit.	VARCHAR or BIGINT	(empty)
PARTITION_BY	The columns to partition by using a Hive partitioning scheme, see the partitioned writes section.	VARCHAR[]	(empty)
PARTITION_PATH	An expression over the PARTITION_BY columns that computes the output subdirectory for each partition, replacing the default col=value Hive layout. The result must be a relative path (no leading /, no .. segments) and non-empty. Requires PARTITION_BY and cannot be combined with HIVE_FILE_PATTERN. See the partitioned writes section. Added in DuckDB 2.0.	VARCHAR	(empty)

Name	Description	Type	Default
ORDER_BY	A parenthesized list of expressions to sort the rows by before writing, each optionally followed by ASC or DESC and NULLS FIRST or NULLS LAST. When combined with PARTITION_BY, the rows are sorted within each partition. Added in DuckDB 2.0.	expression list	(empty)
PRESERVE_ORDER	Whether or not to preserve order during the copy operation. Defaults to the value of the preserve_insertion_order configuration option.	BOOL	(*)
RETURN_FILES	Whether or not to include the created filepath(s) (as a files VARCHAR[] column) in the query result.	BOOL	false
RETURN_STATS	Whether or not to return the files and their column statistics that were written as part of the COPY statement.	BOOL	false
WRITE_PARTITION_COLUMNS	Whether or not to write partition columns into files. Only has an effect when used with PARTITION_BY.	BOOL	false

Syntax

> To ensure compatibility with PostgreSQL, DuckDB accepts COPY ... TO statements that do not fully comply with the railroad diagram shown here. For example, the following is a valid statement:
>
> ```
> COPY (SELECT 42 AS x, 84 AS y) TO 'out.csv' WITH DELIMITER '|' CSV HEADER;
> ```

COPY FROM DATABASE ... TO

The COPY FROM DATABASE ... TO statement copies the entire content from one attached database to another attached database. This includes the schema, including constraints, indexes, sequences, macros and the data itself.

```
ATTACH 'db1.db' AS db1;
CREATE TABLE db1.tbl AS SELECT 42 AS x, 3 AS y;
CREATE MACRO db1.two_x_plus_y(x, y) AS 2 * x + y;

ATTACH 'db2.db' AS db2;
COPY FROM DATABASE db1 TO db2;
SELECT db2.two_x_plus_y(x, y) AS z FROM db2.tbl;
```

z
87

To only copy the **schema** of db1 to db2 but omit copying the data, add SCHEMA to the statement:

```
COPY FROM DATABASE db1 TO db2 (SCHEMA);
```

Syntax

Format-Specific Options

CSV Options

The below options are applicable when writing CSV files.

Name	Description	Type	Default
COMPRESSION	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.csv.gz` will use `gzip`, `file.csv.zst` will use `zstd`, and `file.csv` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`
DATEFORMAT	Specifies the date format to use when writing dates. See Date Format.	VARCHAR	(empty)
DELIM or SEP	The character that is written to separate columns within each row.	VARCHAR	`,`
NEW_LINE	The character that is written to separate each row. Use escaped strings, e.g. `E'\x1e'`	VARCHAR	`\n`
ESCAPE	The character that should appear before a character that matches the `quote` value.	VARCHAR	`"`
FORCE_QUOTE	The list of columns to always add quotes to, even if not required.	VARCHAR[]	`[]`
HEADER	Whether or not to write a header for the CSV file.	BOOL	`true`
NULLSTR	The string that is written to represent a `NULL` value.	VARCHAR	(empty)
PREFIX	Prefixes the CSV file with a specified string. This option must be used in conjunction with SUFFIX and requires HEADER to be set to `false`.	VARCHAR	(empty)
SUFFIX	Appends a specified string as a suffix to the CSV file. This option must be used in conjunction with PREFIX and requires HEADER to be set to `false`.	VARCHAR	(empty)
QUOTE	The quoting character to be used when a data value is quoted.	VARCHAR	`"`
TIMESTAMPFORMAT	Specifies the date format to use when writing timestamps. See Date Format.	VARCHAR	(empty)

Parquet Options

The below options are applicable when writing Parquet files.

Name	Description	Type	Default
COMPRESSION	The compression format to use (`uncompressed`, `snappy`, `gzip`, `zstd`, `brotli`, `lz4`, `lz4_raw`).	VARCHAR	`snappy`
COMPRESSION_LEVEL	Compression level, set between 1 (lowest compression, fastest) and 22 (highest compression, slowest). Only supported for zstd compression.	BIGINT	3
FIELD_IDS	The `field_id` for each column. Pass `auto` to attempt to infer automatically.	STRUCT	(empty)

Name	Description	Type	Default
ROW_GROUP_SIZE_BYTES	The target size of each row group. You can pass either a human-readable string, e.g., 2MB, or an integer, i.e., the number of bytes. This option is only used when you have issued SET preserve_insertion_order = false;{:.language-sql .highlight}, otherwise, it is ignored.	BIGINT	row_group_size * 1024
ROW_GROUP_SIZE	The target size, i.e., number of rows, of each row group.	BIGINT	122880
ROW_GROUPS_PER_FILE	Create a new Parquet file if the current one has a specified number of row groups. If multiple threads are active, the number of row groups in a file may slightly exceed the specified number of row groups to limit the amount of locking – similarly to the behavior of FILE_SIZE_BYTES. However, if per_thread_output is set, only one thread writes to each file, and it becomes accurate again.	BIGINT	(empty)
PARQUET_VERSION	The Parquet version to use (V1, V2).	VARCHAR	V1
CHUNK_SIZE	Alias for ROW_GROUP_SIZE.	BIGINT	122880
KV_METADATA	Custom key-value metadata to embed in the file footer, supplied as a STRUCT of keys to values. BLOB values are written as raw bytes; other values are cast to string.	STRUCT	(empty)
SHREDDING	A STRUCT mapping VARIANT column names to the type they should be shredded into, e.g., {variant_col: 'STRUCT(name VARCHAR, age INTEGER)'}. Enables typed (shredded) storage of VARIANT columns.	STRUCT	(empty)
DATA_PAGE_SIZE_LIMIT	Target uncompressed size of Parquet data pages, in bytes. Pages may still be larger at vector granularity.	UBIGINT	104857600 (100 MiB)
DICTIONARY_SIZE_LIMIT	The maximum size of the dictionary used for dictionary encoding, in number of distinct values. Set to 0 to disable dictionary encoding.	BIGINT	ROW_GROUP_SIZE / 5
WRITE_BLOOM_FILTER	Whether to write Bloom filters that allow readers to skip row groups.	BOOLEAN	true
BLOOM_FILTER_FALSE_POSITIVE_RATIO	The target false positive ratio of the written Bloom filters.	DOUBLE	0.01
GEOPARQUET_VERSION	Which GeoParquet metadata version to write for geometry columns (NONE, V1, V2, BOTH). NONE disables GeoParquet metadata.	VARCHAR	V1
TIMESTAMP_IS_ADJUSTED_TO_UTC	Controls the isAdjustedToUTC flag written for timestamp columns. With AUTO, the flag is set to true for TIMESTAMP WITH TIME ZONE columns and false for other timestamp types. Added in DuckDB 2.0.	VARCHAR (AUTO, TRUE, FALSE)	AUTO
WRITE_TIMESTAMP_AS_INT96	When true, writes TIMESTAMP columns using the legacy Parquet INT96 physical type for compatibility with older readers. Added in DuckDB 2.0.	BOOLEAN	false

> New As of DuckDB 2.0, the TIMESTAMP_IS_ADJUSTED_TO_UTC and WRITE_TIMESTAMP_AS_INT96 options control how timestamp columns are written to Parquet files.

Some examples of FIELD_IDS are as follows.

Assign `field_ids` automatically:

```
COPY
    (SELECT 128 AS i)
    TO 'my.parquet'
    (FIELD_IDS 'auto');
```

Sets the `field_id` of column `i` to 42:

```
COPY
    (SELECT 128 AS i)
    TO 'my.parquet'
    (FIELD_IDS {i: 42});
```

Sets the `field_id` of column `i` to 42, and column `j` to 43:

```
COPY
    (SELECT 128 AS i, 256 AS j)
    TO 'my.parquet'
    (FIELD_IDS {i: 42, j: 43});
```

Sets the `field_id` of column `my_struct` to 42, and column `i` (nested inside `my_struct`) to 43:

```
COPY
    (SELECT {i: 128} AS my_struct)
    TO 'my.parquet'
    (FIELD_IDS {my_struct: {__duckdb_field_id: 42, i: 43}});
```

Sets the `field_id` of column `my_list` to 42, and column `element` (default name of list child) to 43:

```
COPY
    (SELECT [128, 256] AS my_list)
    TO 'my.parquet'
    (FIELD_IDS {my_list: {__duckdb_field_id: 42, element: 43}});
```

Sets the `field_id` of column `my_map` to 42, and columns `key` and `value` (default names of map children) to 43 and 44:

```
COPY
    (SELECT MAP {'key1' : 128, 'key2': 256} my_map)
    TO 'my.parquet'
    (FIELD_IDS {my_map: {__duckdb_field_id: 42, key: 43, value: 44}});
```

JSON Options

The below options are applicable when writing JSON files.

Name	Description	Type	Default
ARRAY	Whether to write a JSON array. If `true`, a JSON array of records is written, if `false`, newline-delimited JSON is written	BOOL	`false`
COMPRESSION	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.json.gz` will use `gzip`, `file.json.zst` will use `zstd`, and `file.json` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`
DATEFORMAT	Specifies the date format to use when writing dates. See Date Format.	VARCHAR	(empty)
TIMESTAMPFORMAT	Specifies the date format to use when writing timestamps. See Date Format.	VARCHAR	(empty)

Sets the value of column `hello` to `QUACK!` and outputs the results to `quack.json`:

```
COPY (SELECT 'QUACK!' AS hello) TO 'quack.json';
--RETURNS: {"hello":"QUACK!"}
```

Sets the value of column `num_list` to `[1,2,3]` and outputs the results to `numbers.json`:

```
COPY (SELECT [1, 2, 3] AS num_list) TO 'numbers.json';
--RETURNS: {"num_list":[1,2,3]}
```

Sets the value of column `compression_type` to `gzip_explicit` and outputs the results to `compression.json.gz` with explicit compression:

```
COPY (SELECT 'gzip_explicit' AS compression_type) TO 'explicit_compression.json' (FORMAT json, COMPRESSION
'GZIP');
-- RETURNS: {"compression_type":"gzip_explicit"}
```

Sets all values of single rows to be returned as nested arrays to `array_true.json`:

```
COPY (SELECT 1 AS id, 'Alice' AS name, [1, 2, 3] AS numbers
      UNION ALL
      SELECT 2, 'Bob', [4, 5, 6] AS numbers)
TO 'array_true.json' (FORMAT json, ARRAY true);

-- RETURNS:
/*
[
    {"id":1,"name":"Alice","numbers":[1,2,3]},
    {"id":2,"name":"Bob","numbers":[4,5,6]}
]
*/
```

Sets all values of single rows to be returned as non-nested arrays to `array_false.json`:

```
COPY (SELECT 1 AS id, 'Alice' AS name, [1, 2, 3] AS numbers
      UNION ALL
      SELECT 2, 'Bob', [4, 5, 6] AS numbers)
TO 'array_false.json' (FORMAT json, ARRAY false);

-- RETURNS:
/*
{"id":1,"name":"Alice","numbers":[1,2,3]}
{"id":2,"name":"Bob","numbers":[4,5,6]}
*/
```

BLOB Options

The BLOB format option allows you to select a single column of a DuckDB table into a `.blob` file. The column must be cast to the BLOB data type. For details on typecasting, see the Casting Operations Matrix.

The below options are applicable when writing BLOB files.

Name	Description	Type	Default
COMPRESSION	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.blob.gz` will use `gzip`, `file.blob.zst` will use `zstd`, and `file.blob` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`

Type casts the string value `foo` to the BLOB data type and outputs the results to `blob_output.blob`:

```
COPY (select 'foo'::BLOB) TO 'blob_output.blob' (FORMAT BLOB);
```

Type casts the string value `foo` to the `BLOB` data type and outputs the results to `blob_output_gzip.blob.gz` with `gzip` compression:

```
COPY (select 'foo'::BLOB) TO 'blob_output_gzip.blob' (FORMAT BLOB, COMPRESSION 'GZIP');
```

Limitations

COPY does not support copying between tables. To copy between tables, use an `INSERT statement`:

```
INSERT INTO tbl2
    FROM tbl1;
```

CREATE MACRO Statement

The `CREATE MACRO` statement defines a named, callable SQL expression as a database schema object.

Macros may declare parameters, which can be referenced by its expression. Once a macro is created, it may be called by referencing its name and by passing values to its parameters; this causes its expression to be evaluated, yielding a value. Depending on the macro type, the value may be scalar, or a `TABLE` value. The `CREATE FUNCTION` statement is an alias for `CREATE MACRO`.

The simplified syntax for creating a macro is:

```
CREATE [OR REPLACE] [TEMPORARY] MACRO [IF NOT EXISTS] <identifier>( [<parameters>] ) AS [TABLE]
<expression>;
```

- The identifier consists of the macro's name, which can be any valid SQL identifier. A macro may be explicitly qualified with an existing database schema. If a schema is specified, then it appears in the usual way: before the name, with a dot separating schema name and macro name. When not explicitly specified, the macro will be associated with the current schema.
- When connected to a persistent database file, the macro will be stored in the database. The optional `TEMPORARY` keyword indicates that the macro is not to be persisted.
- An `OR REPLACE` clause may occur immediately after the `CREATE` keyword, which causes an existing macro with the same name (within the schema) to be overwritten. Without `OR REPLACE` such an attempt will fail with a `Macro Function already exists` error.
- An `IF NOT EXISTS` clause may occur immediately before the identifier, and has the effect of only creating the macro if it does not already exist. Either the `OR REPLACE` or the `IF NOT EXISTS` clause may be present, but not both.
- The macro name is followed by parentheses. If the macro has parameters, then they must be declared within the parentheses.
- The `AS` keyword appears after the right parenthesis but before the expression
- The `TABLE` keyword may appear right after the `AS` keyword, directly before the expression, indicating the macro is a table macro and returns a result set. When omitted, the macro is automatically a scalar macro.
- The expression can be any valid SQL expression provided the expression's type is aligned with the macro's type.

Please review the syntax diagram for a more precise and detailed overview of the `CREATE MACRO` statement.

Types of Macros

The contexts where a particular macro may be called depends on the data type of its result value:

- Scalar macros evaluate to a scalar value. For scalar macros, the expression can either be a simple expression, or a scalar subquery.
- Table macros return a tabular result: when called, they act essentially as table functions and return a table value. Their expression can be a `SELECT` statement, or a call to another table function.

Declaring Parameters

Macros may declare parameters. The parameter declarations appear as a comma-separated list between the parentheses before the AS keyword. The simplified syntax for a single parameter declaration is:

```
<parameter-name> [<datatype>] [ := <default-value> ]
```

- The parameter name is mandatory and can be any valid SQL identifier. Parameter names must be unique within the parameter list. An attempt to define multiple parameters with the same name results in a `Duplicate parameter` error.
- Optionally, a parameter may explicitly specify a particular datatype. This can be any of the existing DuckDB data types. Note: there is no way to specify a parameter of a TABLE type.
- A parameter can optionally specify a default value. This is done with the assignment operator `:=`, followed by the expression that is to be used as default value.
- The default value expression is in principle evaluated at definition-time - NOT at run-time. (There are a few exceptions, like CURRENT_SCHEMA. But it's best not to rely on that: if you need a default value to be dynamic, use a well-known value like NULL as default and use conditional logic in the expression to produce the run-time value).
- Specifying a default value expression effectively makes the parameter optional: when the macro is called, the DuckDb binder will find candidate signatures based on passed parameters, but backfilled by signatures that specify default values for missing parameters.

- A parameter that specifies a default value cannot appear before a definition of a parameter that does not have a default value. In other words, parameters without default values have to be defined ”in the front”; any parameters with default values appear ”at the back”.
- Multiple parameter declarations are separated from one another with a comma.

Overloading

Macros support overloading:

- A single CREATE MACRO statement can define multiple implementations (sometimes called 'overloads'), each having its own parameter list, AS keyword, and expression. Note that all of the implementations are defined in the same CREATE MACRO statement: it is not possible to add, remove, or alter individual implementations after the macro is created.
- Multiple implementations are separated from one another by a comma.
- Each implementation must have a unique parameter-type signature: that is, in one CREATE MACRO statement, all implementations having the same number of parameters must each have a unique sequence of parameter types - regardless of the parameter names. If a parameter type signature is not unique, it results in a `Ambiguity in macro overloads` error.
- Overloading only applies to the parameter-types, but not to the macro type itself: for a single macro, all of its implementations are either scalar or TABLE.
- When overloading table functions that are defined using a SELECT statement as expression, you will probably need to wrap the SELECT statement in parentheses.

Calling Macros

Macros are called by mentioning their name, followed by parentheses. A comma-separated list of value-expressions may appear between the parentheses - these are the actual parameters. The DuckDB binder examines the data types of the actual parameters and tries to find an implementation of the macro that has a matching signature. If an implementation is found, the parameter values are passed and the implementation's expression is evaluated; finally, its resulting value is returned and used in place where the macro was called. This is similar to calling a function.

In general, a call to a macro is valid if its expression could also appear in that context:

- A call to a scalar macro can be used in the SELECT clause or in the WHERE clause of a SELECTstatement.
- If the expression of a scalar macro references an aggregate function, then the macro behaves like an aggregate function too.
- A call to a table macro can appear in the FROM clause of a SELECT statement, or in a CALL statement.

Passing Parameters

Parameter values can be passed to the macro as a comma-separated list of expressions appearing between the parentheses following the macro's name.

Parameter values may be passed either positionally or by name.

- Positional parameter passing means that only the value expression is passed;
- In contrast to positional parameter passing, named parameter passing explicitly assigns the actual parameter value to a specific formal parameter. This is done by mentioning the parameter name followed by the assignment operator `:=`, followed by the parameter value expression.
- Note that some built-in functions also allow named parameters, but allow = as assignment operator. For macros, this won't work! Instead, the = is interpreted as comparison operator. This effectively turns the - intended - named parameter into a positional parameter passing the result of comparing the parameter name with the parameter value expression.
- If a macro call contains named parameters, then they must appear after any positional parameters. In other words, any positional parameters must appear "in the front"; while all named parameters must appear "at the back".
- By definition, positional parameters are passed in the same order as they were declared. But named parameters can appear in any order (provided they appear after any of the positional parameters).

Examples

Scalar Macros

Create a macro that adds two expressions (a and b):

```
CREATE MACRO add(a, b) AS a + b;
```

Create a macro, replacing possible existing definitions:

```
CREATE OR REPLACE MACRO add(a, b) AS a + b;
```

Create a macro if it does not already exist, else do nothing:

```
CREATE MACRO IF NOT EXISTS add(a, b) AS a + b;
```

Create a macro for a CASE expression:

```
CREATE MACRO ifelse(a, b, c) AS CASE WHEN a THEN b ELSE c END;
```

Create a macro that does a subquery:

```
CREATE MACRO one() AS (SELECT 1);
```

Macros are schema-dependent, and have an alias, `FUNCTION`:

```
CREATE FUNCTION main.my_avg(x) AS sum(x) / count(x);
```

Create a macro with a default parameter:

```
CREATE MACRO add_default(a, b := 5) AS a + b;
```

Create a macro `arr_append` (with functionality equivalent to `array_append`):

```
CREATE MACRO arr_append(l, e) AS list_concat(l, list_value(e));
```

Create a macro with a typed parameter:

```
CREATE MACRO is_maximal(a INTEGER) AS a = 2^31 - 1;
```

Table Macros

Create a table macro without parameters:

```
CREATE MACRO static_table() AS TABLE
    SELECT 'Hello' AS column1, 'World' AS column2;
```

Create a table macro with parameters (that can be of any type):

```
CREATE MACRO dynamic_table(col1_value, col2_value) AS TABLE
    SELECT col1_value AS column1, col2_value AS column2;
```

Create a table macro that returns multiple rows. It will be replaced if it already exists, and it is temporary (will be automatically deleted when the connection ends):

```
CREATE OR REPLACE TEMP MACRO dynamic_table(col1_value, col2_value) AS TABLE
    SELECT col1_value AS column1, col2_value AS column2
    UNION ALL
    SELECT 'Hello' AS col1_value, 456 AS col2_value;
```

Pass an argument as a list:

```
CREATE MACRO get_users(i) AS TABLE
    SELECT * FROM users WHERE uid IN (SELECT unnest(i));
```

An example of how to use the `get_users` table macro is the following:

```
CREATE TABLE users AS
    SELECT *
    FROM (VALUES (1, 'Ada'), (2, 'Bob'), (3, 'Carl'), (4, 'Dan'), (5, 'Eve')) t(uid, name);
SELECT * FROM get_users([1, 5]);
```

To define macros on arbitrary tables, use the `query_table` function. For example, the following macro computes a column-wise checksum on a table:

```
CREATE MACRO checksum(tbl) AS TABLE
    SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
    FROM query_table(tbl);

CREATE TABLE tbl AS SELECT unnest([42, 43]) AS x, 100 AS y;
SELECT * FROM checksum('tbl');
```

A table macro's expression is bound when the macro is created, so file names written directly into the expression are resolved at creation time. For example, the following fails at creation time when the file is not present:

```
CREATE MACRO read_pipe() AS TABLE FROM read_csv('/dev/stdin');
```

```
IO Error:
No files found that match the pattern "/dev/stdin"
```

To resolve the file name at call time instead, pass it as a parameter. Supplying a default value keeps the macro callable without arguments:

```
CREATE OR REPLACE TEMPORARY MACRO read_pipe(pipe := '/dev/stdin') AS TABLE
    FROM read_csv(pipe);
FROM read_pipe();
```

Overloading

It is possible to overload a macro based on the types or the number of its parameters; this works for both scalar and table macros.

By providing overloads we can have both `add_x(a, b)` and `add_x(a, b, c)` with different function bodies.

```
CREATE MACRO add_x
    (a, b) AS a + b,
    (a, b, c) AS a + b + c;
```

```
SELECT
    add_x(21, 42) AS two_args,
    add_x(21, 42, 21) AS three_args;
```

two_args	three_args
63	84

```
CREATE OR REPLACE MACRO is_maximal
    (a TINYINT) AS a = 2^7 - 1,
    (a INT) AS a = 2^31 - 1;

SELECT
    is_maximal(127::TINYINT) AS tiny,
    is_maximal(127) AS regular;
```

tiny	regular
true	false

Table macros can be overloaded in the same way. When an implementation is defined using a SELECT statement, wrap that statement in parentheses:

```
CREATE MACRO tbl_range
    (n) AS TABLE (SELECT * FROM range(n)),
    (start, stop) AS TABLE (SELECT * FROM range(start, stop));

SELECT count(*) AS one_arg FROM tbl_range(3);
SELECT count(*) AS two_args FROM tbl_range(3, 8);
```

The first call returns 3, the second returns 5.

Syntax

Macros allow you to create shortcuts for combinations of expressions.

```
CREATE MACRO add(a) AS a + b;
```

```
Binder Error:
Referenced column "b" not found in FROM clause!
```

This works:

```
CREATE MACRO add(a, b) AS a + b;
```

Usage example:

```
SELECT add(1, 2) AS x;
```

x
3

However, this fails:

```
SELECT add('hello', 3);
```

```
Binder Error:
Could not choose a best candidate function for the function call "add(STRING_LITERAL, INTEGER_LITERAL)". In
order to select one, please add explicit type casts.
    Candidate functions:
    add(DATE, INTEGER) -> DATE
    add(INTEGER, INTEGER) -> INTEGER
```

Macros can have default parameters.

b is a default parameter:

```
CREATE MACRO add_default(a, b := 5) AS a + b;
```

The following will result in 42:

```
SELECT add_default(37);
```

The order of named parameters does not matter:

```
CREATE MACRO triple_add(a, b := 5, c := 10) AS a + b + c;
```

```
SELECT triple_add(40, c := 1, b := 1) AS x;
```

x
42

When macros are used, they are expanded (i.e., replaced with the original expression), and the parameters within the expanded expression are replaced with the supplied arguments. Step by step:

The add macro we defined above is used in a query:

```
SELECT add(40, 2) AS x;
```

Internally, add is replaced with its definition of a + b:

```
SELECT a + b AS x;
```

Then, the parameters are replaced by the supplied arguments:

```
SELECT 40 + 2 AS x;
```

Limitations

Using Subquery Macros

Table macros as well as scalar macros defined using scalar subqueries cannot be used in the arguments of table functions. DuckDB will return the following error:

```
Binder Error:
Table function cannot contain subqueries
```

Overloads

Overloads for macro functions have to be set at creation, it is not possible to define a macro by the same name twice without first removing the first definition.

Recursive Functions

Defining recursive functions is not supported. For example, the following macro – supposed to compute the *n*th number of the Fibonacci sequence – fails:

```
CREATE OR REPLACE FUNCTION fibo(n) AS (SELECT 1);
CREATE OR REPLACE FUNCTION fibo(n) AS (
    CASE
        WHEN n <= 1 THEN 1
        ELSE fibo(n - 1)
    END
);
SELECT fibo(3);
```

```
Binder Error:
Max expression depth limit of 1000 exceeded. Use "SET max_expression_depth TO x" to increase the maximum
expression depth.
```

Viewing the List of Macros and Table Macros

You can use the following query to display the list of macros and table macros:

```
SELECT schema_name, function_name, function_type, parameters
FROM duckdb_functions();
```

CREATE SCHEMA Statement

The `CREATE SCHEMA` statement creates a schema in the catalog. The default schema is `main`.

Examples

Create a schema:

```
CREATE SCHEMA s1;
```

Create a schema if it does not exist yet:

```
CREATE SCHEMA IF NOT EXISTS s2;
```

Create a schema or replace a schema if it exists:

```
CREATE OR REPLACE SCHEMA s2;
```

Create table in the schemas:

```
CREATE TABLE s1.t (id INTEGER PRIMARY KEY, other_id INTEGER);
CREATE TABLE s2.t (id INTEGER PRIMARY KEY, j VARCHAR);
```

Compute a join between tables from two schemas:

```
SELECT *
FROM s1.t s1t, s2.t s2t
WHERE s1t.other_id = s2t.id;
```

Nested Schemas

> New Support for nested schemas was introduced in DuckDB 2.0.

Schemas can be nested within other schemas. Objects in a nested schema are referenced using their fully qualified, dot-separated path:

```
CREATE SCHEMA finance;
CREATE SCHEMA finance.reports;
CREATE TABLE finance.reports.q3 (revenue DECIMAL);
```

Syntax

CREATE SECRET Statement

The `CREATE SECRET` statement creates a new secret in the Secrets Manager.

Syntax for CREATE SECRET

> **Warning.** When using the command line client, the `CREATE SECRET` statements are stored in your DuckDB history as plain text.

Syntax for DROP SECRET

CREATE SEQUENCE Statement

The `CREATE SEQUENCE` statement creates a new sequence number generator.

Examples

Generate an ascending sequence starting from 1:

```
CREATE SEQUENCE serial;
```

Generate sequence from a given start number:

```
CREATE SEQUENCE serial START 101;
```

Generate odd numbers using `INCREMENT BY`:

```
CREATE SEQUENCE serial START WITH 1 INCREMENT BY 2;
```

Generate a descending sequence starting from 99:

```
CREATE SEQUENCE serial START WITH 99 INCREMENT BY -1 MAXVALUE 99;
```

By default, cycles are not allowed and will result in error, e.g.:

```
Sequence Error:
nextval: reached maximum value of sequence "serial" (10)
```

```
CREATE SEQUENCE serial START WITH 1 MAXVALUE 10;
```

`CYCLE` allows cycling through the same sequence repeatedly:

```
CREATE SEQUENCE serial START WITH 1 MAXVALUE 10 CYCLE;
```

Creating and Dropping Sequences

Sequences can be created and dropped similarly to other catalog items.

Overwrite an existing sequence:

```
CREATE OR REPLACE SEQUENCE serial;
```

Only create sequence if no such sequence exists yet:

```
CREATE SEQUENCE IF NOT EXISTS serial;
```

Remove sequence:

```
DROP SEQUENCE serial;
```

Remove sequence if exists:

```
DROP SEQUENCE IF EXISTS serial;
```

Using Sequences for Primary Keys

Sequences can be used as `DEFAULT` values in `CREATE TABLE` statements.

The example below uses a sequence to create an integer primary key:

```
CREATE SEQUENCE id_sequence START 1;
CREATE TABLE tbl (id INTEGER PRIMARY KEY DEFAULT nextval('id_sequence'), s VARCHAR);
INSERT INTO tbl (s) VALUES ('hello'), ('world');
SELECT * FROM tbl;
```

The script results in the following table:

id	s
1	hello
2	world

Sequences can also be added using the `ALTER TABLE` statement. The following example adds an `id` column and fills it with values generated by the sequence:

```
CREATE TABLE tbl (s VARCHAR);
INSERT INTO tbl VALUES ('hello'), ('world');
CREATE SEQUENCE id_sequence START 1;
ALTER TABLE tbl ADD COLUMN id INTEGER DEFAULT nextval('id_sequence');
SELECT * FROM tbl;
```

This script results in the same table as the previous example.

Selecting the Next Value

To select the next number from a sequence, use `nextval`:

```
CREATE SEQUENCE serial START 1;
SELECT nextval('serial') AS nextval;
```

nextval
1

Using this sequence in an `INSERT` command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Selecting the Current Value

You may also view the current number from the sequence. Note that the `nextval` function must have already been called before calling `currval`, otherwise a Serialization Error (`sequence is not yet defined in this session`) will be thrown.

```
CREATE SEQUENCE serial START 1;
SELECT nextval('serial') AS nextval;
SELECT currval('serial') AS currval;
```

currval
1

Setting the Sequence Value

> New This feature was introduced in DuckDB 2.0.

Use `setval` to set a sequence to a given value (PostgreSQL-style). The optional third argument `is_called` defaults to `true`: the next `nextval` returns `value + increment`. If `is_called` is `false`, the next `nextval` returns `value`.

```
CREATE SEQUENCE serial START 1;
SELECT setval('serial', 42);
SELECT nextval('serial');
```

nextval
43

```
CREATE SEQUENCE serial START 1;
SELECT setval('serial', 42, false);
SELECT nextval('serial');
```

nextval
42

Syntax

`CREATE SEQUENCE` creates a new sequence number generator.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence in the same schema.

After a sequence is created, you use the function `nextval` to operate on the sequence.

Parameters

Name	Description
`CYCLE` or `NO CYCLE`	The `CYCLE` option allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively. If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If neither `CYCLE` nor `NO CYCLE` are specified, `NO CYCLE` is the default.
`increment`	The optional clause `INCREMENT BY increment` specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

Name	Description
`maxvalue`	The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 2^63 - 1 and -1 for ascending and descending sequences, respectively.
`minvalue`	The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -(2^63 - 1) for ascending and descending sequences, respectively.
`name`	The name (optionally schema-qualified) of the sequence to be created.
`start`	The optional clause `START WITH start` allows the sequence to begin anywhere. The default starting value is `minvalue` for ascending sequences and `maxvalue` for descending ones.
TEMPORARY or TEMP	If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

> Sequences are based on `BIGINT` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Limitations

A sequence cannot be dropped while a column `DEFAULT` still references it:

```
CREATE SEQUENCE id_sequence START 1;

CREATE TABLE tbl (
    id INTEGER DEFAULT nextval('id_sequence'),
    s VARCHAR
);
INSERT INTO tbl(s) VALUES ('default is the next value from id_sequence');
```

Attempting to drop the sequence results in an error:

```
DROP SEQUENCE id_sequence;
```

```
Dependency Error:
Cannot drop entry "id_sequence" because there are entries that depend on it.
table "tbl" depends on sequence "id_sequence".
Use DROP...CASCADE to drop all dependents.
```

As the error message suggests, you can force dropping by adding CASCADE. However, DuckDB currently tracks dependencies at the table level, so attempting to drop with CASCADE drops the entire table:

```
DROP SEQUENCE id_sequence CASCADE;
SELECT * FROM tbl;
```

```
Catalog Error:
Table with name tbl does not exist!
```

For the same reason, `CREATE OR REPLACE SEQUENCE` fails when the sequence is referenced by another object, such as a column `DEFAULT` value, because the replacement first drops the existing sequence:

```
CREATE OR REPLACE SEQUENCE id_sequence;
```

```
Dependency Error:
Cannot drop entry "id_sequence" because there are entries that depend on it.
table "tbl" depends on sequence "id_sequence".
Use DROP...CASCADE to drop all dependents.
```

To replace the sequence, first remove the dependency, for example by dropping the column default:

```
ALTER TABLE tbl ALTER COLUMN id DROP DEFAULT;
CREATE OR REPLACE SEQUENCE id_sequence;
```

CREATE TABLE Statement

The `CREATE TABLE` statement creates a table in the catalog.

Examples

Create a table with two integer columns (`i` and `j`):

```
CREATE TABLE t1 (i INTEGER, j INTEGER);
```

Create a table with a primary key:

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, j VARCHAR);
```

Create a table with a composite primary key:

```
CREATE TABLE t1 (id INTEGER, j VARCHAR, PRIMARY KEY (id, j));
```

Create a table with various different types, constraints and default values:

```
CREATE TABLE t1 (
    i INTEGER NOT NULL DEFAULT 0,
    decimalnr DOUBLE CHECK (decimalnr < 10),
    date DATE UNIQUE,
    time TIMESTAMP
);
```

Create table with `CREATE TABLE ... AS SELECT` (CTAS):

```
CREATE TABLE t1 AS
    SELECT 42 AS i, 84 AS j;
```

Create a table from a CSV file (automatically detecting column names and types):

```
CREATE TABLE t1 AS
    SELECT *
    FROM read_csv('path/file.csv');
```

We can use the FROM-first syntax to omit `SELECT *`:

```
CREATE TABLE t1 AS
    FROM read_csv('path/file.csv');
```

Copy the schema of `t2` to `t1`:

```
CREATE TABLE t1 AS
    FROM t2
    LIMIT 0;
```

Note that only the column names and types are copied to `t1`, other pieces of information (indexes, constraints, default values, etc.) are not copied.

Temporary Tables

Temporary tables are session scoped, meaning that only the specific connection that created them can access them and once the connection to DuckDB is closed they will be automatically dropped (similar to PostgreSQL, for example).

They can be created using the `CREATE TEMP TABLE` or the `CREATE TEMPORARY TABLE` statement (see diagram below) and are part of the `temp.main` schema. While discouraged, their names can overlap with the names of the regular database tables. In these cases, temporary tables take priority in name resolution and full qualification is required to refer to a regular table e.g., `memory.main.t1`.

Temporary tables reside in memory rather than on disk even when connecting to a persistent DuckDB, but if the `temp_directory` configuration is set, data will be spilled to disk if memory becomes constrained.

Create a temporary table from a CSV file (automatically detecting column names and types):

```
CREATE TEMP TABLE t1 AS
    SELECT *
    FROM read_csv('path/file.csv');
```

Allow temporary tables to off-load excess memory to disk:

```
SET temp_directory = '/path/to/directory/';
```

CREATE OR REPLACE

The `CREATE OR REPLACE` syntax allows a new table to be created or for an existing table to be overwritten by the new table. This is shorthand for dropping the existing table and then creating the new one.

Create a table with two integer columns (i and j) even if t1 already exists:

```
CREATE OR REPLACE TABLE t1 (i INTEGER, j INTEGER);
```

IF NOT EXISTS

The `IF NOT EXISTS` syntax will only proceed with the creation of the table if it does not already exist. If the table already exists, no action will be taken and the existing table will remain in the database.

Create a table with two integer columns (`i` and `j`) only if `t1` does not exist yet:

```
CREATE TABLE IF NOT EXISTS t1 (i INTEGER, j INTEGER);
```

CREATE TABLE ... AS SELECT (CTAS)

DuckDB supports the `CREATE TABLE ... AS SELECT` syntax, also known as "CTAS":

```
CREATE TABLE nums AS
    SELECT i
    FROM range(0, 3) t(i);
```

This syntax can be used in combination with the CSV reader, the shorthand to read directly from CSV files without specifying a function, the FROM-first syntax, and the HTTP(S) support, yielding concise SQL commands such as the following:

```
CREATE TABLE flights AS
    FROM 'https://duckdb.org/data/flights.csv';
```

The CTAS construct also works with the `OR REPLACE` modifier, yielding `CREATE OR REPLACE TABLE ... AS` statements:

```
CREATE OR REPLACE TABLE flights AS
    FROM 'https://duckdb.org/data/flights.csv';
```

Copying the Schema

You can create a copy of the table's schema (column names and types only) as follows:

```
CREATE TABLE t1 AS
    FROM t2
    WITH NO DATA;
```

Or:

```
CREATE TABLE t1 AS
    FROM t2
    LIMIT 0;
```

It is not possible to create tables using CTAS statements with constraints (primary keys, check constraints, etc.).

Check Constraints

A CHECK constraint is an expression that must be satisfied by the values of every row in the table.

```
CREATE TABLE t1 (
    id INTEGER PRIMARY KEY,
    percentage INTEGER CHECK (0 <= percentage AND percentage <= 100)
);
INSERT INTO t1 VALUES (1, 5);
INSERT INTO t1 VALUES (2, -1);

Constraint Error:
CHECK constraint failed: t1

INSERT INTO t1 VALUES (3, 101);

Constraint Error:
CHECK constraint failed: t1

CREATE TABLE t2 (id INTEGER PRIMARY KEY, x INTEGER, y INTEGER CHECK (x < y));
INSERT INTO t2 VALUES (1, 5, 10);
INSERT INTO t2 VALUES (2, 5, 3);

Constraint Error:
CHECK constraint failed: t2
```

CHECK constraints can also be added as part of the `CONSTRAINTS` clause:

```
CREATE TABLE t3 (
    id INTEGER PRIMARY KEY,
    x INTEGER,
    y INTEGER,
    CONSTRAINT x_smaller_than_y CHECK (x < y)
);
INSERT INTO t3 VALUES (1, 5, 10);
INSERT INTO t3 VALUES (2, 5, 3);

Constraint Error:
CHECK constraint failed: t3
```

Foreign Key Constraints

A `FOREIGN KEY` is a column (or set of columns) that references another table's primary key. Foreign keys check referential integrity, i.e., the referred primary key must exist in the other table upon insertion.

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, j VARCHAR);
CREATE TABLE t2 (
    id INTEGER PRIMARY KEY,
    t1_id INTEGER,
    FOREIGN KEY (t1_id) REFERENCES t1 (id)
);
```

Example:

```
INSERT INTO t1 VALUES (1, 'a');
INSERT INTO t2 VALUES (1, 1);
INSERT INTO t2 VALUES (2, 2);
```

```
Constraint Error:
Violates foreign key constraint because key "id: 2" does not exist in the referenced table
```

Foreign keys can be defined on composite primary keys:

```
CREATE TABLE t3 (id INTEGER, j VARCHAR, PRIMARY KEY (id, j));
CREATE TABLE t4 (
    id INTEGER PRIMARY KEY, t3_id INTEGER, t3_j VARCHAR,
    FOREIGN KEY (t3_id, t3_j) REFERENCES t3(id, j)
);
```

Example:

```
INSERT INTO t3 VALUES (1, 'a');
INSERT INTO t4 VALUES (1, 1, 'a');
INSERT INTO t4 VALUES (2, 1, 'b');
```

```
Constraint Error:
Violates foreign key constraint because key "id: 1, j: b" does not exist in the referenced table
```

Foreign keys can also be defined on unique columns:

```
CREATE TABLE t5 (id INTEGER UNIQUE, j VARCHAR);
CREATE TABLE t6 (
    id INTEGER PRIMARY KEY,
    t5_id INTEGER,
    FOREIGN KEY (t5_id) REFERENCES t5(id)
);
```

Limitations

Foreign keys have the following limitations.

Foreign keys with cascading deletes (`FOREIGN KEY ... REFERENCES ... ON DELETE CASCADE`) are not supported.

Inserting a row that references another row inserted in the same statement (or itself) into a table with a self-referencing foreign key is currently not supported and will result in the following error:

```
Constraint Error:
Violates foreign key constraint because key "..." does not exist in the referenced table.
```

Generated Columns

The `[type] [GENERATED ALWAYS] AS (expr) [VIRTUAL|STORED]` syntax will create a generated column. The data in this kind of column is generated from its expression, which can reference other (regular or generated) columns of the table. Since they are produced by calculations, these columns cannot be inserted into directly.

DuckDB can infer the type of the generated column based on the expression's return type. This allows you to leave out the type when declaring a generated column. It is possible to explicitly set a type, but insertions into the referenced columns might fail if the type cannot be cast to the type of the generated column.

Generated columns come in two varieties: `VIRTUAL` and `STORED`. The data of virtual generated columns is not stored on disk, instead it is computed from the expression every time the column is referenced (through a select statement).

The data of stored generated columns is stored on disk and is computed every time the data of their dependencies change (through an `INSERT` / `UPDATE` / `DROP` statement).

Currently, only the `VIRTUAL` kind is supported, and it is also the default option if the last field is left blank.

The simplest syntax for a generated column:

The type is derived from the expression, and the variant defaults to `VIRTUAL`:

```
CREATE TABLE t1 (x FLOAT, two_x AS (2 * x));
```

Fully specifying the same generated column for completeness:

```
CREATE TABLE t1 (x FLOAT, two_x FLOAT GENERATED ALWAYS AS (2 * x) VIRTUAL);
```

Syntax

CREATE VIEW Statement

The `CREATE VIEW` statement defines a new view in the catalog.

Examples

Create a simple view:

```
CREATE VIEW view1 AS SELECT * FROM tbl;
```

Create a view or replace it if a view with that name already exists:

```
CREATE OR REPLACE VIEW view1 AS SELECT 42;
```

Create a view and replace the column names:

```
CREATE VIEW view1 (a) AS SELECT 42;
```

> New The `SECURE VIEW` feature was introduced in DuckDB 2.0.

Create a secure view (hides internals from functions such as `stats()` and `EXPLAIN ANALYZE`):

```
CREATE SECURE VIEW users AS
    SELECT username
    FROM all_users
    WHERE access_level = 'standard';
```

The SQL query behind an existing view can be read using the `duckdb_views()` function like this:

```
SELECT sql
FROM duckdb_views()
WHERE view_name = 'view1';
```

Syntax

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced.

`CREATE SECURE VIEW` defines a view that does not leak information about the underlying tables through statistics or query plans. Regular views can expose data that the view query would filter out (for example via `stats()`). Secure views return `NULL` for those internals and show a `Secure View` node in `EXPLAIN ANALYZE`.

If a schema name is given then the view is created in the specified schema. Otherwise, it is created in the current schema. Temporary views exist in a special schema, so a schema name cannot be given when creating a temporary view. The name of the view must be distinct from the name of any other view or table in the same schema.

CREATE TRIGGER Statement

> New The `CREATE TRIGGER` feature was introduced in DuckDB 2.0.

The `CREATE TRIGGER` statement registers a trigger: a set of SQL statements that run automatically in response to `INSERT`, `UPDATE`, or `DELETE` events on a table.

Examples

Record every statement that modifies a table in an audit table using a statement-level `AFTER` trigger. The `REFERENCING OLD TABLE` and `NEW TABLE` clauses expose the rows before and after the change as transition tables:

```
CREATE TABLE target (id INTEGER, val INTEGER);
CREATE TABLE audit (id INTEGER, old_val INTEGER, new_val INTEGER);

CREATE TRIGGER trg_audit
    AFTER UPDATE ON target
    REFERENCING
        OLD TABLE AS o
        NEW TABLE AS n
    FOR EACH STATEMENT
        INSERT INTO audit
        SELECT n.id, o.val, n.val
        FROM o
        JOIN n ON o.id = n.id;

INSERT INTO target VALUES (1, 10), (2, 20);
UPDATE target SET val = val * 10 WHERE id <= 2;
SELECT * FROM audit;
```

id	old_val	new_val
1	10	100
2	20	200

Fire a trigger once per affected row with a row-level `AFTER` trigger. Row-level triggers refer to the affected row through the `NEW` and `OLD` qualifiers:

```
CREATE TABLE t (id INTEGER);
CREATE TABLE log (id INTEGER);

CREATE TRIGGER trg_log
    AFTER INSERT ON t
    FOR EACH ROW
        INSERT INTO log VALUES (NEW.id);

INSERT INTO t VALUES (1), (2);
SELECT * FROM log;
```

Restrict a trigger to changes of specific columns with `UPDATE OF`:

```
CREATE TRIGGER trg_price
    AFTER UPDATE OF price ON products
    FOR EACH STATEMENT
        INSERT INTO price_changes DEFAULT VALUES;
```

Drop a trigger:

```
DROP TRIGGER trg_audit ON target;
```

Drop a trigger if it exists:

```
DROP TRIGGER IF EXISTS trg_audit ON target;
```

Syntax

The triggered statement is a single SQL statement, typically a data modification statement such as `INSERT`, `UPDATE`, or `DELETE`.

Timing

- `BEFORE` triggers run before the triggering statement is applied.
- `AFTER` triggers run after the triggering statement is applied.

Granularity

- `FOR EACH ROW` fires the trigger once for every row affected by the statement. The triggered statement can refer to the affected row through `NEW` (for `INSERT`) and `OLD` (for `DELETE`). Row-level `UPDATE` triggers are not yet supported; see the limitations.
- `FOR EACH STATEMENT` fires the trigger once per statement, regardless of how many rows are affected. Use `REFERENCING OLD TABLE` and `NEW TABLE` to access the affected rows as transition tables.

Transition Tables

Statement-level triggers can expose the rows involved in the change as transition tables:

- `NEW TABLE` contains the inserted or post-update rows and is available for `INSERT` and `UPDATE`.
- `OLD TABLE` contains the deleted or pre-update rows and is available for `UPDATE` and `DELETE`.

`OR REPLACE` and `IF NOT EXISTS`

`CREATE OR REPLACE TRIGGER` replaces an existing trigger with the same name. `CREATE TRIGGER IF NOT EXISTS` does nothing if a trigger with the same name already exists on the table. Without either clause, creating a trigger with a name that already exists results in an error. The two clauses cannot be combined.

Multiple Triggers

A table can have multiple triggers for the same event, as long as they have the same granularity (see Limitations). They fire in the order in which they were created.

Returning Rows

A data modification statement on a table with `FOR EACH STATEMENT` triggers can still include a `RETURNING` clause. The trigger fires as part of the statement, and the `RETURNING` clause reports the rows affected by the statement itself:

```
CREATE TABLE t (id INTEGER);
CREATE TABLE log (row_count BIGINT);

CREATE TRIGGER trg_log
    AFTER INSERT ON t
    FOR EACH STATEMENT
        INSERT INTO log SELECT count(*) FROM t;

INSERT INTO t VALUES (1), (2) RETURNING id;
```

Inspecting Triggers

The triggers defined in the catalog can be listed with the `duckdb_triggers()` function:

```
SELECT trigger_name, table_name, action_timing, event_manipulation, for_each
FROM duckdb_triggers();
```

Dropping Triggers

A trigger is dropped with `DROP TRIGGER`. The table the trigger is attached to must be specified:

For example:

```
DROP TRIGGER trg_audit ON target;
```

Dropping a trigger that does not exist results in an error. Use `IF EXISTS` to skip the statement if the trigger does not exist:

```
DROP TRIGGER IF EXISTS trg_audit ON target;
```

The table must exist even when `IF EXISTS` is specified. The table name can be qualified with a schema name, e.g., `ON my_schema.target`.

Dropping a table also drops all triggers defined on it.

Limitations

DuckDB 2.0 has the following limitations:

- `BEFORE ... FOR EACH ROW` triggers are not yet supported.
- `UPDATE ... FOR EACH ROW` triggers are not yet supported.
- `RETURNING` is not yet supported on tables with `FOR EACH ROW` triggers.
- `INSTEAD OF` triggers and triggers on views are not supported; a trigger requires a base table.
- Temporary triggers are not supported.
- `MERGE INTO` is not supported on tables with triggers. `INSERT ... ON CONFLICT`, `INSERT OR REPLACE` and `INSERT OR IGNORE` are executed as a merge and therefore fail on tables with `UPDATE` or `DELETE` triggers.
- `ON CONFLICT` is not yet supported on tables with `FOR EACH ROW` triggers.
- `FOR EACH ROW` and `FOR EACH STATEMENT` triggers cannot be mixed for the same event on the same table.

CREATE TYPE Statement

The `CREATE TYPE` statement defines a new type in the catalog.

Examples

Create a simple ENUM type:

```
CREATE TYPE mood AS ENUM ('happy', 'sad', 'curious');
```

Create a simple STRUCT type:

```
CREATE TYPE many_things AS STRUCT(k INTEGER, l VARCHAR);
```

Create a simple UNION type:

```
CREATE TYPE one_thing AS UNION(number INTEGER, string VARCHAR);
```

Create a type alias:

```
CREATE TYPE x_index AS INTEGER;
```

Replace an existing type:

```
CREATE OR REPLACE TYPE mood AS ENUM ('cheerful', 'gloomy');
```

Syntax

The `CREATE TYPE` clause defines a new data type available to this DuckDB instance. Use the `OR REPLACE` modifier to redefine a type that already exists. These new types can then be inspected in the `duckdb_types` table.

Limitations

- Extending types to support custom operators (such as the PostgreSQL && operator) is not possible via plain SQL. Instead, it requires adding additional C++ code. To do this, create an extension.

DELETE Statement

The `DELETE` statement removes rows from the table identified by the table-name. If the `WHERE` clause is not present, all records in the table are deleted. If a `WHERE` clause is supplied, then only those rows for which the `WHERE` clause results in true are deleted. Rows for which the expression is false or `NULL` are retained.

Examples

Remove the rows matching the condition `i = 2` from the database:

```
DELETE FROM tbl WHERE i = 2;
```

Delete all rows in the table `tbl`:

```
DELETE FROM tbl;
```

`USING` Clause

The `USING` clause allows deleting based on the content of other tables or subqueries.

`RETURNING` Clause

The `RETURNING` clause allows returning the deleted values. It uses the same syntax as the `SELECT` clause except the `DISTINCT` modifier is not supported.

```
CREATE TABLE employees (name VARCHAR, age INTEGER);
INSERT INTO employees VALUES ('Kat', 32);
DELETE FROM employees RETURNING name, 2025 - age AS approx_birthyear;
```

name	approx_birthyear
Kat	1993

Syntax

The TRUNCATE Statement

The `TRUNCATE` statement removes all rows from a table, acting as an alias for `DELETE FROM` without a `WHERE` clause:

```
TRUNCATE tbl;
```

Limitations on Reclaiming Memory and Disk Space

Running `DELETE` does not mean space is reclaimed. In general, rows are only marked as deleted. DuckDB reclaims space upon performing a `CHECKPOINT`. `VACUUM` currently does not reclaim space.

DESCRIBE Statement

The `DESCRIBE` statement shows the schema of a table, view or query.

Usage

```
DESCRIBE tbl;
```

To describe a query, prepend `DESCRIBE` to a query.

```
DESCRIBE SELECT * FROM tbl;
```

Alias

The `SHOW` statement can also describe tables and queries, but this usage is deprecated as of DuckDB 2.0 and prints a warning.

See Also

For more examples, see the guide on `DESCRIBE`.

DROP Statement

The `DROP` statement removes a catalog entry added previously with the `CREATE` command.

Examples

Delete the table with the name `tbl`:

```
DROP TABLE tbl;
```

Drop the view with the name `view1`; do not throw an error if the view does not exist:

```
DROP VIEW IF EXISTS view1;
```

Drop function `fn`:

```
DROP FUNCTION fn;
```

Drop index `idx`:

```
DROP INDEX idx;
```

Drop schema `sch`:

```
DROP SCHEMA sch;
```

Drop sequence `seq`:

```
DROP SEQUENCE seq;
```

Drop macro `mcr`:

```
DROP MACRO mcr;
```

Drop macro table `mt`:

```
DROP MACRO TABLE mt; -- the `TABLE` is optional since v1.4.0
```

Drop type `typ`:

```
DROP TYPE typ;
```

Syntax

Dependencies of Dropped Objects

DuckDB performs limited dependency tracking for some object types. By default or if the `RESTRICT` clause is provided, the entry will not be dropped if there are any other objects that depend on it. If the `CASCADE` clause is provided then all the objects that are dependent on the object will be dropped as well.

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1 (i INTEGER);
DROP SCHEMA myschema;
```

```
Dependency Error:
Cannot drop entry "myschema" because there are entries that depend on it.
table "t1" depends on schema "myschema".
Use DROP...CASCADE to drop all dependents.
```

The `CASCADE` modifier drops both `myschema` and `myschema.t1`:

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1 (i INTEGER);
DROP SCHEMA myschema CASCADE;
```

The following dependencies are tracked and thus will raise an error if the user tries to drop the depending object without the `CASCADE` modifier.

Depending object type	Dependent object type
SCHEMA	FUNCTION
SCHEMA	MACRO TABLE
SCHEMA	MACRO
SCHEMA	SCHEMA
SCHEMA	SEQUENCE
SCHEMA	TABLE
SCHEMA	TYPE
SCHEMA	VIEW

Limitations

Dependencies on Views

Currently, dependencies are not tracked for views. For example, if a view is created that references a table and the table is dropped, then the view will be in an invalid state:

```
CREATE TABLE tbl (i INTEGER);
CREATE VIEW view1 AS
    SELECT i FROM tbl;
DROP TABLE tbl RESTRICT;
SELECT * FROM view1;
```

This returns the following error message:

```
Catalog Error:
Table with name tbl does not exist!
```

Limitations on Reclaiming Disk Space

Running `DROP TABLE` should free the memory used by the table, but not always disk space. Even if disk space does not decrease, the free blocks will be marked as `free`. For example, if we have a 2 GB file and we drop a 1 GB table, the file might still be 2 GB, but it should have 1 GB of free blocks in it. To check this, use the following PRAGMA and check the number of `free_blocks` in the output:

```
PRAGMA database_size;
```

For instructions on reclaiming space after dropping a table, refer to the "Reclaiming space" page.

EXPORT and IMPORT DATABASE Statements

The `EXPORT DATABASE` command allows you to export the contents of the database to a specific directory. The `IMPORT DATABASE` command allows you to then read the contents again.

Examples

Export the database to the target directory `target_directory` as CSV files:

```
EXPORT DATABASE 'target_directory';
```

Export to directory `target_directory`, using the given options for the CSV serialization:

```
EXPORT DATABASE 'target_directory' (FORMAT csv, DELIMITER '|');
```

Export to directory `target_directory`, tables serialized as Parquet:

```
EXPORT DATABASE 'target_directory' (FORMAT parquet);
```

Export to directory `target_directory`, tables serialized as Parquet, compressed with Zstd, with a `ROW_GROUP_SIZE` of 100,000:

```
EXPORT DATABASE 'target_directory' (
    FORMAT parquet,
    COMPRESSION zstd,
    ROW_GROUP_SIZE 100_000
);
```

Reload the database again:

```
IMPORT DATABASE 'source_directory';
```

Alternatively, use a PRAGMA:

```
PRAGMA import_database('source_directory');
```

For details regarding the writing of Parquet files, see the Parquet Files page in the Data Import section and the COPY Statement page.

EXPORT DATABASE

The `EXPORT DATABASE` command exports the full contents of the database – including schema information, tables, views and sequences – to a specific directory that can then be loaded again. The created directory will be structured as follows:

```
target_directory/schema.sql
target_directory/load.sql
target_directory/t_1.csv
...
target_directory/t_n.csv
```

The `schema.sql` file contains the schema statements that are found in the database. It contains any `CREATE SCHEMA`, `CREATE TABLE`, `CREATE VIEW` and `CREATE SEQUENCE` commands that are necessary to re-construct the database.

> **Warning.** In DuckDB 2.0.0, triggers are not included in the export, and `COPY FROM DATABASE` does not copy them either. Recreate triggers manually after importing or copying a database.

The `load.sql` file contains a set of `COPY` statements that can be used to read the data from the CSV files again. The file contains a single `COPY` statement for every table found in the schema.

Syntax

IMPORT DATABASE

The database can be reloaded by using the `IMPORT DATABASE` command again, or manually by running `schema.sql` followed by `load.sql` to re-load the data.

> `IMPORT DATABASE` expects to import into a fresh database. `schema.sql` uses plain `CREATE` statements (without `OR REPLACE` or `IF NOT EXISTS`), so importing into a database that already contains any of the exported objects – for example a user-defined type – fails with a "already exists" error. Import into an empty database.

Syntax

INSERT Statement

The `INSERT` statement inserts new data into a table.

Examples

Insert the values 1, 2, 3 into `tbl`:

```
INSERT INTO tbl
    VALUES (1), (2), (3);
```

Insert the result of a query into a table:

```
INSERT INTO tbl
    SELECT * FROM other_tbl;
```

Insert values into the `i` column, inserting the default value into other columns:

```
INSERT INTO tbl (i)
    VALUES (1), (2), (3);
```

Explicitly insert the default value into a column:

```
INSERT INTO tbl (i)
    VALUES (1), (DEFAULT), (3);
```

Assuming `tbl` has a primary key/unique constraint, do nothing on conflict:

```
INSERT OR IGNORE INTO tbl (i)
    VALUES (1);
```

Or update the table with the new values instead:

```
INSERT OR REPLACE INTO tbl (i)
    VALUES (1);
```

Syntax

`INSERT INTO` inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

Insert Column Order

It's possible to provide an optional insert column order, this can either be `BY POSITION` (the default) or `BY NAME`. Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or `NULL` if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

`INSERT INTO ... [BY POSITION]`

The order that values are inserted into the columns of the table is determined by the order that the columns were declared in. That is, the values supplied by the `VALUES` clause or query are associated with the column list left-to-right. This is the default option, that can be explicitly specified using the `BY POSITION` option. For example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl
    VALUES (5, 42);
```

Specifying `BY POSITION` is optional and is equivalent to the default behavior:

```
INSERT INTO tbl
    BY POSITION
    VALUES (5, 42);
```

To use a different order, column names can be provided as part of the target, for example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl (b, a)
    VALUES (5, 42);
```

Adding `BY POSITION` results in the same behavior:

```
INSERT INTO tbl
    BY POSITION (b, a)
    VALUES (5, 42);
```

This will insert 5 into b and 42 into a.

`INSERT INTO ... BY NAME`

Using the `BY NAME` modifier, the names of the column list of the `SELECT` statement are matched against the column names of the table to determine the order that values should be inserted into the table. This allows inserting even in cases when the order of the columns in the table differs from the order of the values in the `SELECT` statement or certain columns are missing.

For example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl BY NAME (SELECT 42 AS b, 32 AS a);
INSERT INTO tbl BY NAME (SELECT 22 AS b);
SELECT * FROM tbl;
```

a	b
32	42
NULL	22

It's important to note that when using `INSERT INTO ... BY NAME`, the column names specified in the `SELECT` statement must match the column names in the table. If a column name is misspelled or does not exist in the table, an error will occur. Columns that are missing from the `SELECT` statement will be filled with the default value.

`ON CONFLICT` Clause

An `ON CONFLICT` clause can be used to perform a certain action on conflicts that arise from `UNIQUE` or `PRIMARY KEY` constraints. An example for such a conflict is shown in the following example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84);
```

This raises an error:

```
Constraint Error:
Duplicate key "i: 1" violates primary key constraint.
```

The table will contain the row that was first inserted:

```
SELECT * FROM tbl;
```

i	j
1	42

These error messages can be avoided by explicitly handling conflicts. DuckDB supports two such clauses: `ON CONFLICT DO NOTHING` and `ON CONFLICT DO UPDATE SET ...`.

`DO NOTHING` Clause

The `DO NOTHING` clause causes the error(s) to be ignored, and the values are not inserted or updated. For example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO NOTHING;
```

These statements finish successfully and leave the table with the row `<i: 1, j: 42>`.

`INSERT OR IGNORE INTO` The `INSERT OR IGNORE INTO ...` statement is a shorter syntax alternative to `INSERT INTO ... ON CONFLICT DO NOTHING`. For example, the following statements are equivalent:

```
INSERT OR IGNORE INTO tbl
    VALUES (1, 84);
INSERT INTO tbl
    VALUES (1, 84) ON CONFLICT DO NOTHING;
```

`DO UPDATE` Clause (Upsert)

The `DO UPDATE` clause causes the `INSERT` to turn into an `UPDATE` on the conflicting row(s) instead. The `SET` expressions that follow determine how these rows are updated. The expressions can use the special virtual table `EXCLUDED`, which contains the conflicting values for the row. Optionally you can provide an additional `WHERE` clause that can exclude certain rows from the update. The conflicts that don't meet this condition are ignored instead.

Because we need a way to refer to both the **to-be-inserted** tuple and the **existing** tuple, we introduce the special `EXCLUDED` qualifier. When the `EXCLUDED` qualifier is provided, the reference refers to the **to-be-inserted** tuple, otherwise, it refers to the **existing** tuple. This special qualifier can be used within the `WHERE` clauses and `SET` expressions of the `ON CONFLICT` clause.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl VALUES (1, 42);
INSERT INTO tbl VALUES (1, 52), (1, 62) ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
```

> **Warning.** When a single `INSERT` statement contains multiple to-be-inserted rows that share the same conflict target, DuckDB applies only one update per conflicting key, and the row that wins is not guaranteed to be the last one in insertion order. In the example above, `j` ends up as `52` (the first of the two conflicting rows) rather than `62`. If you need last-writer-wins semantics, deduplicate the input before inserting, for example by keeping only the last row per key with a window function.

Examples An example using `DO UPDATE` is the following:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	84

Rearranging columns and using `BY NAME` is also possible:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl (j, i)
    VALUES (168, 1)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl
    BY NAME (SELECT 1 AS i, 336 AS j)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	336

`INSERT OR REPLACE INTO` The `INSERT OR REPLACE INTO ...` statement is a shorter syntax alternative to `INSERT INTO ... DO UPDATE SET c1 = EXCLUDED.c1, c2 = EXCLUDED.c2, ...`. That is, it updates every column of the **existing** row to the new values of the **to-be-inserted** row. For example, given the following input table:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
```

These statements are equivalent:

```
INSERT OR REPLACE INTO tbl
    VALUES (1, 84);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl (j, i)
    VALUES (84, 1)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl BY NAME
    (SELECT 84 AS j, 1 AS i)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
```

Composite Primary Key When multiple columns need to be part of the uniqueness constraint, use a single `PRIMARY KEY` clause including all relevant columns:

```
CREATE TABLE t1 (id1 INTEGER, id2 INTEGER, val1 DOUBLE, PRIMARY KEY (id1, id2));
INSERT OR REPLACE INTO t1
    VALUES (1, 2, 3);
INSERT OR REPLACE INTO t1
    VALUES (1, 2, 4);
```

Defining a Conflict Target

A conflict target may be provided as `ON CONFLICT (conflict_target)`. This is a group of columns that an index or uniqueness/key constraint is defined on. If the conflict target is omitted, the `PRIMARY KEY` constraint(s) on the table are targeted.

Specifying a conflict target is optional unless using a `DO UPDATE` and there are multiple unique/primary key constraints on the table.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER UNIQUE, k INTEGER);
INSERT INTO tbl
    VALUES (1, 20, 300);
SELECT * FROM tbl;
```

i	j	k
1	20	300

```
INSERT INTO tbl
    VALUES (1, 40, 700)
    ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k;
```

i	j	k
1	20	1400

```
INSERT INTO tbl
    VALUES (1, 20, 900)
    ON CONFLICT (j) DO UPDATE SET k = 5 * EXCLUDED.k;
```

i	j	k
1	20	4500

The `DO UPDATE SET` action may be further restricted with a `WHERE` clause, which is checked for each conflicting row. Conflicting rows that do not meet the condition are left unchanged. This `WHERE` clause is part of the `DO UPDATE SET` action and must appear after it, as shown below.

```
INSERT INTO tbl
    VALUES (1, 40, 700)
    ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k WHERE k < 100;
```

> **Warning.** DuckDB does not support PostgreSQL's partial-index predicate on the conflict target, i.e., a `WHERE` clause placed directly after the conflict target as in `ON CONFLICT (i) WHERE ... DO UPDATE SET ...`. Attempting this returns a `Binder Error`.

`RETURNING` Clause

The `RETURNING` clause may be used to return the contents of the rows that were inserted. This can be useful if some columns are calculated upon insert. For example, if the table contains an automatically incrementing primary key, then the `RETURNING` clause will include the automatically created primary key. This is also useful in the case of generated columns.

Some or all columns can be explicitly chosen to be returned and they may optionally be renamed using aliases. Arbitrary non-aggregating expressions may also be returned instead of simply returning a column. All columns can be returned using the `*` expression, and columns or expressions can be returned in addition to all columns returned by the `*`.

For example:

```
CREATE TABLE t1 (i INTEGER);
INSERT INTO t1
    SELECT 42
    RETURNING *;
```

i
42

A more complex example that includes an expression in the `RETURNING` clause:

```
CREATE TABLE t2 (i INTEGER, j INTEGER);
INSERT INTO t2
    SELECT 2 AS i, 3 AS j
    RETURNING *, i * j AS i_times_j;
```

i	j	i_times_j
2	3	6

The next example shows a situation where the RETURNING clause is more helpful. First, a table is created with a primary key column. Then a sequence is created to allow for that primary key to be incremented as new rows are inserted. When we insert into the table, we do not already know the values generated by the sequence, so it is valuable to return them. For additional information, see the CREATE SEQUENCE page.

```
CREATE TABLE t3 (i INTEGER PRIMARY KEY, j INTEGER);
CREATE SEQUENCE 't3_key';
INSERT INTO t3
    SELECT nextval('t3_key') AS i, 42 AS j
    UNION ALL
    SELECT nextval('t3_key') AS i, 43 AS j
    RETURNING *;
```

i	j
1	42
2	43

LOAD / INSTALL Statements

INSTALL

The INSTALL statement downloads an extension so it can be loaded into a DuckDB session.

Examples

Install the httpfs extension:

```
INSTALL httpfs;
```

Install the h3 community extension:

```
INSTALL h3 FROM community;
```

Syntax

LOAD

The LOAD statement loads an installed DuckDB extension into the current session.

LOAD does not install extensions: if the extension is not installed yet, LOAD fails with an error asking you to run INSTALL first. Use INSTALL AND LOAD to do both in one step. (When a query uses a function or feature of a known core extension, autoloading installs and loads it automatically.) Loading an extension also installs and loads any extensions it depends on. For example, loading iceberg also installs the avro extension that it depends on.

Examples

Load the `httpfs` extension:

```
LOAD httpfs;
```

Load the `spatial` extension:

```
LOAD spatial;
```

Syntax

`INSTALL AND LOAD`

> New The `INSTALL AND LOAD` statement was introduced in DuckDB 2.0.

The `INSTALL AND LOAD` statement installs an extension and loads it into the current session in a single step. It is equivalent to running `INSTALL` followed by `LOAD`. If the extension is already installed, the installation step is skipped; if it is already loaded, the loading step is skipped.

`INSTALL AND LOAD` supports the same clauses as `INSTALL`: it can be prefixed with `FORCE` and can be followed by a `FROM` clause to select the repository and a `VERSION` clause to select the version.

Examples

Install and load the `spatial` extension:

```
INSTALL AND LOAD spatial;
```

Install and load the h3 community extension:

```
INSTALL AND LOAD h3 FROM community;
```

Reinstall the `httpfs` extension from the `core_nightly` repository and load it:

```
FORCE INSTALL AND LOAD httpfs FROM core_nightly;
```

MERGE INTO Statement

The `MERGE INTO` statement is an alternative to `INSERT INTO ... ON CONFLICT` that doesn't need a primary key since it allows for a custom match condition. This is a very useful alternative for upserting use cases (`INSERT` + `UPDATE`) when the destination table does not have a primary key constraint.

Examples

First, let's create a simple table.

```
CREATE TABLE people (id INTEGER, name VARCHAR, salary FLOAT);
INSERT INTO people VALUES (1, 'John', 92_000.0), (2, 'Anna', 100_000.0);
```

The simplest upsert would be to use a whole row in the `USING` clause. This way, if there is a match, the row can be updated to the new row without further instuctions (`WHEN MATCHED THEN UPDATE`), and when there is no match, the row can be trivially inserted into the table (`WHEN NOT MATCHED THEN INSERT`).

```
MERGE INTO people
    USING (
        SELECT
            unnest([3, 1]) AS id,
            unnest(['Sarah', 'John']) AS name,
            unnest([95_000.0, 105_000.0]) AS salary
    ) AS upserts
    ON (upserts.id = people.id)
    WHEN MATCHED THEN UPDATE
    WHEN NOT MATCHED THEN INSERT;

FROM people
ORDER BY id;
```

id	name	salary
1	John	105000.0
2	Anna	100000.0
3	Sarah	95000.0

In the previous example we are updating the whole row if `id` matches. However, it is also a common pattern to receive a *change set* with some keys and the changed value. This is a good use for `SET`. If the match condition uses a column that has the same name in the source and destination, the keyword `USING` can be used in the match condition.

```
MERGE INTO people
    USING (
        SELECT
            1 AS id,
            98_000.0 AS salary
    ) AS salary_updates
    USING (id)
    WHEN MATCHED THEN UPDATE SET salary = salary_updates.salary;

FROM people
ORDER BY id;
```

id	name	salary
1	John	98000.0
2	Anna	100000.0
3	Sarah	95000.0

Another common pattern is to receive a *delete set* of rows, which may only contain ids of rows to be deleted.

```
MERGE INTO people
    USING (
        SELECT
            1 AS id,
    ) AS deletes
    USING (id)
    WHEN MATCHED THEN DELETE;

FROM people
ORDER BY id;
```

id	name	salary
2	Anna	100000.0
3	Sarah	95000.0

`MERGE INTO` also supports more complex conditions, for example, for a given *delete set* we can decide to only remove rows that contain a `salary` bigger or equal than a certain amount.

```
MERGE INTO people
    USING (
        SELECT
            unnest([3, 2]) AS id,
    ) AS deletes
    USING (id)
    WHEN MATCHED AND people.salary >= 100_000.0 THEN DELETE;

FROM people
ORDER BY id;
```

id	name	salary
3	Sarah	95000.0

If needed, DuckDB also supports multiple `UPDATE` and `DELETE` conditions. The `RETURNING` clause can be used to indicate which rows were affected by the `MERGE` statement.

```
-- Let's get John back in!
INSERT INTO people VALUES (1, 'John', 105_000.0);

MERGE INTO people
    USING (
        SELECT
            unnest([3, 1]) AS id,
            unnest([89_000.0, 70_000.0]) AS salary
    ) AS upserts
    USING (id)
    WHEN MATCHED AND people.salary < 100_000.0 THEN UPDATE SET salary = upserts.salary
    -- Second update or delete condition
    WHEN MATCHED AND people.salary > 100_000.0 THEN DELETE
    WHEN NOT MATCHED THEN INSERT BY NAME
    RETURNING merge_action, *;
```

merge_action	id	name	salary
UPDATE	3	Sarah	89000.0
DELETE	1	John	105000.0

In some cases, you may want to perform a different action specifically if the source doesn't meet a condition. For example, if we expect that data that is not present on the source shouldn't be present in the target:

```
CREATE TABLE target AS
    SELECT unnest([1,2]) AS id;

MERGE INTO target
    USING (SELECT 1 AS id) source
    USING (id)
```

```
    WHEN MATCHED THEN UPDATE
    WHEN NOT MATCHED BY SOURCE THEN DELETE
    RETURNING merge_action, *;
```

merge_action	id
UPDATE	1
DELETE	2

There is also the possibility of specifying `WHEN NOT MATCHED BY TARGET`. However, the behavior is, as you may expect, the same as `WHEN NOT MATCHED` since by default when specifying conditions, we look at the target.

Multiple Matching Source Rows

A `MERGE INTO` statement may not affect the same target row more than once. If a target row matches multiple source rows and a `WHEN MATCHED` clause applies (e.g., `UPDATE` or `DELETE`), the statement throws an error:

```
CREATE TABLE target (id INTEGER, v INTEGER);
INSERT INTO target VALUES (1, 0);
CREATE TABLE source (id INTEGER, v INTEGER);
INSERT INTO source VALUES (1, 10), (1, 20);

MERGE INTO target
    USING source
    ON target.id = source.id
    WHEN MATCHED THEN UPDATE SET v = source.v;
```

```
Invalid Input Error:
MERGE INTO command cannot affect the same target row more than once. A target row matched more than one
source row; ensure the source rows are deduplicated or the ON condition is unique.
```

To avoid this, deduplicate the source rows or make the ON condition unique. Note that this differs from `UPDATE ... FROM`, which uses an arbitrary matching source row.

Syntax

PIVOT Statement

The `PIVOT` statement allows distinct values within a column to be separated into their own columns. The values within those new columns are calculated using an aggregate function on the subset of rows that match each distinct value.

DuckDB implements both the SQL Standard `PIVOT` syntax and a simplified `PIVOT` syntax that automatically detects the columns to create while pivoting. `PIVOT_WIDER` may also be used in place of the `PIVOT` keyword.

For details on how the `PIVOT` statement is implemented, see the Pivot Internals site.

> The `UNPIVOT` statement is the inverse of the `PIVOT` statement.

Simplified `PIVOT` Syntax

The full syntax diagram is below, but the simplified `PIVOT` syntax can be summarized using spreadsheet pivot table naming conventions as:

```
PIVOT <dataset>
ON <columns>
USING <values>
GROUP BY <rows>
ORDER BY <columns_with_order_directions>
LIMIT <number_of_rows>;
```

The ON, USING, and GROUP BY clauses are each optional, but they may not all be omitted.

Example Data

All examples use the dataset produced by the queries below:

```
CREATE TABLE cities (
    country VARCHAR, name VARCHAR, year INTEGER, population INTEGER
);
INSERT INTO cities VALUES
    ('NL', 'Amsterdam', 2000, 1005),
    ('NL', 'Amsterdam', 2010, 1065),
    ('NL', 'Amsterdam', 2020, 1158),
    ('US', 'Seattle', 2000, 564),
    ('US', 'Seattle', 2010, 608),
    ('US', 'Seattle', 2020, 738),
    ('US', 'New York City', 2000, 8015),
    ('US', 'New York City', 2010, 8175),
    ('US', 'New York City', 2020, 8772);

SELECT *
FROM cities;
```

country	name	year	population
NL	Amsterdam	2000	1005
NL	Amsterdam	2010	1065
NL	Amsterdam	2020	1158
US	Seattle	2000	564
US	Seattle	2010	608
US	Seattle	2020	738
US	New York City	2000	8015
US	New York City	2010	8175
US	New York City	2020	8772

`PIVOT` `ON` and `USING`

Use the PIVOT statement below to create a separate column for each year and calculate the total population in each. The ON clause specifies which column(s) to split into separate columns. It is equivalent to the columns parameter in a spreadsheet pivot table.

The USING clause determines how to aggregate the values that are split into separate columns. This is equivalent to the values parameter in a spreadsheet pivot table. If the USING clause is not included, it defaults to `count(*)`.

```
PIVOT cities
ON year
USING sum(population);
```

country	name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

In the above example, the `sum` aggregate is always operating on a single value. If we only want to change the orientation of how the data is displayed without aggregating, use the `first` aggregate function. In this example, we are pivoting numeric values, but the `first` function works very well for pivoting out a text column. (This is something that is difficult to do in a spreadsheet pivot table, but easy in DuckDB!)

This query produces a result that is identical to the one above:

```
PIVOT cities
ON year
USING first(population);
```

> **Note.** The SQL syntax permits `FILTER` clauses with aggregate functions in the `USING` clause. In DuckDB, the `PIVOT` statement currently does not support these and they are silently ignored.

`PIVOT ON`, `USING`, and `GROUP BY`

By default, the `PIVOT` statement retains all columns not specified in the `ON` or `USING` clauses. To include only certain columns and further aggregate, specify columns in the `GROUP BY` clause. This is equivalent to the rows parameter of a spreadsheet pivot table.

In the below example, the `name` column is no longer included in the output, and the data is aggregated up to the `country` level.

```
PIVOT cities
ON year
USING sum(population)
GROUP BY country;
```

country	2000	2010	2020
NL	1005	1065	1158
US	8579	8783	9510

`IN` Filter for `ON` Clause

To only create a separate column for specific values within a column in the `ON` clause, use an optional `IN` expression. Let's say for example that we wanted to forget about the year 2020 for no particular reason...

```
PIVOT cities
ON year IN (2000, 2010)
USING sum(population)
GROUP BY country;
```

country	2000	2010
NL	1005	1065
US	8579	8783

Multiple Expressions per Clause

Multiple columns can be specified in the ON and GROUP BY clauses, and multiple aggregate expressions can be included in the USING clause.

Multiple ON Columns and ON Expressions Multiple columns can be pivoted out into their own columns. DuckDB will find the distinct values in each ON clause column and create one new column for all combinations of those values (a Cartesian product).

In the below example, all combinations of unique countries and unique cities receive their own column. Some combinations may not be present in the underlying data, so those columns are populated with NULL values.

```
PIVOT cities
ON country, name
USING sum(population);
```

year	NL_Amsterdam	NL_New York City	NL_Seattle	US_Amsterdam	US_New York City	US_Seattle
2000	1005	NULL	NULL	NULL	8015	564
2010	1065	NULL	NULL	NULL	8175	608
2020	1158	NULL	NULL	NULL	8772	738

To pivot only the combinations of values that are present in the underlying data, use an expression in the ON clause. Multiple expressions and/or columns may be provided.

Here, country and name are concatenated together and the resulting concatenations each receive their own column. Any arbitrary non-aggregating expression may be used. In this case, concatenating with an underscore is used to imitate the naming convention the PIVOT clause uses when multiple ON columns are provided (like in the prior example).

```
PIVOT cities
ON country || '_' || name
USING sum(population);
```

year	NL_Amsterdam	US_New York City	US_Seattle
2000	1005	8015	564
2010	1065	8175	608
2020	1158	8772	738

Multiple USING Expressions An alias may also be included for each expression in the USING clause. It will be appended to the generated column names after an underscore (_). This makes the column naming convention much cleaner when multiple expressions are included in the USING clause.

In this example, both the sum and max of the population column are calculated for each year and are split into separate columns.

```
PIVOT cities
ON year
USING sum(population) AS total, max(population) AS max
GROUP BY country;
```

country	2000_total	2000_max	2010_total	2010_max	2020_total	2020_max
US	8579	8015	8783	8175	9510	8772
NL	1005	1005	1065	1065	1158	1158

country	2000_total	2000_max	2010_total	2010_max	2020_total	2020_max

Multiple GROUP BY Columns Multiple GROUP BY columns may also be provided. Note that column names must be used rather than column positions (1, 2, etc.), and that expressions are not supported in the GROUP BY clause.

```
PIVOT cities
ON year
USING sum(population)
GROUP BY country, name;
```

country	name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

Using PIVOT within a SELECT Statement

The PIVOT statement may be included within a SELECT statement as a CTE (a Common Table Expression, or WITH clause), or a subquery. This allows for a PIVOT to be used alongside other SQL logic, as well as for multiple PIVOTs to be used in one query.

No SELECT is needed within the CTE, the PIVOT keyword can be thought of as taking its place.

```
WITH pivot_alias AS (
    PIVOT cities
    ON year
    USING sum(population)
    GROUP BY country
)
SELECT * FROM pivot_alias;
```

A PIVOT may be used in a subquery and must be wrapped in parentheses. Note that this behavior is different than the SQL Standard Pivot, as illustrated in subsequent examples.

```
SELECT *
FROM (
    PIVOT cities
    ON year
    USING sum(population)
    GROUP BY country
) pivot_alias;
```

Multiple PIVOT Statements

Each PIVOT can be treated as if it were a SELECT node, so they can be joined together or manipulated in other ways.

For example, if two PIVOT statements share the same GROUP BY expression, they can be joined together using the columns in the GROUP BY clause into a wider pivot.

```
SELECT *
FROM (PIVOT cities ON year USING sum(population) GROUP BY country) year_pivot
JOIN (PIVOT cities ON name USING sum(population) GROUP BY country) name_pivot
USING (country);
```

country	2000	2010	2020	Amsterdam	New York City	Seattle
NL	1005	1065	1158	3228	NULL	NULL
US	8579	8783	9510	NULL	24962	1910

Simplified `PIVOT` Full Syntax Diagram

Below is the full syntax diagram of the `PIVOT` statement.

SQL Standard `PIVOT` Syntax

The full syntax diagram is below, but the SQL Standard `PIVOT` syntax can be summarized as:

```
SELECT *
FROM <dataset>
PIVOT (
    <values>
    FOR
        <column_1> IN (<in_list>)
        <column_2> IN (<in_list>)
        ...
    GROUP BY <rows>
);
```

Unlike the simplified syntax, the `IN` clause must be specified for each column to be pivoted. If you are interested in dynamic pivoting, the simplified syntax is recommended.

Note that no commas separate the expressions in the `FOR` clause, but that `value` and `GROUP BY` expressions must be comma-separated!

Examples

This example uses a single value expression, a single column expression, and a single row expression:

```
SELECT *
FROM cities
PIVOT (
    sum(population)
    FOR
        year IN (2000, 2010, 2020)
    GROUP BY country
);
```

country	2000	2010	2020
NL	1005	1065	1158
US	8579	8783	9510

This example is somewhat contrived, but serves as an example of using multiple value expressions and multiple columns in the `FOR` clause.

```
SELECT *
FROM cities
PIVOT (
    sum(population) AS total,
```

```
    count(population) AS count
    FOR
        year IN (2000, 2010)
        country IN ('NL', 'US')
);
```

name	2000_NL_total	2000_NL_count	2000_US_total	2000_US_count	2010_NL_total	2010_NL_count	2010_US_total	2010_US_count
Amsterdam	1005	1	NULL	0	1065	1	NULL	0
Seattle	NULL	0	564	1	NULL	0	608	1
New York City	NULL	0	8015	1	NULL	0	8175	1

SQL Standard PIVOT Full Syntax Diagram

Below is the full syntax diagram of the SQL Standard version of the PIVOT statement.

Expressions in the USING Clause

The USING clause also accepts expressions over aggregate functions. For example, the following query gets the population as the number of people instead of thousands of people (i.e., instead of 564, get 564000):

```
PIVOT cities
ON year
USING sum(population) * 1000;
```

Alternatively, perform the PIVOT with the aggregation only, then use the COLUMNS expression:

```
SELECT country, name, 1000 * COLUMNS(* EXCLUDE (country, name))
FROM (
    PIVOT cities
    ON year
    USING sum(population)
);
```

Profiling Queries

DuckDB supports profiling queries via the EXPLAIN and EXPLAIN ANALYZE statements.

EXPLAIN

To see the query plan of a query without executing it, run:

```
EXPLAIN <query>;
```

The output of EXPLAIN contains the estimated cardinalities for each operator.

EXPLAIN (SQL)

> New As of DuckDB 2.0, EXPLAIN (SQL) renders the optimized logical plan back as an executable DuckDB SQL string instead of as a plan tree:

```
EXPLAIN (SQL) SELECT 1 + 2 AS answer;
```

explain_key	explain_value
sql	SELECT CAST(3 AS INTEGER) AS answer

Because the output is the *optimized* plan, it can differ substantially from the input query (for example through join reordering, constant folding, or subquery introduction). The generated query preserves the result column names, column order, and logical types, but the exact formatting and any generated aliases are not a stable interface. `EXPLAIN (SQL)` does not execute the query, supports only `SELECT`, `VALUES`, `WITH`, `INSERT`, `UPDATE`, `DELETE` and `MERGE INTO` statements, and cannot be combined with `ANALYZE` or `FORMAT`.

EXPLAIN ANALYZE

To profile a query, run:

```
EXPLAIN ANALYZE <query>;
```

The `EXPLAIN ANALYZE` statement runs the query, and shows the actual cardinalities for each operator, as well as the cumulative wall-clock time spent in each operator.

PREPARE, EXECUTE, and DEALLOCATE Statements

The `PREPARE` statement creates a prepared statement for later execution. The `EXECUTE` statement runs it with optional arguments. The `DEALLOCATE` statement removes it.

For details about auto-incremented, positional, and named parameters, see Prepared Statements.

PREPARE

Prepare a statement named `query_person`:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $name_start)
      AND age >= $minimum_age;
```

Syntax

EXECUTE

Execute a prepared statement with named arguments:

```
EXECUTE query_person(name_start := 'B', minimum_age := 40);
```

Execute a prepared statement with positional arguments:

```
EXECUTE add_values(20, 22);
```

Syntax

DEALLOCATE

Remove a prepared statement:

```
DEALLOCATE query_person;
```

The optional PREPARE keyword is accepted for PostgreSQL compatibility:

```
DEALLOCATE PREPARE add_values;
```

Syntax

SELECT Statement

The SELECT statement retrieves rows from the database.

Examples

Select all columns from the table `tbl`:

```
SELECT * FROM tbl;
```

Select the rows from `tbl`:

```
SELECT j FROM tbl WHERE i = 3;
```

Perform an aggregate grouped by the column `i`:

```
SELECT i, sum(j) FROM tbl GROUP BY i;
```

Select only the top 3 rows from the `tbl`:

```
SELECT * FROM tbl ORDER BY i DESC LIMIT 3;
```

Join two tables together using the USING clause:

```
SELECT * FROM t1 JOIN t2 USING (a, b);
```

Use column indexes to select the first and third column from the table `tbl`:

```
SELECT #1, #3 FROM tbl;
```

Select all unique cities from the addresses table:

```
SELECT DISTINCT city FROM addresses;
```

Return a STRUCT by using a row variable:

```
SELECT d
FROM (SELECT 1 AS a, 2 AS b) d;
```

Syntax

The SELECT statement retrieves rows from the database. The canonical order of a SELECT statement is as follows, with less common clauses being indented:

```
SELECT <select_list>
FROM <tables>
    USING SAMPLE <sample_expression>
WHERE <condition>
GROUP BY <groups>
HAVING <group_filter>
    WINDOW <window_expression>
    QUALIFY <qualify_filter>
ORDER BY <order_expression>
LIMIT <n>;
```

Optionally, the SELECT statement can be prefixed with a WITH clause.

As the SELECT statement is so complex, we have split up the syntax diagrams into several parts. The full syntax diagram can be found at the bottom of the page.

SELECT Clause

The SELECT clause specifies the list of columns that will be returned by the query. While it appears first in the clause, *logically* the expressions here are executed only at the end. The SELECT clause can contain arbitrary expressions that transform the output, as well as aggregates and window functions. The DISTINCT keyword ensures that only unique tuples are returned.

> Column names are case-insensitive. See the Rules for Case Sensitivity for more details.

FROM Clause

The FROM clause specifies the *source* of the data on which the remainder of the query should operate. Logically, the FROM clause is where the query starts execution. The FROM clause can contain a single table, a combination of multiple tables that are joined together, or another SELECT query inside a subquery node.

SAMPLE Clause

The SAMPLE clause allows you to run the query on a sample from the base table. This can significantly speed up processing of queries, at the expense of accuracy in the result. Samples can also be used to quickly see a snapshot of the data when exploring a dataset. The SAMPLE clause is applied right after anything in the FROM clause (i.e., after any joins, but before the where clause or any aggregates). See the Samples page for more information.

WHERE Clause

The WHERE clause specifies any filters to apply to the data. This allows you to select only a subset of the data in which you are interested. Logically the WHERE clause is applied immediately after the FROM clause.

GROUP BY and HAVING Clauses

The GROUP BY clause specifies which grouping columns should be used to perform any aggregations in the SELECT clause. If the GROUP BY clause is specified, the query is always an aggregate query, even if no aggregations are present in the SELECT clause.

WINDOW Clause

The WINDOW clause allows you to specify named windows that can be used within window functions. These are useful when you have multiple window functions, as they allow you to avoid repeating the same window clause.

`QUALIFY` Clause

The `QUALIFY` clause is used to filter the result of `WINDOW` functions.

`ORDER BY`, `LIMIT` and `OFFSET` Clauses

`ORDER BY`, `LIMIT` and `OFFSET` are output modifiers. Logically they are applied at the very end of the query. The `ORDER BY` clause sorts the rows on the sorting criteria in either ascending or descending order. The `LIMIT` clause restricts the amount of rows fetched, while the `OFFSET` clause indicates at which position to start reading the values.

`VALUES` List

A `VALUES` list is a set of values that is supplied instead of a `SELECT` statement.

Row IDs

For each table, the `rowid` pseudocolumn returns the row identifiers based on the physical storage.

```
CREATE TABLE t (id INTEGER, content VARCHAR);
INSERT INTO t VALUES (42, 'hello'), (43, 'world');
SELECT rowid, id, content FROM t;
```

rowid	id	content
0	42	hello
1	43	world

In the current storage, these identifiers are contiguous unsigned integers (0, 1, ...) if no rows were deleted. Deletions introduce gaps in the rowids which may be reclaimed later:

```
CREATE OR REPLACE TABLE t AS (FROM range(10) r(i));
DELETE FROM t WHERE i % 2 = 0;
SELECT rowid FROM t;
```

rowid
1
3
5
7
9

The `rowid` values are stable within a transaction.

> **Best practice.** It is strongly advised to avoid using rowids as identifiers.

> If there is a user-defined column named `rowid`, it shadows the `rowid` pseudocolumn.

Common Table Expressions

Full Syntax Diagram

Below is the full syntax diagram of the SELECT statement:

SET and RESET Statements

The SET statement modifies the provided DuckDB configuration option at the specified scope.

Examples

Update the `memory_limit` configuration value:

```
SET memory_limit = '10GB';
```

Configure the system to use 1 thread:

```
SET threads = 1;
```

Or use the TO keyword:

```
SET threads TO 1;
```

Change configuration option to default value:

```
RESET threads;
```

Retrieve configuration value:

```
SELECT current_setting('threads');
```

Set the default collation for the session:

```
SET SESSION default_collation = 'nocase';
```

Set a Global Variable

Set the default sort order globally:

```
SET GLOBAL default_order = 'DESC';
```

Set the default threads globally:

```
SET GLOBAL threads = 4;
```

Syntax

SET updates a DuckDB configuration option to the provided value.

RESET

The RESET statement changes the given DuckDB configuration option to the default value.

Scopes

Configuration options can have different scopes:

- `GLOBAL`: Configuration value is used (or reset) across the entire DuckDB instance.
- `SESSION`: Configuration value is used (or reset) only for the current session attached to a DuckDB instance.
- `LOCAL`: Not yet implemented.

When not specified, the default scope for the configuration option is used. For most options this is `GLOBAL`.

> `GLOBAL` settings apply to the current DuckDB instance. They are not written to the database file or persisted across DuckDB instances. If all connections to an instance are closed and a new instance is started, configuration options use their defaults unless they are set again.

For example, in DuckDB 2.0 the `TimeZone` option defaults to the `SESSION` scope, so the following two statements are equivalent, and both change the time zone for the current session only:

```
SET TimeZone = 'UTC';
SET SESSION TimeZone = 'UTC';
```

To change the time zone for every session connected to the instance, use the `GLOBAL` scope explicitly:

```
SET GLOBAL TimeZone = 'UTC';
```

Configuration

See the Configuration page for the full list of configuration options.

SET VARIABLE and RESET VARIABLE Statements

DuckDB supports the definition of SQL-level variables using the `SET VARIABLE` and `RESET VARIABLE` statements.

Variable Scopes

DuckDB supports two levels of variable scopes:

Scope	Description
`SESSION`	Variables with a `SESSION` scope are local to you and only affect the current session.
`GLOBAL`	Variables with a `GLOBAL` scope are specific configuration option variables that affect the entire DuckDB instance and all sessions. For example, see Set a Global Variable.

SET VARIABLE

The `SET VARIABLE` statement assigns a value to a variable, which can be accessed using the `getvariable` call:

```
SET VARIABLE my_var = 30;
SELECT 20 + getvariable('my_var') AS total;
```

total
50

> New The `$<variable>` shorthand was introduced in DuckDB 2.0.

As a shorthand for `getvariable`, a variable can be referenced directly with the `$<variable>` syntax anywhere an expression is allowed:

```
SET VARIABLE my_var = 30;
SELECT 20 + $my_var AS total;
```

Variable names that are not valid identifiers can be quoted, for example `$"my var"`.

If `SET VARIABLE` is invoked on an existing variable, it will overwrite its value:

```
SET VARIABLE my_var = 30;
SET VARIABLE my_var = 100;
SELECT 20 + getvariable('my_var') AS total;
```

total
120

Variables can have different types:

```
SET VARIABLE my_date = DATE '2018-07-13';
SET VARIABLE my_string = 'Hello world';
SET VARIABLE my_map = MAP {'k1': 10, 'k2': 20};
```

Variables can also be assigned to results of queries:

```
-- write some CSV files
COPY (SELECT 42 AS a) TO 'test1.csv';
COPY (SELECT 84 AS a) TO 'test2.csv';

-- add a list of CSV files to a table
CREATE TABLE csv_files (file VARCHAR);
INSERT INTO csv_files VALUES ('test1.csv'), ('test2.csv');

-- initialize a variable with the list of csv files
SET VARIABLE list_of_files = (SELECT list(file) FROM csv_files);

-- read the CSV files
SELECT * FROM read_csv(getvariable('list_of_files'), filename := True);
```

a	filename
42	test1.csv
84	test2.csv

If a variable is not set, the `getvariable` function returns `NULL`:

```
SELECT getvariable('undefined_var') AS result;
```

result
NULL

The `getvariable` function can also be used in a COLUMNS expression:

```
SET VARIABLE column_to_exclude = 'col1';
CREATE TABLE tbl AS SELECT 12 AS col0, 34 AS col1, 56 AS col2;
SELECT COLUMNS(lambda c: c != getvariable('column_to_exclude')) FROM tbl;
```

col0	col2
12	56

The $name Shorthand

> New Referencing a variable with the `$name` shorthand was introduced in DuckDB 2.0.

As a shorthand for `getvariable`, you can reference a session variable directly with `$name`:

```
SET VARIABLE my_var = 30;
SELECT 20 + $my_var AS total;
```

total
50

For a variable that is set, `$name` and `getvariable('name')` return the same value:

```
SELECT $my_var = getvariable('my_var') AS same;
```

same
true

The two forms differ when the variable is not set: `getvariable` returns `NULL`, while a `$name` reference raises an error.

```
SELECT $undefined_var;
```

```
Invalid Input Error:
Values were not provided for the following parameters: undefined_var
```

This is because `$name` is a prepared statement parameter that is filled from the session variable of the same name. This also lets variables act as default values for prepared statements.

Syntax

RESET VARIABLE

The `RESET VARIABLE` statement unsets a variable.

```
SET VARIABLE my_var = 30;
RESET VARIABLE my_var;
SELECT getvariable('my_var') AS my_var;
```

my_var
NULL

Syntax

SHOW, SHOW DATABASES, and SHOW SCHEMAS Statements

SHOW Statement

The SHOW statement can also show the schema of a table, view or query, like DESCRIBE. As of DuckDB 2.0, this usage is deprecated and prints a warning; use DESCRIBE instead.

> New The following feature was introduced in DuckDB 2.0.

SHOW can also print a configuration setting when the name is not a table (PostgreSQL-style):

```
SET threads = 42;
SHOW threads;
```

threads
42

If both a table and a setting share the name, SHOW describes the table and prints a deprecation warning (use DESCRIBE for tables). DESCRIBE never resolves a setting.

The show_behavior setting configures how SHOW resolves a bare identifier:

- 'auto' (default): describe a table if one exists, otherwise show the setting.
- 'table': always describe a table.
- 'setting': always show a setting.

```
SET show_behavior = 'setting';
```

SHOW DATABASES Statement

The SHOW DATABASES statement shows a list of all attached databases:

```
ATTACH 'my.duckdb' AS my_database;
SHOW DATABASES;
```

database_name
memory
my_database

```
DETACH my_database;
SHOW DATABASES;
```

database_name
memory

SHOW SCHEMAS Statement

> This statement was introduced in DuckDB 1.5.

The SHOW SCHEMAS statement shows a list of all schemas across non-internal databases:

```
SHOW SCHEMAS;
```

database_name	schema_name	current
memory	main	true

The `current` column indicates which schema is the default schema (set via the USE statement).

SUMMARIZE Statement

The SUMMARIZE statement returns summary statistics for a table, view or a query.

Usage

```
SUMMARIZE tbl;
```

To summarize a query, prepend SUMMARIZE to a query.

```
SUMMARIZE SELECT * FROM tbl;
```

See Also

For more examples, see the guide on SUMMARIZE.

Transaction Management

DuckDB supports ACID database transactions. Transactions provide isolation, i.e., changes made by a transaction are not visible from concurrent transactions until it is committed. A transaction can also be aborted, which discards any changes it made so far.

Statements

DuckDB provides the following statements for transaction management.

Starting a Transaction

To start a transaction, run:

```
BEGIN TRANSACTION;
```

Committing a Transaction

You can commit a transaction to make it visible to other transactions and to write it to persistent storage (if using DuckDB in persistent mode). To commit a transaction, run:

```
COMMIT;
```

If you are not in an active transaction, the `COMMIT` statement will fail.

Rolling Back a Transaction

You can abort a transaction. This operation, also known as rolling back, will discard any changes the transaction made to the database. To abort a transaction, run:

```
ROLLBACK;
```

You can also use the abort command, which has an identical behavior:

```
ABORT;
```

If you are not in an active transaction, the `ROLLBACK` and `ABORT` statements will fail.

Multi-Statement Transactions

When multiple SQL statements are submitted together (e.g., separated by semicolons), they are executed one after another in auto-commit mode: each statement that succeeds is committed immediately, and execution stops at the first statement that fails. Statements that ran before the failing one are *not* rolled back. To make a batch atomic, wrap it in an explicit transaction using `BEGIN TRANSACTION` and `COMMIT`.

Isolation Level

DuckDB's concurrency model guarantees snapshot isolation. Transactions that violate this isolation level are aborted.

Using PostgreSQL's transaction isolation levels, DuckDB guarantees *repeatable reads*.

Example

We illustrate the use of transactions through a simple example.

```
CREATE TABLE person (name VARCHAR, age BIGINT);

BEGIN TRANSACTION;
INSERT INTO person VALUES ('Ada', 52);
COMMIT;

BEGIN TRANSACTION;
DELETE FROM person WHERE name = 'Ada';
INSERT INTO person VALUES ('Bruce', 39);
ROLLBACK;

SELECT * FROM person;
```

The first transaction (inserting “Ada”) was committed but the second (deleting “Ada” and inserting “Bruce”) was aborted. Therefore, the resulting table will only contain `<'Ada', 52>`.

UNPIVOT Statement

The UNPIVOT statement allows multiple columns to be stacked into fewer columns. In the basic case, multiple columns are stacked into two columns: a NAME column (which contains the name of the source column) and a VALUE column (which contains the value from the source column).

DuckDB implements both the SQL Standard UNPIVOT syntax and a simplified UNPIVOT syntax. Both can utilize a COLUMNS expression to automatically detect the columns to unpivot. PIVOT_LONGER may also be used in place of the UNPIVOT keyword.

For details on how the UNPIVOT statement is implemented, see the Pivot Internals site.

> The PIVOT statement is the inverse of the UNPIVOT statement.

Simplified UNPIVOT Syntax

The full syntax diagram is below, but the simplified UNPIVOT syntax can be summarized using spreadsheet pivot table naming conventions as:

```
UNPIVOT <dataset>
ON <column(s)>
INTO
    NAME <name_column_name>
    VALUE <value_column_name(s)>
ORDER BY <column(s)_with_order_direction(s)>
LIMIT <number_of_rows>;
```

Example Data

All examples use the dataset produced by the queries below:

```
CREATE OR REPLACE TABLE monthly_sales
    (empid INTEGER, dept TEXT, Jan INTEGER, Feb INTEGER, Mar INTEGER, Apr INTEGER, May INTEGER, Jun
INTEGER);
INSERT INTO monthly_sales VALUES
    (1, 'electronics', 1, 2, 3, 4, 5, 6),
    (2, 'clothes', 10, 20, 30, 40, 50, 60),
    (3, 'cars', 100, 200, 300, 400, 500, 600);

FROM monthly_sales;
```

empid	dept	Jan	Feb	Mar	Apr	May	Jun
1	electronics	1	2	3	4	5	6
2	clothes	10	20	30	40	50	60
3	cars	100	200	300	400	500	600

UNPIVOT Manually

The most typical UNPIVOT transformation is to take already pivoted data and re-stack it into a column each for the name and value. In this case, all months will be stacked into a month column and a sales column.

```
UNPIVOT monthly_sales
ON jan, feb, mar, apr, may, jun
INTO
    NAME month
    VALUE sales;
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

`UNPIVOT` Dynamically Using `COLUMNS` Expression

In many cases, the number of columns to unpivot is not easy to predetermine ahead of time. In the case of this dataset, the query above would have to change each time a new month is added. The COLUMNS expression can be used to select all columns that are not `empid` or `dept`. This enables dynamic unpivoting that will work regardless of how many months are added. The query below returns identical results to the one above.

```
UNPIVOT monthly_sales
ON COLUMNS(* EXCLUDE (empid, dept))
INTO
    NAME month
    VALUE sales;
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30

empid	dept	month	sales
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

`UNPIVOT` into Multiple Value Columns

The `UNPIVOT` statement has additional flexibility: more than 2 destination columns are supported. This can be useful when the goal is to reduce the extent to which a dataset is pivoted, but not completely stack all pivoted columns. To demonstrate this, the query below will generate a dataset with a separate column for the number of each month within the quarter (month 1, 2, or 3), and a separate row for each quarter. Since there are fewer quarters than months, this does make the dataset longer, but not as long as the above.

To accomplish this, multiple sets of columns are included in the `ON` clause. The `q1` and `q2` aliases are optional. The number of columns in each set of columns in the `ON` clause must match the number of columns in the `VALUE` clause.

```
UNPIVOT monthly_sales
    ON (jan, feb, mar) AS q1, (apr, may, jun) AS q2
    INTO
        NAME quarter
        VALUE month_1_sales, month_2_sales, month_3_sales;
```

empid	dept	quarter	month_1_sales	month_2_sales	month_3_sales
1	electronics	q1	1	2	3
1	electronics	q2	4	5	6
2	clothes	q1	10	20	30
2	clothes	q2	40	50	60
3	cars	q1	100	200	300
3	cars	q2	400	500	600

Using `UNPIVOT` within a `SELECT` Statement

The `UNPIVOT` statement may be included within a `SELECT` statement as a CTE (a Common Table Expression, or WITH clause), or a subquery. This allows for an `UNPIVOT` to be used alongside other SQL logic, as well as for multiple `UNPIVOT`s to be used in one query.

No `SELECT` is needed within the CTE, the `UNPIVOT` keyword can be thought of as taking its place.

```
WITH unpivot_alias AS (
    UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME month
        VALUE sales
```

```
)
SELECT * FROM unpivot_alias;
```

An UNPIVOT may be used in a subquery and must be wrapped in parentheses. Note that this behavior is different than the SQL Standard Unpivot, as illustrated in subsequent examples.

```
SELECT *
FROM (
    UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME month
        VALUE sales
) unpivot_alias;
```

Expressions within UNPIVOT Statements

DuckDB allows expressions within the UNPIVOT statements, provided that they only involve a single column. These can be used to perform computations as well as explicit casts. For example:

```
UNPIVOT
    (SELECT 42 AS col1, 'woot' AS col2)
    ON
        (col1 * 2)::VARCHAR,
        col2;
```

name	value
col1	84
col2	woot

Simplified UNPIVOT Full Syntax Diagram

Below is the full syntax diagram of the UNPIVOT statement.

SQL Standard UNPIVOT Syntax

The full syntax diagram is below, but the SQL Standard UNPIVOT syntax can be summarized as:

```
FROM [dataset]
UNPIVOT [INCLUDE NULLS] (
    [value-column-name(s)]
    FOR [name-column-name] IN [column(s)]
);
```

Note that only one column can be included in the `name-column-name` expression.

SQL Standard UNPIVOT Manually

To complete the basic UNPIVOT operation using the SQL standard syntax, only a few additions are needed.

```
FROM monthly_sales UNPIVOT (
    sales
    FOR month IN (jan, feb, mar, apr, may, jun)
);
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

SQL Standard `UNPIVOT` Dynamically Using the `COLUMNS` Expression

The `COLUMNS` expression can be used to determine the `IN` list of columns dynamically. This will continue to work even if additional `month` columns are added to the dataset. It produces the same result as the query above.

```
FROM monthly_sales UNPIVOT (
    sales
    FOR month IN (columns(* EXCLUDE (empid, dept)))
);
```

SQL Standard `UNPIVOT` into Multiple Value Columns

The `UNPIVOT` statement has additional flexibility: more than 2 destination columns are supported. This can be useful when the goal is to reduce the extent to which a dataset is pivoted, but not completely stack all pivoted columns. To demonstrate this, the query below will generate a dataset with a separate column for the number of each month within the quarter (month 1, 2, or 3), and a separate row for each quarter. Since there are fewer quarters than months, this does make the dataset longer, but not as long as the above.

To accomplish this, multiple columns are included in the `value-column-name` portion of the `UNPIVOT` statement. Multiple sets of columns are included in the `IN` clause. The `q1` and `q2` aliases are optional. The number of columns in each set of columns in the `IN` clause must match the number of columns in the `value-column-name` portion.

```
FROM monthly_sales
UNPIVOT (
    (month_1_sales, month_2_sales, month_3_sales)
    FOR quarter IN (
        (jan, feb, mar) AS q1,
        (apr, may, jun) AS q2
    )
);
```

empid	dept	quarter	month_1_sales	month_2_sales	month_3_sales
1	electronics	q1	1	2	3
1	electronics	q2	4	5	6
2	clothes	q1	10	20	30
2	clothes	q2	40	50	60
3	cars	q1	100	200	300
3	cars	q2	400	500	600

SQL Standard `UNPIVOT` Full Syntax Diagram

Below is the full syntax diagram of the SQL Standard version of the `UNPIVOT` statement.

UPDATE Statement

The `UPDATE` statement modifies the values of rows in a table.

Examples

For every row where `i` is `NULL`, set the value to 0 instead:

```
UPDATE tbl
SET i = 0
WHERE i IS NULL;
```

Set all values of `i` to 1 and all values of `j` to 2:

```
UPDATE tbl
SET i = 1, j = 2;
```

Syntax

`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the `SET` clause; columns not explicitly modified retain their previous values.

`RETURNING` Clause

The `RETURNING` clause returns the values of the rows that were updated. It uses the same syntax as the `SELECT` clause, except the `DISTINCT` modifier is not supported. The returned values reflect the state of the rows after the update.

```
CREATE TABLE tbl (i INTEGER, j INTEGER);
INSERT INTO tbl VALUES (0, 1), (1, 2);
UPDATE tbl SET j = j + 10 WHERE i = 1 RETURNING *;
```

i	j
1	12

Update from Other Table

A table can be updated based upon values from another table. This can be done by specifying a table in a FROM clause, or using a sub-select statement. Both approaches have the benefit of completing the UPDATE operation in bulk for increased performance.

```
CREATE OR REPLACE TABLE original AS
    SELECT 1 AS key, 'original value' AS value
    UNION ALL
    SELECT 2 AS key, 'original value 2' AS value;

CREATE OR REPLACE TABLE new AS
    SELECT 1 AS key, 'new value' AS value
    UNION ALL
    SELECT 2 AS key, 'new value 2' AS value;

SELECT *
FROM original;
```

key	value
1	original value
2	original value 2

```
UPDATE original
    SET value = new.value
    FROM new
    WHERE original.key = new.key;
```

Or:

```
UPDATE original
    SET value = (
        SELECT
            new.value
        FROM new
        WHERE original.key = new.key
    );

SELECT *
FROM original;
```

key	value
1	new value
2	new value 2

Multiple Matching Source Rows

If a target row matches multiple rows in the FROM clause, the target row is updated only once, using the values of one of the matching source rows. Which source row is used is not specified. For example, the following query sets v to either 10 or 20:

```
CREATE TABLE target (id INTEGER, v INTEGER);
INSERT INTO target VALUES (1, 0);
CREATE TABLE source (id INTEGER, v INTEGER);
INSERT INTO source VALUES (1, 10), (1, 20);

UPDATE target
    SET v = source.v
    FROM source
    WHERE target.id = source.id;
```

To get deterministic results, ensure that each target row matches at most one source row, e.g., by deduplicating the source. In contrast, the sub-select form throws an error if the subquery returns more than one row (unless the `scalar_subquery_error_on_multiple_rows` setting is set to `false`), and `MERGE INTO` throws an error if a target row matches more than one source row.

Update from Same Table

The only difference between this case and the above is that a different table alias must be specified on both the target table and the source table. In this example `AS true_original` and `AS new` are both required.

```
UPDATE original AS true_original
    SET value = (
        SELECT
            new.value || ' a change!' AS value
        FROM original AS new
        WHERE true_original.key = new.key
    );
```

Update Using Joins

To select the rows to update, `UPDATE` statements can use the `FROM` clause and express joins via the `WHERE` clause. For example:

```
CREATE TABLE city (name VARCHAR, revenue BIGINT, country_code VARCHAR);
CREATE TABLE country (code VARCHAR, name VARCHAR);
INSERT INTO city VALUES ('Paris', 700, 'FR'), ('Lyon', 200, 'FR'), ('Brussels', 400, 'BE');
INSERT INTO country VALUES ('FR', 'France'), ('BE', 'Belgium');
```

To increase the revenue of all cities in France, join the `city` and the `country` tables, and filter on the latter:

```
UPDATE city
SET revenue = revenue + 100
FROM country
WHERE city.country_code = country.code
  AND country.name = 'France';

SELECT *
FROM city;
```

name	revenue	country_code
Paris	800	FR
Lyon	300	FR
Brussels	400	BE

Upsert (Insert or Update)

See the Insert documentation for details.

UPDATE EXTENSIONS

The `UPDATE EXTENSIONS` statement allows synchronizing the locally installed extension state with the repository that published a given extension. This statement is the recommended way to keep up to date with new features or bug fixes being rolled out by extension developers.

Note that DuckDB extensions cannot be reloaded during runtime, therefore `UPDATE EXTENSIONS` does not reload the updated extensions. To use the updated extensions, restart the process running DuckDB.

Updating All Extensions

To update all extensions installed for the DuckDB version of your client:

```
UPDATE EXTENSIONS;
```

This will iterate over the extensions and return their repositories and the update result:

extension_name varchar	repository varchar	update_result varchar	previous_version varchar	current_version varchar
iceberg	core_nightly	UPDATED	6386ab5	b3ec51a
tpch	core	NO_UPDATE_AVAILABLE	v2.0.0	v2.0.0
httpfs	core	NO_UPDATE_AVAILABLE	77392a7	77392a7
spatial	core	NO_UPDATE_AVAILABLE	4e066c1	4e066c1
aws	core_nightly	NO_UPDATE_AVAILABLE	d3c5013	d3c5013

Extensions that are statically linked into the DuckDB binary, such as `json`, `parquet` and `icu` in the official DuckDB builds, are not updated by this statement.

Updating Selected Extensions

For more fine-grained control, you can also provide a list of extension names to be updated:

```
UPDATE EXTENSIONS (name_a, name_b, name_c);
```

How It Works

`UPDATE EXTENSIONS` is implemented by storing, if available, the ETag information, and sending a GET request conditional on the fact that the remote extension is different (using the ETag as proxy) than the local available one. This ensures that subsequent `UPDATE EXTENSIONS` calls – if the remote state has not changed – are inexpensive.

If a change is found for a given extension, DuckDB performs the following operation. For example, if `name_a` and `name_c` changed, then:

```
UPDATE EXTENSIONS (name_a, name_b, name_c);
```

This will result in the following commands:

```
FORCE INSTALL name_a;
FORCE INSTALL name_c;
```

USE Statement

The USE statement selects a database and optional schema, or just a schema to use as the default.

Examples

```
--- Sets the 'memory' database as the default. Will use 'main' schema implicitly or error
--- if it does not exist.
USE memory;
--- Sets the 'duck.main' database and schema as the default
USE duck.main;
-- Sets the `main` schema of the currently selected database as the default, in this case 'duck.main'
USE main;
```

Syntax

The USE statement sets a default database, schema or database/schema combination to use for future operations. For instance, tables created without providing a fully qualified table name will be created in the default database.

VACUUM Statement

The VACUUM statement only has basic support in DuckDB and is mostly provided for PostgreSQL-compatibility.

Some variants of it, such as when calling for a given column, recompute the distinct statistics (the number of distinct entities) if they have become stale due to updates.

> **Warning.** The behavior of VACUUM is not consistent with PostgreSQL semantics and it is likely going to change in the future.

Examples

No-op:

```
VACUUM;
```

No-op:

```
VACUUM ANALYZE;
```

Calling VACUUM on a given table-column pair rebuilds statistics for the table and column:

```
VACUUM my_table(my_column);
VACUUM ANALYZE my_table(my_column);
```

The following operation is not supported:

```
VACUUM FULL;
```

```
Not implemented Error:
FULL is not yet implemented
```

Vacuum with Indexes

> New Starting with DuckDB 2.0, checkpoint vacuuming also compacts tables with ART indexes.

When a checkpoint vacuums (compacts) row groups, the row IDs of the moved rows change. On databases using storage version 2.0.0 or later, DuckDB incrementally remaps the affected ART index entries to the new row IDs, so tables with ART indexes (including those created by `PRIMARY KEY` and `UNIQUE` constraints) are vacuumed by default and no full index rebuild is required.

On databases using an older storage version, the row IDs cannot be remapped, so tables with indexes are skipped by checkpoint vacuuming. For such databases, the deprecated `vacuum_rebuild_indexes` setting provides a fallback: it specifies a maximum table row count up to which the indexes are rebuilt after vacuuming (the default, `0`, disables the fallback). It can only be set before the database is started or per database as an `ATTACH` option.

Reclaiming Space

The VACUUM statement does not reclaim space. For instructions on reclaiming space, refer to the "Reclaiming space" page.

Syntax

Query Syntax

SELECT Clause

The SELECT clause specifies the list of columns that will be returned by the query. While it appears first in the clause, *logically* the expressions here are executed only at the end. The SELECT clause can contain arbitrary expressions that transform the output, as well as aggregates and window functions.

Examples

Select all columns from the table called `tbl`:

```
SELECT * FROM tbl;
```

Perform arithmetic on the columns in a table, and provide an alias:

```
SELECT col1 + col2 AS res, sqrt(col1) AS root FROM tbl;
```

Use prefix aliases:

```
SELECT
    res: col1 + col2,
    root: sqrt(col1)
FROM tbl;
```

Select all unique cities from the `addresses` table:

```
SELECT DISTINCT city FROM addresses;
```

Return the total number of rows in the `addresses` table:

```
SELECT count(*) FROM addresses;
```

Select all columns except the city column from the `addresses` table:

```
SELECT * EXCLUDE (city) FROM addresses;
```

Select all columns from the `addresses` table, but replace `city` with `lower(city)`:

```
SELECT * REPLACE (lower(city) AS city) FROM addresses;
```

Select all columns matching the given regular expression from the table:

```
SELECT COLUMNS('number\d+') FROM addresses;
```

Compute a function on all given columns of a table:

```
SELECT min(COLUMNS(*)) FROM addresses;
```

To select columns with spaces or special characters, use double quotes (`"`):

```
SELECT "Some Column Name" FROM tbl;
```

Syntax

SELECT List

The SELECT clause contains a list of expressions that specify the result of a query. The select list can refer to any columns in the FROM clause, and combine them using expressions. As the output of a SQL query is a table – every expression in the SELECT clause also has a name. The expressions can be explicitly named using the AS clause (e.g., `expr AS name`). If a name is not provided by the user the expressions are named automatically by the system.

> Column names are case-insensitive. See the Rules for Case Sensitivity for more details.

Star Expressions

Select all columns from the table called `tbl`:

```
SELECT *
FROM tbl;
```

Select all columns matching the given regular expression from the table:

```
SELECT COLUMNS('number\d+')
FROM addresses;
```

The star expression is a special expression that expands to *multiple expressions* based on the contents of the FROM clause. In the simplest case, `*` expands to **all** expressions in the FROM clause. Columns can also be selected using regular expressions or lambda functions. See the star expression page for more details.

DISTINCT Clause

Select all unique cities from the addresses table:

```
SELECT DISTINCT city
FROM addresses;
```

The DISTINCT clause can be used to return **only** the unique rows in the result – so that any duplicate rows are filtered out.

> Queries starting with `SELECT DISTINCT` run deduplication, which is an expensive operation. Therefore, only use DISTINCT if necessary.

DISTINCT ON Clause

Select only the highest population city for each country:

```
SELECT DISTINCT ON(country) city, population
FROM cities
ORDER BY population DESC;
```

The `DISTINCT ON` clause returns only one row per unique value in the set of expressions as defined in the ON clause. If an `ORDER BY` clause is present, the row that is returned is the first row that is encountered as per the `ORDER BY` criteria. If an `ORDER BY` clause is not present, the first row that is encountered is not defined and can be any row in the table.

> When querying large datasets, using DISTINCT on all columns can be expensive. Therefore, consider using `DISTINCT ON` on a column (or a set of columns) which guarantees a sufficient degree of uniqueness for your results. For example, using `DISTINCT ON` on the key column(s) of a table guarantees full uniqueness.

Aggregates

Return the total number of rows in the addresses table:

```
SELECT count(*)
FROM addresses;
```

Return the total number of rows in the addresses table grouped by city:

```
SELECT city, count(*)
FROM addresses
GROUP BY city;
```

Aggregate functions are special functions that *combine* multiple rows into a single value. When aggregate functions are present in the `SELECT` clause, the query is turned into an aggregate query. In an aggregate query, **all** expressions must either be part of an aggregate function, or part of a group (as specified by the `GROUP BY clause`).

Window Functions

Generate a `row_number` column containing incremental identifiers for each row:

```
SELECT row_number() OVER ()
FROM sales;
```

Compute the difference between the current amount, and the previous amount, by order of time:

```
SELECT amount - lag(amount) OVER (ORDER BY time)
FROM sales;
```

Window functions are special functions that allow the computation of values relative to *other rows* in a result. Window functions are marked by the `OVER` clause which contains the *window specification*. The window specification defines the frame or context in which the window function is computed. See the window functions page for more information.

`unnest` Function

Unnest an array by one level:

```
SELECT unnest([1, 2, 3]);
```

Unnest a struct by one level:

```
SELECT unnest({'a': 42, 'b': 84});
```

The `unnest` function is a special function that can be used together with arrays, lists, or structs. The unnest function strips one level of nesting from the type. For example, `INTEGER[]` is transformed into `INTEGER`. `STRUCT(a INTEGER, b INTEGER)` is transformed into `a INTEGER, b INTEGER`. The unnest function can be used to transform nested types into regular scalar types, which makes them easier to operate on.

FROM and JOIN Clauses

The `FROM` clause specifies the *source* of the data on which the remainder of the query should operate. Logically, the `FROM` clause is where the query starts execution. The `FROM` clause can contain a single table, a combination of multiple tables that are joined together using `JOIN` clauses, or another `SELECT` query inside a subquery node. DuckDB also has an optional `FROM`-first syntax which enables you to also query without a `SELECT` statement.

Examples

Select all columns from the table called `tbl`:

```
SELECT *
FROM tbl;
```

Select all columns from the table using the FROM-first syntax:

```
FROM tbl
SELECT *;
```

Select all columns using the FROM-first syntax and omitting the `SELECT` clause:

```
FROM tbl;
```

Select all columns from the table called `tbl` through an alias `tn`:

```
SELECT tn.*
FROM tbl tn;
```

Use a prefix alias:

```
SELECT tn.*
FROM tn: tbl;
```

Select all columns from the table `tbl` in the schema `schema_name`:

```
SELECT *
FROM schema_name.tbl;
```

Select the column `i` from the table function `range`, where the first column of the range function is renamed to `i`:

```
SELECT t.i
FROM range(100) AS t(i);
```

Select all columns from the CSV file called `test.csv`:

```
SELECT *
FROM 'test.csv';
```

Select all columns from a subquery:

```
SELECT *
FROM (SELECT * FROM tbl);
```

Select the entire row of the table as a struct:

```
SELECT t
FROM t;
```

Select the entire row of the subquery as a struct (i.e., a single column):

```
SELECT t
FROM (SELECT unnest(generate_series(41, 43)) AS x, 'hello' AS y) t;
```

Join two tables together:

```
SELECT *
FROM tbl
JOIN other_table
  ON tbl.key = other_table.key;
```

Select a 10% sample from a table:

```
SELECT *
FROM tbl
TABLESAMPLE 10%;
```

Select a sample of 10 rows from a table:

```
SELECT *
FROM tbl
TABLESAMPLE 10 ROWS;
```

Use the FROM-first syntax with WHERE clause and aggregation:

```
FROM range(100) AS t(i)
SELECT sum(t.i)
WHERE i % 2 = 0;
```

Table Functions

Some functions in DuckDB return entire tables rather than individual values. These functions are accordingly called *table functions* and can be used with a FROM clause like regular table references. Examples include `read_csv`, `read_parquet`, `range`, `generate_series`, `repeat`, `unnest`, and `glob` (note that some of the examples here can be used as both scalar and table functions).

For example,

```
SELECT *
FROM 'test.csv';
```

is implicitly translated to a call of the `read_csv` table function:

```
SELECT *
FROM read_csv('test.csv');
```

All table functions support a `WITH ORDINALITY` suffix, which extends the returned table by an integer column `ordinality` that enumerates the generated rows starting at 1.

```
SELECT *
FROM read_csv('test.csv') WITH ORDINALITY;
```

Note that the same result could be achieved using the `row_number` window function. In the presence of joins, however, `WITH ORDINALITY` allows enumerating one side of the join instead of the final result set, without having to resort to sub-queries.

Joins

Joins are a fundamental relational operation used to connect two tables or relations horizontally. The relations are referred to as the *left* and *right* sides of the join based on how they are written in the join clause. Each result row has the columns from both relations.

A join uses a rule to match pairs of rows from each relation. Often this is a predicate, but there are other implied rules that may be specified.

Outer Joins

Rows that do not have any matches can still be returned if an OUTER join is specified. Outer joins can be one of:

- LEFT (All rows from the left relation appear at least once)
- RIGHT (All rows from the right relation appear at least once)
- FULL (All rows from both relations appear at least once)

A join that is not OUTER is INNER (only rows that get paired are returned).

When an unpaired row is returned, the attributes from the other table are set to NULL.

Cross Product Joins (Cartesian Product)

The simplest type of join is a `CROSS JOIN`. There are no conditions for this type of join, and it just returns all the possible pairs.

Return all pairs of rows:

```
SELECT a.*, b.*
FROM a
CROSS JOIN b;
```

This is equivalent to omitting the `JOIN` clause:

```
SELECT a.*, b.*
FROM a, b;
```

Conditional Joins

Most joins are specified by a predicate that connects attributes from one side to attributes from the other side. The conditions can be explicitly specified using an `ON` clause with the join (clearer) or implied by the `WHERE` clause (old-fashioned).

We use the `l_regions` and the `l_nations` tables from the TPC-H schema:

```
CREATE TABLE l_regions (
    r_regionkey INTEGER NOT NULL PRIMARY KEY,
    r_name      CHAR(25) NOT NULL,
    r_comment   VARCHAR(152)
);

CREATE TABLE l_nations (
    n_nationkey INTEGER NOT NULL PRIMARY KEY,
    n_name      CHAR(25) NOT NULL,
    n_regionkey INTEGER NOT NULL,
    n_comment   VARCHAR(152),
    FOREIGN KEY (n_regionkey) REFERENCES l_regions(r_regionkey)
);
```

Return the regions for the nations:

```
SELECT n.*, r.*
FROM l_nations n
JOIN l_regions r ON (n_regionkey = r_regionkey);
```

If the column names are the same and are required to be equal, then the simpler `USING` syntax can be used:

```
CREATE TABLE l_regions (regionkey INTEGER NOT NULL PRIMARY KEY,
                        name      CHAR(25) NOT NULL,
                        comment   VARCHAR(152));

CREATE TABLE l_nations (nationkey INTEGER NOT NULL PRIMARY KEY,
                        name      CHAR(25) NOT NULL,
                        regionkey INTEGER NOT NULL,
                        comment   VARCHAR(152),
                        FOREIGN KEY (regionkey) REFERENCES l_regions(regionkey));
```

Return the regions for the nations:

```
SELECT n.*, r.*
FROM l_nations n
JOIN l_regions r USING (regionkey);
```

The expressions do not have to be equalities – any predicate can be used:

Return the pairs of jobs where one ran longer but cost less:

```
SELECT s1.t_id, s2.t_id
FROM west s1, west s2
WHERE s1.time > s2.time
  AND s1.cost < s2.cost;
```

Natural Joins

Natural joins join two tables based on attributes that share the same name.

For example, take the following example with cities, airport codes and airport names. Note that both tables are intentionally incomplete, i.e., they do not have a matching pair in the other table.

```
CREATE TABLE city_airport (city_name VARCHAR, iata VARCHAR);
CREATE TABLE airport_names (iata VARCHAR, airport_name VARCHAR);
INSERT INTO city_airport VALUES
    ('Amsterdam', 'AMS'),
    ('Rotterdam', 'RTM'),
    ('Eindhoven', 'EIN'),
    ('Groningen', 'GRQ');
INSERT INTO airport_names VALUES
    ('AMS', 'Amsterdam Airport Schiphol'),
    ('RTM', 'Rotterdam The Hague Airport'),
    ('MST', 'Maastricht Aachen Airport');
```

To join the tables on their shared `IATA` attributes, run:

```
SELECT *
FROM city_airport
NATURAL JOIN airport_names;
```

This produces the following result:

city_name	iata	airport_name
Amsterdam	AMS	Amsterdam Airport Schiphol
Rotterdam	RTM	Rotterdam The Hague Airport

Note that only rows where the same `iata` attribute was present in both tables were included in the result.

We can also express this query using the vanilla `JOIN` clause with the `USING` keyword:

```
SELECT *
FROM city_airport
JOIN airport_names
USING (iata);
```

Semi and Anti Joins

Semi joins return rows from the left table that have at least one match in the right table. Anti joins return rows from the left table that have *no* matches in the right table. When using a semi or anti join the result will never have more rows than the left hand side table. Semi joins provide the same logic as the `IN` operator statement. Anti joins provide the same logic as the `NOT IN` operator, except anti joins ignore `NULL` values from the right table.

Semi Join Example Return a list of city–airport code pairs from the `city_airport` table where the airport name **is available** in the `airport_names` table:

```
SELECT *
FROM city_airport
SEMI JOIN airport_names
    USING (iata);
```

city_name	iata
Amsterdam	AMS
Rotterdam	RTM

This query is equivalent to:

```
SELECT *
FROM city_airport
WHERE iata IN (SELECT iata FROM airport_names);
```

Anti Join Example Return a list of city–airport code pairs from the `city_airport` table where the airport name **is not available** in the `airport_names` table:

```
SELECT *
FROM city_airport
ANTI JOIN airport_names
    USING (iata);
```

city_name	iata
Eindhoven	EIN
Groningen	GRQ

This query is equivalent to:

```
SELECT *
FROM city_airport
WHERE iata NOT IN (SELECT iata FROM airport_names WHERE iata IS NOT NULL);
```

Lateral Joins

The LATERAL keyword allows subqueries in the FROM clause to refer to previous subqueries. This feature is also known as a *lateral join*.

```
SELECT *
FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);
```

i	j
0	1
2	3
1	2

Lateral joins are a generalization of correlated subqueries, as they can return multiple values per input value rather than only a single value.

```
SELECT *
FROM
    generate_series(0, 1) t(i),
    LATERAL (SELECT i + 10 UNION ALL SELECT i + 100) t2(j);
```

i	j
0	10
1	11
0	100
1	101

It may be helpful to think about LATERAL as a loop where we iterate through the rows of the first subquery and use it as input to the second (LATERAL) subquery. In the examples above, we iterate through table `t` and refer to its column `i` from the definition of table `t2`. The rows of `t2` form column `j` in the result.

It is possible to refer to multiple attributes from the LATERAL subquery. Using the table from the first example:

```
CREATE TABLE t1 AS
    SELECT *
    FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);

SELECT *
    FROM t1, LATERAL (SELECT i + j) t2(k)
    ORDER BY ALL;
```

i	j	k
0	1	1
1	2	3
2	3	5

> DuckDB detects when LATERAL joins should be used, making the use of the LATERAL keyword optional.

Positional Joins

When working with data frames or other embedded tables of the same size, the rows may have a natural correspondence based on their physical order. In scripting languages, this is easily expressed using a loop:

```
for (i = 0; i < n; i++) {
    f(t1.a[i], t2.b[i]);
}
```

It is difficult to express this in standard SQL because relational tables are not ordered, but imported tables such as data frames or disk files (like CSVs or Parquet files) do have a natural ordering.

Connecting them using this ordering is called a *positional join:*

```
CREATE TABLE t1 (x INTEGER);
CREATE TABLE t2 (s VARCHAR);

INSERT INTO t1 VALUES (1), (2), (3);
INSERT INTO t2 VALUES ('a'), ('b');

SELECT *
FROM t1
POSITIONAL JOIN t2;
```

x	s
1	a
2	b
3	NULL

Positional joins are always `FULL OUTER` joins, i.e., the resulting table has the length of the longer input table and the missing entries are filled with `NULL` values.

As-Of Joins

A common operation when working with temporal or similarly-ordered data is to find the nearest (first) event in a reference table (such as prices). This is called an *as-of join:*

Attach prices to stock trades:

```
SELECT t.*, p.price
FROM trades t
ASOF JOIN prices p
       ON t.symbol = p.symbol AND t.when >= p.when;
```

The ASOF join requires at least one inequality condition on the ordering field. The inequality can be any inequality condition (>=, >, <=, <) on any data type, but the most common form is >= on a temporal type. Any other conditions must be equalities (or `NOT DISTINCT`). This means that the left/right order of the tables is significant.

ASOF joins each left side row with at most one right side row. It can be specified as an OUTER join to find unpaired rows (e.g., trades without prices or prices which have no trades.)

Attach prices or NULLs to stock trades:

```
SELECT *
FROM trades t
ASOF LEFT JOIN prices p
            ON t.symbol = p.symbol
           AND t.when >= p.when;
```

ASOF joins can also specify join conditions on matching column names with the `USING` syntax, but the *last* attribute in the list must be the inequality, which will be greater than or equal to (>=):

```
SELECT *
FROM trades t
ASOF JOIN prices p USING (symbol, "when");
```

Returns symbol, trades.when, price (but NOT prices.when):

If you combine `USING` with a `SELECT *` like this, the query will return the left side (probe) column values for the matches, not the right side (build) column values. To get the `prices` times in the example, you will need to list the columns explicitly:

```
SELECT t.symbol, t.when AS trade_when, p.when AS price_when, price
FROM trades t
ASOF LEFT JOIN prices p USING (symbol, "when");
```

Nearest Joins

> New Support for nearest joins was introduced in DuckDB 2.0.

A nearest join matches each left side row with the top-k right side rows according to a similarity or distance measure. This turns top-k similarity search into a join clause, which is handy for vector and embedding workloads.

The join is written as `NEAREST <k> BY SIMILARITY <expression>` (higher is closer) or `NEAREST <k> BY DISTANCE <expression>` (lower is closer). Prefixing it with `APPROX` performs an approximate search, which can be faster for large inputs:

```
SELECT q.user_id, t.product_id
FROM users q
    INNER JOIN products t APPROX NEAREST 2
    BY SIMILARITY array_cosine_similarity(q.embedding, t.embedding);
```

Nearest joins can be combined with the regular join types, for example `LEFT JOIN`:

```
SELECT q.user_id, t.product_id
FROM users q
    LEFT JOIN products t NEAREST 1
    BY DISTANCE array_distance(q.embedding, t.embedding);
```

Self-Joins

DuckDB allows self-joins for all types of joins. Note that tables need to be aliased, using the same table name without aliases will result in an error:

```
CREATE TABLE t (x INTEGER);
SELECT * FROM t JOIN t USING(x);
```

```
Binder Error:
Ambiguous reference to table "memory.main.t" (duplicate alias "memory.main.t", explicitly alias one of the
tables using "AS my_alias")
```

Adding the aliases allows the query to parse successfully:

```
SELECT * FROM t AS t1 JOIN t AS t2 USING(x);
```

Shorthands in the `JOIN` Clause

You can specify column names in the `JOIN` clause:

```
CREATE TABLE t1 (x INTEGER);
CREATE TABLE t2 (y INTEGER);
INSERT INTO t1 VALUES (1), (2), (4);
INSERT INTO t2 VALUES (2), (3);
SELECT * FROM t1 NATURAL JOIN t2 t2(x);
```

x
2

You can also use the `VALUES` clause in the `JOIN` clause:

```
SELECT * FROM t1 NATURAL JOIN (VALUES (2), (4)) _(x);
```

x
2
4

Specifying the Join Type Explicitly

> New This feature was introduced in DuckDB 2.0.

The `JOIN BY (TYPE <join_type>)`{:.language-sql .highlight} syntax allows you to explicitly select the join type, including DuckDB's internal join types that have no syntax in standard SQL:

```
SELECT ...
FROM <left_table>
JOIN BY (TYPE <join_type>) <right_table> ON <condition>;
```

The join type name is case-insensitive and it can optionally have a `_join` suffix, e.g., `mark` and `MARK_JOIN` are equivalent. The supported join types are the following:

Join type	Description
`inner`	Inner join.
`left`	Left outer join.
`right`	Right outer join.
`full`	Full outer join.
`semi`	Semi join: returns the rows of the left side that have a match.
`anti`	Anti join: returns the rows of the left side that have no match.
`right_semi`	Returns the rows of the right side that have a match.
`right_anti`	Returns the rows of the right side that have no match.
`mark`	Returns all rows of the left side along with a `BOOLEAN` column `__mark_join_marker`, which has the same three-valued result as an `IN` expression.
`single`	Returns all rows of the left side paired with their matching row from the right side (or `NULL` if there is no match). Similarly to scalar subqueries, it throws an error if a left row has more than one match (see the `scalar_subquery_error_on_multiple_rows` setting).

For example, to run a mark join:

```
CREATE TABLE t1 (a INTEGER);
INSERT INTO t1 VALUES (1), (2), (3), (NULL);
CREATE TABLE t2 (b INTEGER);
INSERT INTO t2 VALUES (2), (3), (NULL);

SELECT a, __mark_join_marker
FROM t1
JOIN BY (TYPE mark) t2 ON (a = b)
ORDER BY a NULLS LAST;
```

a	__mark_join_marker
1	NULL
2	true
3	true
NULL	NULL

FROM-First Syntax

DuckDB's SQL supports the FROM-first syntax, i.e., it allows putting the FROM clause before the SELECT clause or completely omitting the SELECT clause. We use the following example to demonstrate it:

```
CREATE TABLE tbl AS
    SELECT *
    FROM (VALUES ('a'), ('b')) t1(s), range(1, 3) t2(i);
```

FROM-First Syntax with a SELECT Clause

The following statement demonstrates the use of the FROM-first syntax:

```
FROM tbl
SELECT i, s;
```

This is equivalent to:

```
SELECT i, s
FROM tbl;
```

i	s
1	a
2	a
1	b
2	b

FROM-First Syntax without a SELECT Clause

The following statement demonstrates the use of the optional SELECT clause:

```
FROM tbl;
```

This is equivalent to:

```
SELECT *
FROM tbl;
```

s	i
a	1
a	2
b	1
b	2

Syntax

WHERE Clause

The WHERE clause specifies any filters to apply to the data. This allows you to select only a subset of the data in which you are interested. Logically the WHERE clause is applied immediately after the FROM clause.

Examples

Select all rows where the `id` is equal to 3:

```
SELECT *
FROM tbl
WHERE id = 3;
```

Select all rows that match the given **case-sensitive** `LIKE` expression:

```
SELECT *
FROM tbl
WHERE name LIKE '%mark%';
```

Select all rows that match the given **case-insensitive** expression formulated with the `ILIKE` operator:

```
SELECT *
FROM tbl
WHERE name ILIKE '%mark%';
```

Select all rows that match the given composite expression:

```
SELECT *
FROM tbl
WHERE id = 3 OR id = 7;
```

Syntax

GROUP BY Clause

The GROUP BY clause specifies which grouping columns should be used to perform any aggregations in the SELECT clause. If the GROUP BY clause is specified, the query is always an aggregate query, even if no aggregations are present in the SELECT clause.

When a GROUP BY clause is specified, all tuples that have matching data in the grouping columns (i.e., all tuples that belong to the same group) will be combined. The values of the grouping columns themselves are unchanged, and any other columns can be combined using an aggregate function (such as `count`, `sum`, `avg`, etc).

GROUP BY ALL

Use GROUP BY ALL to GROUP BY all columns in the SELECT statement that are not wrapped in aggregate functions. This simplifies the syntax by allowing the columns list to be maintained in a single location, and prevents bugs by keeping the SELECT granularity aligned to the GROUP BY granularity (e.g., it prevents duplication). See examples below and additional examples in the "Friendlier SQL with DuckDB" blog post.

Multiple Dimensions

Normally, the GROUP BY clause groups along a single dimension. Using the GROUPING SETS, CUBE or ROLLUP clauses it is possible to group along multiple dimensions. See the GROUPING SETS page for more information.

Grouping by UNNEST

> New Grouping by UNNEST was introduced in DuckDB 2.0.

UNNEST can be used directly in the GROUP BY clause to group by the elements of a list. This is useful for aggregating over the individual values of a list column:

```
SELECT unnest(tags) AS tag, count(*) AS n
FROM (VALUES (['a', 'b']), (['b', 'c']), (['b'])) t(tags)
GROUP BY unnest(tags)
ORDER BY tag;
```

tag	n
a	1
b	3
c	1

Examples

Count the number of entries in the `addresses` table that belong to each different city:

```
SELECT city, count(*)
FROM addresses
GROUP BY city;
```

Compute the average income per `city` per `street_name`:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY city, street_name;
```

`GROUP BY ALL` Examples

Group by `city` and `street_name` to remove any duplicate values:

```
SELECT city, street_name
FROM addresses
GROUP BY ALL;
```

Compute the average income per city per street_name. Since income is wrapped in an aggregate function, do not include it in the GROUP BY:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY ALL;
-- GROUP BY city, street_name:
```

Syntax

GROUPING SETS

GROUPING SETS, ROLLUP and CUBE can be used in the GROUP BY clause to perform a grouping over multiple dimensions within the same query. Note that this syntax is not compatible with `GROUP BY ALL`.

Examples

Compute the average income along the provided four different dimensions:

```
-- the syntax () denotes the empty set (i.e., computing an ungrouped aggregate)
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY GROUPING SETS ((city, street_name), (city), (street_name), ());
```

Compute the average income along the same dimensions:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY CUBE (city, street_name);
```

Compute the average income along the dimensions `(city, street_name)`, `(city)` and `()`:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY ROLLUP (city, street_name);
```

Description

`GROUPING SETS` performs the same aggregate across different `GROUP BY` clauses in a single query.

```
CREATE TABLE students (course VARCHAR, type VARCHAR);
INSERT INTO students (course, type)
VALUES
    ('CS', 'Bachelor'), ('CS', 'Bachelor'), ('CS', 'PhD'), ('Math', 'Masters'),
    ('CS', NULL), ('CS', NULL), ('Math', NULL);
```

```
SELECT course, type, count(*)
FROM students
GROUP BY GROUPING SETS ((course, type), course, type, ());
```

course	type	count_star()
Math	NULL	1
NULL	NULL	7
CS	PhD	1
CS	Bachelor	2
Math	Masters	1
CS	NULL	2
Math	NULL	2
CS	NULL	5
NULL	NULL	3
NULL	Masters	1
NULL	Bachelor	2
NULL	PhD	1

In the above query, we group across four different sets: `course, type`, `course`, `type` and `()` (the empty group). The result contains `NULL` for a group which is not in the grouping set for the result, i.e., the above query is equivalent to the following statement of `UNION ALL` clauses:

```
-- Group by course, type:
SELECT course, type, count(*)
FROM students
GROUP BY course, type
UNION ALL
-- Group by type:
SELECT NULL AS course, type, count(*)
FROM students
GROUP BY type
UNION ALL
```

```
-- Group by course:
SELECT course, NULL AS type, count(*)
FROM students
GROUP BY course
UNION ALL
-- Group by nothing:
SELECT NULL AS course, NULL AS type, count(*)
FROM students;
```

CUBE and ROLLUP are syntactic sugar to easily produce commonly used grouping sets.

The ROLLUP clause will produce all "sub-groups" of a grouping set, e.g., `ROLLUP (country, city, zip)` produces the grouping sets `(country, city, zip)`, `(country, city)`, `(country)`, `()`. This can be useful for producing different levels of detail of a group by clause. This produces `n+1` grouping sets where n is the amount of terms in the ROLLUP clause.

CUBE produces grouping sets for all combinations of the inputs, e.g., `CUBE (country, city, zip)` will produce `(country, city, zip)`, `(country, city)`, `(country, zip)`, `(city, zip)`, `(country)`, `(city)`, `(zip)`, `()`. This produces `2^n` grouping sets.

Identifying Grouping Sets with `GROUPING_ID()`

The super-aggregate rows generated by `GROUPING SETS`, ROLLUP and CUBE can often be identified by NULL-values returned for the respective column in the grouping. But if the columns used in the grouping can themselves contain actual NULL-values, then it can be challenging to distinguish whether the value in the resultset is a "real" NULL-value coming out of the data itself, or a NULL-value generated by the grouping construct. The `GROUPING_ID()` or `GROUPING()` function is designed to identify which groups generated the super-aggregate rows in the result.

`GROUPING_ID()` is an aggregate function that takes the column expressions that make up the grouping(s). It returns a BIGINT value. The return value is `0` for the rows that are not super-aggregate rows. But for the super-aggregate rows, it returns an integer value that identifies the combination of expressions that make up the group for which the super-aggregate is generated. At this point, an example might help. Consider the following query:

```
WITH days AS (
    SELECT
        year("generate_series")    AS y,
        quarter("generate_series") AS q,
        month("generate_series")   AS m
    FROM generate_series(DATE '2023-01-01', DATE '2023-12-31', INTERVAL 1 DAY)
)
SELECT y, q, m, GROUPING_ID(y, q, m) AS "grouping_id()"
FROM days
GROUP BY GROUPING SETS (
    (y, q, m),
    (y, q),
    (y),
    ()
)
ORDER BY y, q, m;
```

These are the results:

y	q	m	grouping_id()
2023	1	1	0
2023	1	2	0
2023	1	3	0
2023	1	NULL	1
2023	2	4	0

y	q	m	grouping_id()
2023	2	5	0
2023	2	6	0
2023	2	NULL	1
2023	3	7	0
2023	3	8	0
2023	3	9	0
2023	3	NULL	1
2023	4	10	0
2023	4	11	0
2023	4	12	0
2023	4	NULL	1
2023	NULL	NULL	3
NULL	NULL	NULL	7

In this example, the lowest level of grouping is at the month level, defined by the grouping set `(y, q, m)`. Result rows corresponding to that level are simply aggregate rows and the `GROUPING_ID(y, q, m)` function returns `0` for those. The grouping set `(y, q)` results in super-aggregate rows over the month level, leaving a `NULL`-value for the `m` column, and for which `GROUPING_ID(y, q, m)` returns `1`. The grouping set `(y)` results in super-aggregate rows over the quarter level, leaving `NULL`-values for the `m` and `q` column, for which `GROUPING_ID(y, q, m)` returns `3`. Finally, the `()` grouping set results in one super-aggregate row for the entire resultset, leaving `NULL`-values for `y`, `q` and `m` and for which `GROUPING_ID(y, q, m)` returns `7`.

To understand the relationship between the return value and the grouping set, you can think of `GROUPING_ID(y, q, m)` writing to a bitfield, where the first bit corresponds to the last expression passed to `GROUPING_ID()`, the second bit to the one-but-last expression passed to `GROUPING_ID()`, and so on. This may become clearer by casting `GROUPING_ID()` to `BIT`:

```
WITH days AS (
    SELECT
        year("generate_series")    AS y,
        quarter("generate_series") AS q,
        month("generate_series")   AS m
    FROM generate_series(DATE '2023-01-01', DATE '2023-12-31', INTERVAL 1 DAY)
)
SELECT
    y, q, m,
    GROUPING_ID(y, q, m) AS "grouping_id(y, q, m)",
    right(GROUPING_ID(y, q, m)::BIT::VARCHAR, 3) AS "y_q_m_bits"
FROM days
GROUP BY GROUPING SETS (
    (y, q, m),
    (y, q),
    (y),
    ()
)
ORDER BY y, q, m;
```

Which returns these results:

y	q	m	grouping_id(y, q, m)	y_q_m_bits
2023	1	1	0	000
2023	1	2	0	000

y	q	m	grouping_id(y, q, m)	y_q_m_bits
2023	1	3	0	000
2023	1	NULL	1	001
2023	2	4	0	000
2023	2	5	0	000
2023	2	6	0	000
2023	2	NULL	1	001
2023	3	7	0	000
2023	3	8	0	000
2023	3	9	0	000
2023	3	NULL	1	001
2023	4	10	0	000
2023	4	11	0	000
2023	4	12	0	000
2023	4	NULL	1	001
2023	NULL	NULL	3	011
NULL	NULL	NULL	7	111

Note that the number of expressions passed to `GROUPING_ID()`, or the order in which they are passed is independent from the actual group definitions appearing in the `GROUPING SETS`-clause (or the groups implied by `ROLLUP` and `CUBE`). As long as the expressions passed to `GROUPING_ID()` are expressions that appear somewhere in the `GROUPING SETS`-clause, `GROUPING_ID()` will set a bit corresponding to the position of the expression whenever that expression is rolled up to a super-aggregate.

> New As of DuckDB 2.0, calling `GROUPING_ID()` or `GROUPING()` with no arguments defaults to using all columns in the `GROUP BY` clause, in `GROUP BY` order. This is equivalent to listing every grouping column explicitly.

Syntax

HAVING Clause

The `HAVING` clause can be used after the `GROUP BY` clause to provide filter criteria *after* the grouping has been completed. In terms of syntax the `HAVING` clause is identical to the `WHERE` clause, but while the `WHERE` clause occurs before the grouping, the `HAVING` clause occurs after the grouping.

Examples

Count the number of entries in the `addresses` table that belong to each different `city`, filtering out cities with a count below 50:

```
SELECT city, count(*)
FROM addresses
GROUP BY city
HAVING count(*) >= 50;
```

Compute the average income per city per `street_name`, filtering out cities with an average `income` bigger than twice the median `income`:

```sql
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY city, street_name
HAVING avg(income) > 2 * median(income);
```

Syntax

ORDER BY Clause

ORDER BY is an output modifier. Logically it is applied near the very end of the query (just prior to LIMIT or OFFSET, if present). The ORDER BY clause sorts the rows on the sorting criteria in either ascending or descending order. In addition, every order clause can specify whether NULL values should be moved to the beginning or to the end.

The ORDER BY clause may contain one or more expressions, separated by commas. An error will be thrown if no expressions are included, since the ORDER BY clause should be removed in that situation. The expressions may begin with either an arbitrary scalar expression (which could be a column name), a column position number (where the indexing starts from 1), or the keyword ALL. Each expression can optionally be followed by an order modifier (ASC or DESC, default is ASC), and/or a NULL order modifier (NULLS FIRST or NULLS LAST, default is NULLS LAST).

ORDER BY ALL

The ALL keyword indicates that the output should be sorted by every column in order from left to right. The direction of this sort may be modified using either ORDER BY ALL ASC or ORDER BY ALL DESC and/or NULLS FIRST or NULLS LAST. Note that ALL may not be used in combination with other expressions in the ORDER BY clause – it must be by itself. See examples below.

NULL Order Modifier

By default, DuckDB sorts ASC and NULLS LAST, i.e., the values are sorted in ascending order and NULL values are placed last. For the default ASC direction this matches PostgreSQL. Note, however, that DuckDB keeps NULLS LAST even for DESC ordering, whereas PostgreSQL places NULLs first on DESC (it treats NULL as the largest value). To match PostgreSQL in both directions, set `default_null_order = 'NULLS_LAST_ON_ASC_FIRST_ON_DESC'`. The default sort order can be changed with the following configuration options.

Use the `default_null_order` option to change the default NULL sorting order to either NULLS_FIRST, NULLS_LAST, NULLS_FIRST_ON_ASC_LAST_ON_DESC or NULLS_LAST_ON_ASC_FIRST_ON_DESC:

```sql
SET default_null_order = 'NULLS_FIRST';
```

Use the `default_order` option to change the direction of the default sorting order to either DESC or ASC:

```sql
SET default_order = 'DESC';
```

Collations

Text is sorted using the binary comparison collation by default, which means values are sorted on their binary UTF-8 values. While this works well for ASCII text (e.g., for English language data), the sorting order can be incorrect for other languages. For this purpose, DuckDB provides collations. For more information on collations, see the Collation page.

Examples

All examples use this example table:

```
CREATE OR REPLACE TABLE addresses AS
    SELECT '123 Quack Blvd' AS address, 'DuckTown' AS city, '11111' AS zip
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'DuckTown', '11111'
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'Duck Town', '11111'
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'Duck Town', '11111-0001';
```

Select the addresses, ordered by city name using the default NULL order and default order:

```
SELECT *
FROM addresses
ORDER BY city;
```

Select the addresses, ordered by city name in descending order with nulls at the end:

```
SELECT *
FROM addresses
ORDER BY city DESC NULLS LAST;
```

Order by city and then by zip code, both using the default orderings:

```
SELECT *
FROM addresses
ORDER BY city, zip;
```

Order by city using German collation rules:

```
SELECT *
FROM addresses
ORDER BY city COLLATE DE;
```

`ORDER BY ALL` Examples

Order from left to right (by address, then by city, then by zip) in ascending order:

```
SELECT *
FROM addresses
ORDER BY ALL;
```

address	city	zip
111 Duck Duck Goose Ln	Duck Town	11111
111 Duck Duck Goose Ln	Duck Town	11111-0001
111 Duck Duck Goose Ln	DuckTown	11111
123 Quack Blvd	DuckTown	11111

Order from left to right (by address, then by city, then by zip) in descending order:

```
SELECT *
FROM addresses
ORDER BY ALL DESC;
```

address	city	zip
123 Quack Blvd	DuckTown	11111
111 Duck Duck Goose Ln	DuckTown	11111

address	city	zip
111 Duck Duck Goose Ln	Duck Town	11111-0001
111 Duck Duck Goose Ln	Duck Town	11111

Syntax

LIMIT and OFFSET Clauses

`LIMIT` is an output modifier. Logically it is applied at the very end of the query. The `LIMIT` clause restricts the amount of rows fetched. The `OFFSET` clause indicates at which position to start reading the values, i.e., the first `OFFSET` values are ignored.

Note that while `LIMIT` can be used without an `ORDER BY` clause, the results might not be deterministic without the `ORDER BY` clause. This can still be useful, however, for example when you want to inspect a quick snapshot of the data.

Examples

Select the first 5 rows from the addresses table:

```
SELECT *
FROM addresses
LIMIT 5;
```

Select the 5 rows from the addresses table, starting at position 5 (i.e., ignoring the first 5 rows):

```
SELECT *
FROM addresses
LIMIT 5
OFFSET 5;
```

Select the top 5 cities with the highest population:

```
SELECT city, count(*) AS population
FROM addresses
GROUP BY city
ORDER BY population DESC
LIMIT 5;
```

Select 10% of the rows from the addresses table:

```
SELECT *
FROM addresses
LIMIT 10%;
```

FETCH FIRST

> New The SQL-standard `FETCH FIRST` syntax was introduced in DuckDB 2.0.

As an alternative to `LIMIT`, DuckDB supports the SQL-standard `FETCH FIRST ... ROWS ONLY` syntax. It can be combined with `OFFSET`, in which case `FETCH NEXT` reads better:

```
SELECT *
FROM addresses
ORDER BY city
FETCH FIRST 5 ROWS ONLY;
```

```
SELECT *
FROM addresses
ORDER BY city
OFFSET 5 ROWS
FETCH NEXT 10 ROWS ONLY;
```

FIRST and NEXT are interchangeable, as are ROW and ROWS. Omitting the row count fetches a single row:

```
SELECT *
FROM addresses
ORDER BY city
FETCH FIRST ROW ONLY;
```

Syntax

SAMPLE Clause

The SAMPLE clause allows you to run the query on a sample from the base table. This can significantly speed up processing of queries, at the expense of accuracy in the result. Samples can also be used to quickly see a snapshot of the data when exploring a dataset. The sample clause is applied right after anything in the FROM clause (i.e., after any joins, but before the WHERE clause or any aggregates). See the SAMPLE page for more information.

Examples

Select a sample of 1% of the addresses table using default (system) sampling:

```
SELECT *
FROM addresses
USING SAMPLE 1%;
```

Select a sample of 1% of the addresses table using bernoulli sampling:

```
SELECT *
FROM addresses
USING SAMPLE 1% (bernoulli);
```

Select a sample of 10 rows from the subquery:

```
SELECT *
FROM (SELECT * FROM addresses)
USING SAMPLE 10 ROWS;
```

Syntax

Placement in a Query

The statement above that the sample clause is "applied right after anything in the FROM clause" describes the *logical* point at which sampling takes effect, not where the clause is written. Syntactically, the USING SAMPLE clause appears at the end of the SELECT statement body: after the WHERE, GROUP BY, HAVING, and QUALIFY clauses, and before ORDER BY and LIMIT. So it is written after grouping and filtering, but it is *applied* to the result of the FROM clause before those operations run.

To sample an individual table expression rather than the result of the entire FROM clause, use the TABLESAMPLE clause, which is written directly after the table expression it applies to. See Table Samples.

Unnesting

Unnesting is an operation that decomposes values of a composite types into its components. Values of the `LIST` and `STRUCT` type may be unnested using the `unnest()` function.

- Unnesting turns a `LIST`-typed value into a table column: each list-element creates a row, and each element become a column value.
- Unnesting a `STRUCT`-typed value creates a column for each member. The member key becomes the column name, and the member value becomes a column value. Values of a `STRUCT`-type may also be unnested using dot-star (`<struct>.*{:.language-sql .highlight}`) shorthand syntax, but the `unnest()` function offers some additional functionality.
- For a `TUPLE` (unnamed struct), the unnesting operation is the same as for a named `STRUCT`, but in this case the column name is generated based on the (1-based) ordinal position of the member, prefixed by `element`. The dot-star (`<struct>.*{:.language-sql .highlight}`) shorthand syntax is not available for tuples: tuples can only be unnested using the `unnest()` function.

Calling `unnest()`

The `LIST` or `STRUCT` value that is to be unnested is always passed as the first – mandatory – argument to the `unnest()` function. The `unnest()` function has a number of optional additional arguments to control the behavior for recursive unnesting.

The `unnest` function can be called in the `SELECT`-clause as if it is a scalar function. The `unnest` function may not be called in other contexts where one would normally be able to invoke scalar functions, like the `WHERE` or `ORDER BY` clauses. Since DuckDB 2.0, `unnest` can be used in the `GROUP BY` clause. When applied to values of the `LIST`-type, `unnest()` may also appear in clauses where one could normally use a table function.

Unnesting `LIST`-Typed Values

To fully understand unnesting `LIST`-typed values, it is useful to distinguish between the 'input' row that provided the `LIST`-typed value that is to be unnested, and the 'output' row(s) created by the unnesting operation. An individual invocation to `unnest()` on the `LIST`-typed value has the effect of duplicating the 'input' row while populating the field corresponding to the `unnest()` call with the element value. In other words, the original row becomes a repeating group for element unnnested from the `LIST`-typed value. Consequently, if `unnest()` gets called on an empty list (or a `NULL` value), no elements are unnested, and no 'output' rows are generated.

Unnesting Multiple Lists Multiple `LIST`-typed values may be unnested within the same `SELECT`-clause, so for one 'input' row there may be multiple row sets resulting from an `unnest()` invocation, and each may have its own number of rows. Each result becomes a column of the output table, aligning their values by ordinal position, and backfilling columns with `NULL`-values when a particular result has a smaller number of elements than any of the other results. In a final step, the columns of the input row are added to this result. In other words, the repeating group is created only once, and for all `unnest()` results, rather than again for each individual `unnest()` result.

Getting the Element Index Unnesting a value of the `LIST`-type yields only the element values. To also keep track of their indices (the subscripts), you can use the built-in macro `generate_subscripts()`. The `generate_subscripts` macro takes a value of the `LIST`-type as first argument.

`unnest()` as Table Function Since a call to `unnest` on a value of the `LIST`-type yields a rowset, it may also be treated as a table function. This means it may appear in the `FROM`-clause or a `CALL`-statement.

Calls to `unnest()` in a `FROM`-clause or `CALL`-statement do not accept additional parameters, and can thus not be used for recursive unnesting.

Recursive Unnesting

By default, `unnest()` only unpacks the outermost components of the composite-type value. Additional parameters may be passed to allow the unnesting operation to be applied to the unnested member values, and to their unnested values, and so on, recursively. These additional parameters are:

- `recursive`: BOOLEAN, default: `false`. Pass `true` to recursively continue applying `unnest` to unnested member values. When explicitly passing `false`, recursion is disabled. In that case, other additional parameters such as `max_depth` and `keep_parent_names` are effectively ignored
- `max_depth`: UINT32, default: `1`. This controls how many levels of recursion should at most be applied. Values greater than `1` imply recursion, and in such cases `recursive` need not be explicitly passed as `true`.
- `keep_parent_names`: BOOLEAN, default: `false`. Whether to create column names using the keys of all ancestor members. This argument is applicable only when unnesting STRUCT-values.

Note that recursive unnesting always respects the type of the outermost call to `unnest()`:

- If a `LIST`-typed value is passed, then `LIST`-typed elements will be recursively unnested, while `STRUCT`-typed elements are not further unpacked.
- If a `STRUCT`-typed value is passed, then `STRUCT`-typed member values will be recursively unnested, while `LIST`-typed member values are not further unpacked.

Examples

Unnest a list, generating 3 rows (1, 2, 3):

```
SELECT unnest([1, 2, 3]);
```

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': 42, 'b': 84});
```

Recursive unnest of a list of structs:

```
SELECT unnest([{'a': 42, 'b': 84}, {'a': 100, 'b': NULL}], recursive := true);
```

Limit depth of recursive unnest using `max_depth`:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 2);
```

Unnesting Lists Unnest a list, generating 3 rows (1, 2, 3):

```
SELECT unnest([1, 2, 3]);
```

Unnest a list, generating 3 rows ((1, 10), (2, 10), (3, 10)):

```
SELECT unnest([1, 2, 3]), 10;
```

Unnest two lists of different sizes, generating 3 rows ((1, 10), (2, 11), (3, NULL)):

```
SELECT unnest([1, 2, 3]), unnest([10, 11]);
```

Unnest a list column from a subquery:

```
SELECT unnest(l) + 10 FROM (VALUES ([1, 2, 3]), ([4, 5])) tbl(l);
```

Empty result:

```
SELECT unnest([]);
```

Empty result:

```
SELECT unnest(NULL);
```

Using `unnest` on a list emits one row per list entry. Regular scalar expressions in the same `SELECT` clause are repeated for every emitted row. When multiple lists are unnested in the same `SELECT` clause, the lists are unnested side-by-side. If one list is longer than the other, the shorter list is padded with `NULL` values.

Empty and `NULL` lists both unnest to zero rows.

Unnesting Structs Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': 42, 'b': 84});
```

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': 42, 'b': {'x': 84}});
```

`unnest` on a struct will emit one column per entry in the struct.

Recursive Unnest Unnesting a list of lists recursively, generating 5 rows (1, 2, 3, 4, 5):

```
SELECT unnest([[1, 2, 3], [4, 5]], recursive := true);
```

Unnesting a list of structs recursively, generating two rows of two columns (a, b):

```
SELECT unnest([{'a': 42, 'b': 84}, {'a': 100, 'b': NULL}], recursive := true);
```

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': [1, 2, 3], 'b': 88}, recursive := true);
```

Calling `unnest` with the `recursive` setting will fully unnest lists, followed by fully unnesting structs. This can be useful to fully flatten columns that contain lists within lists, or lists of structs. Note that lists *within* structs are not unnested.

Setting the Maximum Depth of Unnesting The `max_depth` parameter allows limiting the maximum depth of recursive unnesting (which is assumed by default and does not have to be specified separately). For example, unnesting to `max_depth` of 2 yields the following:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 2) AS x;
```

x
[1, 2]
[3, 4]
[5, 6]
[7, 8, 9]
[]
[10, 11]

Meanwhile, unnesting to `max_depth` of 3 results in:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 3) AS x;
```

x
1
2
3
4
5
6
7

x
8
9
10
11

Keeping Track of List Entry Positions To keep track of each entry's position within the original list, `unnest` may be combined with `generate_subscripts`:

```
SELECT unnest(l) AS x, generate_subscripts(l, 1) AS index
FROM (VALUES ([1, 2, 3]), ([4, 5])) tbl(l);
```

x	index
1	1
2	2
3	3
4	1
5	2

Keep Column Names When Recursively Unnesting The `keep_parent_names` parameter can be used to retain the parent column names when recursively unnesting a named struct. For example, unnesting the following query with `keep_parent_names` enabled:

```
SELECT unnest([{'a': 0, 'b': {'bb': {'bbb': 1}}}], recursive := true, keep_parent_names := true);
```

yields the following result:

a	b.bb.bbb
0	1

In this case, the field names are preserved, showing the path to the innermost value. This is particularly useful when working with complex nested data structures, as it maintains the structure and naming convention of the original data. The parameter can also be used in conjunction with the `max_depth` parameter, allowing more control and enabling more precise management of nested structures.

WITH Clause

The `WITH` clause allows you to specify common table expressions (CTEs). Regular (non-recursive) common-table-expressions are essentially views that are limited in scope to a particular query. CTEs can reference each other and can be nested. Recursive CTEs can reference themselves.

Basic CTE Examples

Create a CTE called `cte` and use it in the main query:

```
WITH cte AS (SELECT 42 AS x)
SELECT * FROM cte;
```

x
42

Create two CTEs `cte1` and `cte2`, where the second CTE references the first CTE:

```
WITH
    cte1 AS (SELECT 42 AS i),
    cte2 AS (SELECT i * 100 AS x FROM cte1)
SELECT * FROM cte2;
```

x
4200

You can specify column names for CTEs:

```
WITH cte(j) AS (SELECT 42 AS i)
FROM cte;
```

CTE Materialization

DuckDB handles CTEs as *materialized* by default, meaning that the CTE is evaluated once and the result is stored in a temporary table. However, under certain conditions, DuckDB can *inline* the CTE into the main query, which means that the CTE is not materialized and its definition is duplicated in each place it is referenced. Inlining is done using the following heuristics:

- The CTE is not referenced more than once.
- The CTE does not contain a `VOLATILE` function.
- The CTE is using `AS NOT MATERIALIZED` and does not use `AS MATERIALIZED`.
- The CTE does not perform a grouped aggregation.

Materialization can be explicitly activated by defining the CTE using `AS MATERIALIZED` and disabled by using `AS NOT MATERIALIZED`. Note that inlining is not always possible, even if the heuristics are met. For example, if the CTE contains a `read_csv` function, it cannot be inlined.

Take the following query for example, which invokes the same CTE three times:

```
WITH t(x) AS (<complex_query>)
SELECT *
FROM
    t AS t1,
    t AS t2,
    t AS t3;
```

Inlining duplicates the definition of `t` for each reference which results in the following query:

```
SELECT *
FROM
    (<complex_query>) AS t1(x),
    (<complex_query>) AS t2(x),
    (<complex_query>) AS t3(x);
```

If `complex_query` is expensive, materializing it with the `MATERIALIZED` keyword can improve performance. In this case, `complex_query` is evaluated only once.

```
WITH t(x) AS MATERIALIZED (<complex_query>)
SELECT *
FROM
    t AS t1,
    t AS t2,
    t AS t3;
```

If one wants to disable materialization, use `NOT MATERIALIZED`:

```
WITH t(x) AS NOT MATERIALIZED (<complex_query>)
SELECT *
FROM
    t AS t1,
    t AS t2,
    t AS t3;
```

Generally, it is not recommended to use explicit materialization hints, as DuckDB's query optimizer is capable of deciding when to materialize or inline a CTE based on the query structure and the heuristics mentioned above. However, in some cases, it may be beneficial to use `MATERIALIZED` or `NOT MATERIALIZED` to control the behavior explicitly.

Data-Modifying CTEs

> New Support for data-modifying statements in CTEs was introduced in DuckDB 2.0.

A CTE body can be a data-modifying statement: `INSERT`, `UPDATE`, `DELETE`, or `COPY`. This lets you chain modifications into a single pipeline. A data-modifying CTE that is followed by other statements should use a `RETURNING` clause so that its result can be referenced.

Move rows from one table to another in a single statement by combining a `DELETE` and an `INSERT`:

```
WITH moved AS MATERIALIZED (
    DELETE FROM staging
    RETURNING *
)
INSERT INTO archive
SELECT * FROM moved;
```

Count the rows affected by an `UPDATE`:

```
WITH updated AS (
    UPDATE orders
    SET status = 'shipped'
    WHERE status = 'ready'
    RETURNING *
)
SELECT count(*) AS shipped FROM updated;
```

> **Best practice.** Mark a data-modifying CTE as `MATERIALIZED` when later parts of the query depend on the modification having already run, so that it is not inlined or executed more than once.

Recursive CTEs

`WITH RECURSIVE` allows the definition of CTEs which can refer to themselves. Note that the query must be formulated in a way that ensures termination, otherwise, it may run into an infinite loop.

Example: Fibonacci Sequence

`WITH RECURSIVE` can be used to make recursive calculations. For example, here is how `WITH RECURSIVE` could be used to calculate the first ten Fibonacci numbers:

```
WITH RECURSIVE FibonacciNumbers (
    RecursionDepth, FibonacciNumber, NextNumber
) AS (
        -- Base case
        SELECT
            0 AS RecursionDepth,
            0 AS FibonacciNumber,
            1 AS NextNumber
        UNION ALL
        -- Recursive step
        SELECT
            fib.RecursionDepth + 1 AS RecursionDepth,
            fib.NextNumber AS FibonacciNumber,
            fib.FibonacciNumber + fib.NextNumber AS NextNumber
        FROM
            FibonacciNumbers fib
        WHERE
            fib.RecursionDepth + 1 < 10
    )
SELECT
    fn.RecursionDepth AS FibonacciNumberIndex,
    fn.FibonacciNumber
FROM
    FibonacciNumbers fn;
```

FibonacciNumberIndex	FibonacciNumber
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

Example: Tree Traversal

`WITH RECURSIVE` can be used to traverse trees. For example, take a hierarchy of tags:

```
CREATE TABLE tag (id INTEGER, name VARCHAR, subclassof INTEGER);
INSERT INTO tag VALUES
    (1, 'U2',     5),
    (2, 'Blur',   5),
    (3, 'Oasis',  5),
    (4, '2Pac',   6),
    (5, 'Rock',   7),
    (6, 'Rap',    7),
    (7, 'Music',  9),
    (8, 'Movies', 9),
    (9, 'Art', NULL);
```

The following query returns the path from the node `Oasis` to the root of the tree (`Art`).

```
WITH RECURSIVE tag_hierarchy(id, source, path) AS (
```

```
        SELECT id, name, [name] AS path
        FROM tag
        WHERE subclassof IS NULL
    UNION ALL
        SELECT tag.id, tag.name, list_prepend(tag.name, tag_hierarchy.path)
        FROM tag, tag_hierarchy
        WHERE tag.subclassof = tag_hierarchy.id
    )
SELECT path
FROM tag_hierarchy
WHERE source = 'Oasis';
```

path
[Oasis, Rock, Music, Art]

Graph Traversal

The `WITH RECURSIVE` clause can be used to express graph traversal on arbitrary graphs. However, if the graph has cycles, the query must perform cycle detection to prevent infinite loops. One way to achieve this is to store the path of a traversal in a list and, before extending the path with a new edge, check whether its endpoint has been visited before (see the example later).

Take the following directed graph from the LDBC Graphalytics benchmark:

```
CREATE TABLE edge (node1id INTEGER, node2id INTEGER);
INSERT INTO edge VALUES
    (1, 3), (1, 5), (2, 4), (2, 5), (2, 10), (3, 1),
    (3, 5), (3, 8), (3, 10), (5, 3), (5, 4), (5, 8),
    (6, 3), (6, 4), (7, 4), (8, 1), (9, 4);
```

Note that the graph contains directed cycles, e.g., between nodes 1, 5 and 8.

Enumerate All Paths from a Node The following query returns **all paths** starting in node 1:

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
        SELECT -- Define the path as the first edge of the traversal
            node1id AS startNode,
            node2id AS endNode,
            [node1id, node2id] AS path
        FROM edge
        WHERE startNode = 1
        UNION ALL
        SELECT -- Concatenate new edge to the path
            paths.startNode AS startNode,
            node2id AS endNode,
            array_append(path, node2id) AS path
        FROM paths
        JOIN edge ON paths.endNode = node1id
        -- Prevent adding a repeated node to the path.
        -- This ensures that no cycles occur.
        WHERE list_position(paths.path, node2id) IS NULL
    )
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

startNode	endNode	path
1	3	[1, 3]

startNode	endNode	path
1	5	[1, 5]
1	5	[1, 3, 5]
1	8	[1, 3, 8]
1	10	[1, 3, 10]
1	3	[1, 5, 3]
1	4	[1, 5, 4]
1	8	[1, 5, 8]
1	4	[1, 3, 5, 4]
1	8	[1, 3, 5, 8]
1	8	[1, 5, 3, 8]
1	10	[1, 5, 3, 10]

Note that the result of this query is not restricted to shortest paths, e.g., for node 5, the results include paths `[1, 5]` and `[1, 3, 5]`.

Enumerate Unweighted Shortest Paths from a Node In most cases, enumerating all paths is not practical or feasible. Instead, only the **(unweighted) shortest paths** are of interest. To find these, the second half of the `WITH RECURSIVE` query should be adjusted such that it only includes a node if it has not yet been visited. This is implemented by using a subquery that checks if any of the previous paths includes the node:

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
        SELECT -- Define the path as the first edge of the traversal
            node1id AS startNode,
            node2id AS endNode,
            [node1id, node2id] AS path
        FROM edge
        WHERE startNode = 1
        UNION ALL
        SELECT -- Concatenate new edge to the path
            paths.startNode AS startNode,
            node2id AS endNode,
            array_append(path, node2id) AS path
        FROM paths
        JOIN edge ON paths.endNode = node1id
        -- Prevent adding a node that was visited previously by any path.
        -- This ensures that (1) no cycles occur and (2) only nodes that
        -- were not visited by previous (shorter) paths are added to a path.
        WHERE NOT EXISTS (
                FROM paths previous_paths
                WHERE list_contains(previous_paths.path, node2id)
              )
    )
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

startNode	endNode	path
1	3	[1, 3]
1	5	[1, 5]
1	8	[1, 3, 8]

startNode	endNode	path
1	10	[1, 3, 10]
1	4	[1, 5, 4]
1	8	[1, 5, 8]

Enumerate Unweighted Shortest Paths between Two Nodes `WITH RECURSIVE` can also be used to find **all (unweighted) shortest paths between two nodes**. To ensure that the recursive query is stopped as soon as we reach the end node, we use a window function which checks whether the end node is among the newly added nodes.

The following query returns all unweighted shortest paths between nodes 1 (start node) and 8 (end node):

```
WITH RECURSIVE paths(startNode, endNode, path, endReached) AS (
   SELECT -- Define the path as the first edge of the traversal
        node1id AS startNode,
        node2id AS endNode,
        [node1id, node2id] AS path,
        (node2id = 8) AS endReached
     FROM edge
     WHERE startNode = 1
   UNION ALL
   SELECT -- Concatenate new edge to the path
        paths.startNode AS startNode,
        node2id AS endNode,
        array_append(path, node2id) AS path,
        max(CASE WHEN node2id = 8 THEN 1 ELSE 0 END)
            OVER (ROWS BETWEEN UNBOUNDED PRECEDING
                           AND UNBOUNDED FOLLOWING) AS endReached
     FROM paths
     JOIN edge ON paths.endNode = node1id
    WHERE NOT EXISTS (
            FROM paths previous_paths
            WHERE list_contains(previous_paths.path, node2id)
          )
      AND paths.endReached = 0
)
SELECT startNode, endNode, path
FROM paths
WHERE endNode = 8
ORDER BY length(path), path;
```

startNode	endNode	path
1	8	[1, 3, 8]
1	8	[1, 5, 8]

Accessing the Union Table with `recurring`

Within the recursive term of a `WITH RECURSIVE` CTE, the CTE name (e.g., `counter`) refers to the rows produced by the *last iteration*. To access *all rows accumulated so far* (the union table), use the `recurring` schema prefix:

```
WITH RECURSIVE counter(i) AS (
    SELECT 1
        UNION ALL
    SELECT i + 1
    FROM counter
    WHERE (SELECT max(i) FROM recurring.counter) < 5
```

```
)
SELECT *
FROM counter;
```

i
1
2
3
4
5

Here, `recurring.counter` gives access to all rows accumulated across all previous iterations, while `counter` in the `FROM` clause only contains the rows from the most recent iteration. This is useful when termination conditions or calculations depend on the full accumulated result rather than just the previous iteration.

Recursive CTEs with `USING KEY`

> **Update.** DuckDB 1.5.0 deprecated recursive `UNION`s in `USING KEY` CTEs because they behaved identically to `UNION ALL`. DuckDB 2.0.0 gives the two keywords distinct meanings and supports both: `UNION ALL` forwards every candidate row to the next iteration, while `UNION` only forwards new or changed keys. See `UNION` vs. `UNION ALL` in `USING KEY` CTEs. The `deprecated_using_key_syntax` setting was removed.

`USING KEY` alters the behavior of a regular recursive CTE.

In each iteration, a regular recursive CTE appends result rows to the union table, which ultimately defines the overall result of the CTE. In contrast, a CTE with `USING KEY` has the ability to update rows that have been placed in the union table in an earlier iteration: if the current iteration produces a row with key k, it replaces a row with the same key k in the union table (like a dictionary). If no such row exists in the union table yet, the new row is appended to the union table as usual.

This allows a CTE to exercise fine-grained control over the union table contents. Avoiding the append-only behavior can lead to significantly smaller union table sizes. This helps query runtime, memory consumption, and makes it feasible to access the union table while the iteration is still ongoing. In a CTE `WITH RECURSIVE T(...) USING KEY ...`, table `T` denotes the rows added by the last iteration (as is usual for recursive CTEs), while table `recurring.T` denotes the union table built so far. References to `recurring.T` allow for the elegant and idiomatic translation of rather complex algorithms into readable SQL code.

Example: `USING KEY`

This is a recursive CTE where `USING KEY` has a key column (a) and a payload column (b). The payload columns correspond to the columns to be overwritten. In the first iteration we have two different keys, 1 and 2. These two keys will generate two new rows, `(1, 3)` and `(2, 4)`. In the next iteration we produce a new key, 3, which generates a new row. We also generate the row `(2, 3)`, where 2 is a key that already exists from the previous iteration. This will overwrite the old payload 4 with the new payload 3.

```
WITH RECURSIVE tbl(a, b) USING KEY (a) AS (
    SELECT a, b
    FROM (VALUES (1, 3), (2, 4)) t(a, b)
        UNION ALL
    SELECT a + 1, b
    FROM tbl
    WHERE a < 3
)
SELECT *
FROM tbl;
```

a	b
1	3
2	3
3	3

Aggregate Payloads in `USING KEY`

By default, each payload column (i.e., each column not listed in `USING KEY`) uses `last` semantics: a new row for an existing key overwrites the previous payload value. You can also specify an aggregate function for a payload column in the `USING KEY` clause. In this case, all rows produced for a key are combined using the aggregate function. Payload columns without an aggregate keep the default `last` semantics.

For example, the following query computes the average of all values produced for key `1`:

```
WITH RECURSIVE tbl(a, b) USING KEY (a, avg(b)) AS (
    SELECT 1, 5
        UNION ALL
    SELECT a, b - 1
    FROM tbl
    WHERE b > 0
)
TABLE tbl;
```

a	b
1	2.5

The following query maintains the maximum of `b`, while `c` uses the default `last` semantics:

```
WITH RECURSIVE tbl(a, b, c) USING KEY (a, max(b)) AS (
    SELECT 1, 5, 'x'
        UNION ALL
    SELECT a, b - 1, 'y'
    FROM tbl
    WHERE b > 0
)
TABLE tbl;
```

a	b	c
1	5	y

`UNION` vs. `UNION ALL` in `USING KEY` CTEs

Both `UNION` and `UNION ALL` update the union table in the same way: all rows produced by an iteration (the candidates) are applied to the union table, i.e., they insert new keys, overwrite payloads or are combined by the payload aggregates. The two forms differ in which rows are passed to the next iteration:

- With `UNION ALL`, every candidate row produced by the recursive term is forwarded to the next iteration, even if it did not change the union table.
- With `UNION`, only rows for keys that are new or whose (finalized) payload changed are forwarded to the next iteration. Therefore, the recursion stops once the union table stops changing.

For example, the following query computes single-source shortest paths in a weighted graph that contains a cycle. Using `min(dist)` as the payload aggregate, the union table keeps the shortest distance found so far for each node. With `UNION`, the recursion terminates once no shorter distances are found. With `UNION ALL`, the query would not terminate, as candidates would continue to be forwarded around the cycle.

```
CREATE TABLE weighted_edges (src INTEGER, dst INTEGER, weight INTEGER);
INSERT INTO weighted_edges VALUES
    (1, 2, 4), (1, 3, 1), (3, 2, 1), (2, 4, 1), (3, 4, 5), (4, 1, 1);

WITH RECURSIVE shortest_paths(node, dist) USING KEY (node, min(dist)) AS (
    SELECT 1, 0
        UNION
    SELECT e.dst, sp.dist + e.weight
    FROM shortest_paths AS sp
    JOIN weighted_edges AS e ON e.src = sp.node
)
SELECT *
FROM shortest_paths
ORDER BY node;
```

node	dist
1	0
2	2
3	1
4	3

For more details on the semantics and the implementation, see the "How DuckDB Runs Recursive CTEs Faster" blog post.

Using VALUES

You can use the VALUES clause for the initial (anchor) part of the CTE:

```
WITH RECURSIVE tbl(a, b) USING KEY (a) AS (
    VALUES (1, 3), (2, 4)
        UNION ALL
    SELECT a + 1, b
    FROM tbl
    WHERE a < 3
)
SELECT *
FROM tbl;
```

Example: `USING KEY` References Union Table

As well as using the union table as a dictionary, we can now reference it in queries. This allows you to use results from not just the previous iteration, but also earlier ones. This new feature makes certain algorithms easier to implement.

One example is the connected components algorithm. For each node, the algorithm determines the node with the lowest ID to which it is connected. To achieve this, we use the entries in the union table to track the lowest ID found for a node. If a new incoming row contains a lower ID, we update this value.

```
CREATE TABLE nodes (id INTEGER);
INSERT INTO nodes VALUES (1), (2), (3), (4), (5), (6), (7), (8);
CREATE TABLE edges (node1id INTEGER, node2id INTEGER);
INSERT INTO edges VALUES
    (1, 3), (2, 3), (3, 7), (7, 8), (5, 4), (6, 4);
```

```
WITH RECURSIVE connected_components(id, comp) USING KEY (id) AS (
    SELECT n.id, n.id AS comp
    FROM nodes AS n
        UNION ALL (
    SELECT DISTINCT ON (previous_iter.id) previous_iter.id, initial_iter.comp
    FROM
        recurring.connected_components AS previous_iter,
        connected_components AS initial_iter,
        edges AS e
    WHERE ((e.node1id, e.node2id) = (previous_iter.id, initial_iter.id)
       OR (e.node2id, e.node1id) = (previous_iter.id, initial_iter.id))
      AND initial_iter.comp < previous_iter.comp
    ORDER BY initial_iter.id ASC, previous_iter.comp ASC)
)
TABLE connected_components
ORDER BY id;
```

id	comp
1	1
2	1
3	1
4	4
5	4
6	4
7	1
8	1

Limitations

DuckDB does not support mutually recursive CTEs. See the related issue and discussion in the DuckDB repository.

Syntax

WINDOW Clause

The `WINDOW` clause allows you to specify named windows that can be used within window functions. These are useful when you have multiple window functions, as they allow you to avoid repeating the same window clause.

Syntax

QUALIFY Clause

The `QUALIFY` clause is used to filter the results of `WINDOW` functions. This filtering of results is similar to how a `HAVING` clause filters the results of aggregate functions applied based on the `GROUP BY` clause.

The `QUALIFY` clause avoids the need for a subquery or `WITH` clause to perform this filtering (much like `HAVING` avoids a subquery). An example using a `WITH` clause instead of `QUALIFY` is included below the `QUALIFY` examples.

Note that this is filtering based on `WINDOW` functions, not necessarily based on the `WINDOW` clause. The `WINDOW` clause is optional and can be used to simplify the creation of multiple `WINDOW` function expressions.

The position of where to specify a QUALIFY clause is following the WINDOW clause in a SELECT statement (WINDOW does not need to be specified), and before the ORDER BY.

Examples

Each of the following examples produce the same output, located below.

Filter based on a window function defined in the QUALIFY clause:

```
SELECT
    schema_name,
    function_name,
    -- In this example the function_rank column in the select clause is for reference
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
FROM duckdb_functions()
QUALIFY
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) < 3;
```

Filter based on a window function defined in the SELECT clause:

```
SELECT
    schema_name,
    function_name,
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
FROM duckdb_functions()
QUALIFY
    function_rank < 3;
```

Filter based on a window function defined in the QUALIFY clause, but using the WINDOW clause:

```
SELECT
    schema_name,
    function_name,
    -- In this example the function_rank column in the select clause is for reference
    row_number() OVER my_window AS function_rank
FROM duckdb_functions()
WINDOW
    my_window AS (PARTITION BY schema_name ORDER BY function_name)
QUALIFY
    row_number() OVER my_window < 3;
```

Filter based on a window function defined in the SELECT clause, but using the WINDOW clause:

```
SELECT
    schema_name,
    function_name,
    row_number() OVER my_window AS function_rank
FROM duckdb_functions()
WINDOW
    my_window AS (PARTITION BY schema_name ORDER BY function_name)
QUALIFY
    function_rank < 3;
```

Equivalent query based on a WITH clause (without a QUALIFY clause):

```
WITH ranked_functions AS (
    SELECT
        schema_name,
        function_name,
        row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
    FROM duckdb_functions()
)
SELECT
    *
FROM ranked_functions
```

```
WHERE
    function_rank < 3;
```

schema_name	function_name	function_rank
main	!=	1
main	!__postfix	2
pg_catalog	col_description	1
pg_catalog	current_database	2

Syntax

VALUES Clause

The `VALUES` clause is used to specify a fixed number of rows. The `VALUES` clause can be used as a stand-alone statement, as part of the `FROM` clause, or as input to an `INSERT INTO` statement.

Examples

Generate two rows and directly return them:

```
VALUES ('Amsterdam', 1), ('London', 2);
```

Generate two rows as part of a `FROM` clause, and rename the columns:

```
SELECT *
FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
```

Generate two rows and insert them into a table:

```
INSERT INTO cities
VALUES ('Amsterdam', 1), ('London', 2);
```

Create a table directly from a `VALUES` clause:

```
CREATE TABLE cities AS
    SELECT *
    FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
```

Syntax

FILTER Clause

The `FILTER` clause may optionally follow an aggregate function in a `SELECT` statement. This will filter the rows of data that are fed into the aggregate function in the same way that a `WHERE` clause filters rows, but localized to the specific aggregate function.

There are multiple types of situations where this is useful, including when evaluating multiple aggregates with different filters, and when creating a pivoted view of a dataset. `FILTER` provides a cleaner syntax for pivoting data when compared with the more traditional `CASE WHEN` approach discussed below.

Some aggregate functions also do not filter out `NULL` values, so using a `FILTER` clause will return valid results when at times the `CASE WHEN` approach will not. This occurs with the functions `first` and `last`, which are desirable in a non-aggregating pivot operation where the goal is to simply re-orient the data into columns rather than re-aggregate it. For `list` and `array_agg`, `FILTER` excludes rows that do not satisfy its condition, whereas a `CASE WHEN` expression can contribute `NULL` placeholders for those rows. However, `FILTER` does **not** automatically remove `NULL` values from rows that do satisfy the condition. To omit them, include an explicit condition on the aggregated value, such as `list(x) FILTER (WHERE x IS NOT NULL)`.

Examples

Filtering NULL Values from Lists

A `FILTER` condition acts on rows, not on the values stored in the resulting list. Here the first expression keeps the `NULL` value because `y` is `true` for that row, while the second explicitly excludes null values from `x`:

```
SELECT
    list(x) FILTER (WHERE y) AS selected_rows,
    list(x) FILTER (WHERE x IS NOT NULL) AS non_null_values
FROM (VALUES (NULL::INTEGER, true), (2, true), (1, false)) t(x, y);
```

The results are `[NULL, 2]` for `selected_rows` and `[2, 1]` for `non_null_values`.

Return the following:

- The total number of rows
- The number of rows where `i <= 5`
- The number of rows where `i` is odd

```
SELECT
    count() AS total_rows,
    count() FILTER (i <= 5) AS lte_five,
    count() FILTER (i % 2 = 1) AS odds
FROM generate_series(1, 10) tbl(i);
```

total_rows	lte_five	odds
10	5	5

> Simply counting rows that satisfy a condition can also be achieved without the `FILTER` clause, using the boolean `sum` aggregate function, e.g., `sum(i <= 5)`.

Different aggregate functions may be used, and multiple `WHERE` expressions are also permitted:

```
SELECT
    sum(i) FILTER (i <= 5) AS lte_five_sum,
    median(i) FILTER (i % 2 = 1) AS odds_median,
    median(i) FILTER (i % 2 = 1 AND i <= 5) AS odds_lte_five_median
FROM generate_series(1, 10) tbl(i);
```

lte_five_sum	odds_median	odds_lte_five_median
15	5.0	3.0

The `FILTER` clause can also be used to pivot data from rows into columns. This is a static pivot, as columns must be defined prior to runtime in SQL. However, this kind of statement can be dynamically generated in a host programming language to leverage DuckDB's SQL engine for rapid, larger than memory pivoting.

First generate an example dataset:

```
CREATE TEMP TABLE stacked_data AS
    SELECT
        i,
        CASE WHEN i <= rows * 0.25  THEN 2022
             WHEN i <= rows * 0.5   THEN 2023
             WHEN i <= rows * 0.75  THEN 2024
             WHEN i <= rows * 0.875 THEN 2025
             ELSE NULL
```

```
            END AS year
    FROM (
        SELECT
            i,
            count(*) OVER () AS rows
        FROM generate_series(1, 100_000_000) tbl(i)
    ) tbl;
```

"Pivot" the data out by year (move each year out to a separate column):

```
SELECT
    count(i) FILTER (year = 2022) AS "2022",
    count(i) FILTER (year = 2023) AS "2023",
    count(i) FILTER (year = 2024) AS "2024",
    count(i) FILTER (year = 2025) AS "2025",
    count(i) FILTER (year IS NULL) AS "NULLs"
FROM stacked_data;
```

This syntax produces the same results as the `FILTER` clauses above:

```
SELECT
    count(CASE WHEN year = 2022 THEN i END) AS "2022",
    count(CASE WHEN year = 2023 THEN i END) AS "2023",
    count(CASE WHEN year = 2024 THEN i END) AS "2024",
    count(CASE WHEN year = 2025 THEN i END) AS "2025",
    count(CASE WHEN year IS NULL THEN i END) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
25000000	25000000	25000000	12500000	12500000

However, the `CASE WHEN` approach will not work as expected when using an aggregate function that does not ignore `NULL` values. The `first` function falls into this category, so `FILTER` is preferred in this case.

"Pivot" the data out by year (move each year out to a separate column):

```
SELECT
    first(i) FILTER (year = 2022) AS "2022",
    first(i) FILTER (year = 2023) AS "2023",
    first(i) FILTER (year = 2024) AS "2024",
    first(i) FILTER (year = 2025) AS "2025",
    first(i) FILTER (year IS NULL) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
1474561	25804801	50749441	76431361	87500001

This will produce `NULL` values whenever the first evaluation of the `CASE WHEN` clause returns a `NULL`:

```
SELECT
    first(CASE WHEN year = 2022 THEN i END) AS "2022",
    first(CASE WHEN year = 2023 THEN i END) AS "2023",
    first(CASE WHEN year = 2024 THEN i END) AS "2024",
    first(CASE WHEN year = 2025 THEN i END) AS "2025",
    first(CASE WHEN year IS NULL THEN i END) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
1228801	NULL	NULL	NULL	NULL

Aggregate Function Syntax (Including `FILTER` Clause)

Set Operations

Set operations allow queries to be combined according to set operation semantics. Set operations refer to the `UNION [ALL]`, `INTERSECT [ALL]` and `EXCEPT [ALL]` clauses. The vanilla variants use set semantics, i.e., they eliminate duplicates, while the variants with `ALL` use bag semantics.

Traditional set operations unify queries **by column position**, and require the to-be-combined queries to have the same number of input columns. If the columns are not of the same type, casts may be added. The result will use the column names from the first query.

DuckDB also supports `UNION [ALL] BY NAME`, which joins columns by name instead of by position. `UNION BY NAME` does not require the inputs to have the same number of columns. `NULL` values will be added in case of missing columns.

`UNION`

The `UNION` clause can be used to combine rows from multiple queries. The queries are required to return the same number of columns. Implicit casting to one of the returned types is performed to combine columns of different types where necessary. If this is not possible, the `UNION` clause throws an error.

Vanilla `UNION` (Set Semantics)

The vanilla `UNION` clause follows set semantics, therefore it performs duplicate elimination, i.e., only unique rows will be included in the result.

```
SELECT * FROM range(2) t1(x)
UNION
SELECT * FROM range(3) t2(x);
```

x
2
1
0

`UNION ALL` (Bag Semantics)

`UNION ALL` returns all rows of both queries following bag semantics, i.e., *without* duplicate elimination.

```
SELECT * FROM range(2) t1(x)
UNION ALL
SELECT * FROM range(3) t2(x);
```

x
0

x
1
0
1
2

`UNION [ALL] BY NAME`

The `UNION [ALL] BY NAME` clause can be used to combine rows from different tables by name, instead of by position. `UNION BY NAME` does not require both queries to have the same number of columns. Any columns that are only found in one of the queries are filled with `NULL` values for the other query.

Take the following tables for example:

```
CREATE TABLE capitals (city VARCHAR, country VARCHAR);
INSERT INTO capitals VALUES
    ('Amsterdam', 'NL'),
    ('Berlin', 'Germany');
CREATE TABLE weather (city VARCHAR, degrees INTEGER, date DATE);
INSERT INTO weather VALUES
    ('Amsterdam', 10, '2022-10-14'),
    ('Seattle', 8, '2022-10-12');
```

```
SELECT * FROM capitals
UNION BY NAME
SELECT * FROM weather;
```

city	country	degrees	date
Seattle	NULL	8	2022-10-12
Amsterdam	NL	NULL	NULL
Berlin	Germany	NULL	NULL
Amsterdam	NULL	10	2022-10-14

`UNION BY NAME` follows set semantics (therefore it performs duplicate elimination), whereas `UNION ALL BY NAME` follows bag semantics.

`INTERSECT`

The `INTERSECT` clause can be used to select all rows that occur in the result of **both** queries.

Vanilla `INTERSECT` (Set Semantics)

Vanilla `INTERSECT` performs duplicate elimination, so only unique rows are returned.

```
SELECT * FROM range(2) t1(x)
INTERSECT
SELECT * FROM range(6) t2(x);
```

x
0
1

`INTERSECT ALL` (Bag Semantics)

`INTERSECT ALL` follows bag semantics, so duplicates are returned.

```
SELECT unnest([5, 5, 6, 6, 6, 6, 7, 8]) AS x
INTERSECT ALL
SELECT unnest([5, 6, 6, 7, 7, 9]);
```

x
5
6
6
7

EXCEPT

The EXCEPT clause can be used to select all rows that **only** occur in the left query.

Vanilla EXCEPT (Set Semantics)

Vanilla EXCEPT follows set semantics, therefore, it performs duplicate elimination, so only unique rows are returned.

```
SELECT * FROM range(5) t1(x)
EXCEPT
SELECT * FROM range(2) t2(x);
```

x
2
3
4

`EXCEPT ALL` (Bag Semantics)

`EXCEPT ALL` uses bag semantics:

```
SELECT unnest([5, 5, 6, 6, 6, 6, 7, 8]) AS x
EXCEPT ALL
SELECT unnest([5, 6, 6, 7, 7, 9]);
```

x
5

x
8
6
6

Syntax

Prepared Statements

DuckDB supports prepared statements where parameters are substituted when the query is executed. This can improve readability and is useful for preventing SQL injections.

Syntax

There are three syntaxes for denoting parameters in prepared statements: auto-incremented (`?`), positional (`$1`), and named (`$param`). Note that not all clients support all of these syntaxes, e.g., the JDBC client only supports auto-incremented parameters in prepared statements.

Example Dataset

In the following, we introduce the three different syntaxes and illustrate them with examples using the following table.

```
CREATE TABLE person (name VARCHAR, age BIGINT);
INSERT INTO person VALUES ('Alice', 37), ('Ana', 35), ('Bob', 41), ('Bea', 25);
```

In our example query, we'll look for people whose name starts with a B and are at least 40 years old. This will return a single row `<'Bob', 41>`.

Auto-Incremented Parameters: ?

DuckDB supports using prepared statements with auto-incremented indexing, i.e., the position of the parameters in the query corresponds to their position in the execution statement. For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, ?)
      AND age >= ?;
```

Using the CLI client, the statement is executed as follows.

```
EXECUTE query_person('B', 40);
```

Positional Parameters: $1

Prepared statements can use positional parameters, where parameters are denoted with an integer (`$1`, `$2`). For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $2)
      AND age >= $1;
```

Using the CLI client, the statement is executed as follows. Note that the first parameter corresponds to `$1`, the second to `$2`, and so on.

```
EXECUTE query_person(40, 'B');
```

Named Parameters: `$parameter`

DuckDB also supports named parameters where parameters are denoted with `$parameter_name`. For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $name_start_letter)
      AND age >= $minimum_age;
```

Using the CLI client, the statement is executed as follows.

```
EXECUTE query_person(name_start_letter := 'B', minimum_age := 40);
```

Session Variables as Default Values

> New Using session variables as default values for named parameters was introduced in DuckDB 2.0.

When a named parameter is not supplied at execution time, DuckDB fills it from the session variable of the same name, if one is set. An explicit argument always takes precedence over the variable.

```
SET VARIABLE multiplier = 10;
SET VARIABLE x = 42;
PREPARE compute AS SELECT $multiplier * $x AS result;
```

With no arguments, both parameters come from the variables, computing `10 * 42`:

```
EXECUTE compute;
```

result
420

Supplying an argument overrides the matching variable, while omitted parameters still fall back to theirs:

```
EXECUTE compute(multiplier := 3);
```

result
126

Because the fallback is resolved at execution time, re-executing the statement after changing a variable picks up the new value:

```
SET VARIABLE multiplier = 100;
EXECUTE compute;
```

result
4200

The fallback applies to named parameters only. Auto-incremented (`?`) and positional (`$1`) parameters are matched by position and are never filled from a session variable, even one whose name is a number such as `1`.

Dropping Prepared Statements: DEALLOCATE

To drop a prepared statement, use the DEALLOCATE statement:

```
DEALLOCATE query_person;
```

Alternatively, use:

```
DEALLOCATE PREPARE query_person;
```

Data Types

Data Types

General-Purpose Data Types

The table below shows all the built-in general-purpose data types. The alternatives listed in the aliases column can be used to refer to these types as well, however, note that the aliases are not part of the SQL standard and hence might not be accepted by other database engines.

Name	Aliases	Description
`BIGINT`	`INT8`, `LONG`	Signed eight-byte integer
`BIT`	`BITSTRING`	String of 1s and 0s
`BLOB`	`BYTEA`, `BINARY`, `VARBINARY`	Variable-length binary data
`BIGNUM`		Variable-length integer
`BOOLEAN`	`BOOL`, `LOGICAL`	Logical Boolean (`true` / `false`)
`DATE`		Calendar date (year, month, day)
`DECIMAL(prec, scale)`	`NUMERIC(prec, scale)`	Fixed-precision number with the given width (precision) and scale, defaults to `prec = 18` and `scale = 3`
`DOUBLE`	`FLOAT8`	Double precision floating-point number (8 bytes)
`FLOAT`	`FLOAT4`, `REAL`	Single precision floating-point number (4 bytes)
`GEOMETRY`		Geospatial entity, see the GEOMETRY data type
`HUGEINT`		Signed sixteen-byte integer
`INTEGER`	`INT4`, `INT`, `SIGNED`	Signed four-byte integer
`INTERVAL`		Date / time delta
`JSON`		JSON object (via the `json` extension)
`SMALLINT`	`INT2`, `SHORT`	Signed two-byte integer
`TIME`		Time of day (no time zone)
`TIMESTAMP WITH TIME ZONE`	`TIMESTAMPTZ`	Combination of time and date that uses the current time zone
`TIMESTAMP`	`DATETIME`	Combination of time and date
`TINYINT`	`INT1`	Signed one-byte integer
`UBIGINT`		Unsigned eight-byte integer
`UHUGEINT`		Unsigned sixteen-byte integer
`UINTEGER`		Unsigned four-byte integer
`USMALLINT`		Unsigned two-byte integer
`UTINYINT`		Unsigned one-byte integer

Name	Aliases	Description
UUID		UUID data type
VARCHAR	CHAR, BPCHAR, TEXT, STRING	Variable-length character string

Implicit and explicit typecasting is possible between numerous types, see the Typecasting page for details.

Nested / Composite Types

DuckDB supports the following nested data types: ARRAY, LIST, MAP, STRUCT, TUPLE, UNION and VARIANT. Each supports different use cases and has a different structure.

Name	Description	Rules when used in a column	Build from values	Define in DDL/CREATE
ARRAY	An ordered, fixed-length sequence of data values of the same type.	Each row must have the same data type within each instance of the ARRAY and the same number of elements.	[1, 2, 3]	INTEGER[3]
LIST	An ordered sequence of data values of the same type.	Each row must have the same data type within each instance of the LIST, but can have any number of elements.	[1, 2, 3]	INTEGER[]
MAP	A dictionary of multiple named values, each key having the same type and each value having the same type. Keys and values can be any type and can be different types from one another.	Rows may have different keys.	map([1, 2], ['a', 'b'])	MAP(INTEGER, VARCHAR)
STRUCT	A dictionary of multiple named values, where each key is a string, but the value can be a different type for each key.	Each row must have the same keys.	{'i': 42, 'j': 'a'}	STRUCT(i INTEGER, j VARCHAR)
TUPLE	A sequence of multiple unnamed values, accessed by position, where the value can be a different type for each position.	Each row must have the same number of entries with the same types.	(42, 'a')	TUPLE(INTEGER, VARCHAR)
UNION	A union of multiple alternative data types, storing one of them in each value at a time. A union also contains a discriminator "tag" value to inspect and access the currently set member type.	Rows may be set to different member types of the union.	union_value(num := 2)	UNION(num INTEGER, text VARCHAR)
VARIANT	A semi-structured type where each value is self-contained with its own type information.	Each row may hold a value of a different type.	42::VARIANT	VARIANT

Rules for Case Sensitivity

The keys of MAPs are case-sensitive, while keys of UNIONs and STRUCTs are case-insensitive. For examples, see the Rules for Case Sensitivity section.

Updating Values of Nested Types

When performing *updates* on values of nested types, DuckDB performs a *delete* operation followed by an *insert* operation. When used in a table with ART indexes (either via explicit indexes or primary keys/unique constraints), this can lead to unexpected constraint violations.

Nesting

ARRAY, LIST, MAP, STRUCT, TUPLE and UNION types can be arbitrarily nested to any depth, so long as the type rules are observed.

Struct with LISTs:

```
SELECT {'birds': ['duck', 'goose', 'heron'], 'aliens': NULL, 'amphibians': ['frog', 'toad']};
```

Struct with list of MAPs:

```
SELECT {'test': [MAP([1, 5], [42.1, 45]), MAP([1, 5], [42.1, 45])]};
```

A list of UNIONs:

```
SELECT [union_value(num := 2), union_value(str := 'ABC')::UNION(str VARCHAR, num INTEGER)];
```

Performance Implications

The choice of data types can have a strong effect on performance. Please consult the Performance Guide for details.

Array Type

An ARRAY column stores fixed-sized arrays. All fields in the column must have the same length and the same underlying type. Arrays are typically used to store arrays of numbers, but can contain any uniform data type, including ARRAY, LIST and STRUCT types.

Arrays can be used to store vectors such as word embeddings or image embeddings.

To store variable-length lists, use the LIST type. See the data types overview for a comparison between nested data types.

> The ARRAY type in PostgreSQL allows variable-length fields. DuckDB's ARRAY type is fixed-length.

Creating Arrays

Arrays can be created using the `array_value(expr, ...)` function.

Construct with the `array_value` function:

```
SELECT array_value(1, 2, 3);
```

You can always implicitly cast an array to a list (and use list functions, like `list_extract`, `[i]`):

```
SELECT array_value(1, 2, 3)[2];
```

You can cast from a list to an array (the dimensions have to match):

```
SELECT [3, 2, 1]::INTEGER[3];
```

Arrays can be nested:

```
SELECT array_value(array_value(1, 2), array_value(3, 4), array_value(5, 6));
```

Arrays can store structs:

```
SELECT array_value({'a': 1, 'b': 2}, {'a': 3, 'b': 4});
```

Defining an Array Field

Arrays can be created using the `<TYPE_NAME>[<LENGTH>]{:.language-sql .highlight}` syntax. For example, to create an array field for 3 integers, run:

```
CREATE TABLE array_table (id INTEGER, arr INTEGER[3]);
INSERT INTO array_table VALUES (10, [1, 2, 3]), (20, [4, 5, 6]);
```

Retrieving Values from Arrays

Retrieving one or more values from an array can be accomplished using brackets and slicing notation, or through list functions like `list_extract` and `array_extract`. Using the example in Defining an Array Field.

The following queries for extracting the first element of an array are equivalent:

```
SELECT id, arr[1] AS element FROM array_table;
SELECT id, list_extract(arr, 1) AS element FROM array_table;
SELECT id, array_extract(arr, 1) AS element FROM array_table;
```

id	element
10	1
20	4

Using the slicing notation returns a `LIST`:

```
SELECT id, arr[1:2] AS elements FROM array_table;
```

id	elements
10	[1, 2]
20	[4, 5]

Functions

All `LIST` functions work with the ARRAY type. Additionally, several ARRAY-native functions are also supported. See the ARRAY functions.

Examples

Create sample data:

```
CREATE TABLE x (i INTEGER, v FLOAT[3]);
CREATE TABLE y (i INTEGER, v FLOAT[3]);
INSERT INTO x VALUES (1, array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT));
INSERT INTO y VALUES (1, array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT));
```

Compute cross product:

```
SELECT array_cross_product(x.v, y.v)
FROM x, y
WHERE x.i = y.i;
```

Compute cosine similarity:

```
SELECT array_cosine_similarity(x.v, y.v)
FROM x, y
WHERE x.i = y.i;
```

Ordering

The ordering of ARRAY instances is defined using a lexicographical order. NULL values compare greater than all other values and are considered equal to each other.

See Also

For more functions, see List Functions.

Bitstring Type

Name	Aliases	Description
BITSTRING	BIT	Variable-length strings of 1s and 0s

Bitstrings are strings of 1s and 0s. The bit type data is of variable length. A bitstring value requires 1 byte for each group of 8 bits, plus a fixed amount to store some metadata.

By default bitstrings will not be padded with zeroes. Bitstrings can be very large, having the same size restrictions as BLOBs.

Creating a Bitstring

A string encoding a bitstring can be cast to a BITSTRING:

```
SELECT '101010'::BITSTRING AS b;
```

b
101010

A bitstring can also be created using a bit string literal, which is a string of 1s and 0s prefixed with B:

```
SELECT B'101010' AS b;
```

b
101010

> New Starting with DuckDB 2.0, `B'…'` literals are `BITSTRING` values. In earlier versions, `B'0101'` was parsed as the `VARCHAR` string `'b0101'`.

Creating a `BITSTRING` with a predefined length is possible with the `bitstring` function. The resulting bitstring will be left-padded with zeroes.

```
SELECT bitstring('0101011', 12) AS b;
```

b
000000101011

Numeric values (integer and float values) can also be converted to a `BITSTRING` via casting. For example:

```
SELECT 123::BITSTRING AS b;
```

b
00000000000000000000000001111011

Functions

See Bitstring Functions.

Blob Type

Name	Aliases	Description
BLOB	BYTEA, BINARY, VARBINARY	Variable-length binary data

The blob (**B**inary **L**arge **OB**ject) type represents an arbitrary binary object stored in the database system. The blob type can contain any type of binary data with no restrictions. What the actual bytes represent is opaque to the database system.

Create a `BLOB` value with a single byte (170):

```
SELECT '\xAA'::BLOB;
```

Create a `BLOB` value with three bytes (170, 171, 172):

```
SELECT '\xAA\xAB\xAC'::BLOB;
```

Create a `BLOB` value with two bytes (65, 66):

```
SELECT 'AB'::BLOB;
```

A `BLOB` value can also be created using a hex string literal, which is a string of hexadecimal digits prefixed with `X`. The number of hex digits must be even. For example, create a `BLOB` value with two bytes (202, 254):

```
SELECT X'CAFE';
```

> New Starting with DuckDB 2.0, `X'…'` literals are `BLOB` values. In earlier versions, `X'CAFE'` was parsed as the `VARCHAR` string `'xCAFE'`.

Blobs are typically used to store non-textual objects that the database does not provide explicit support for, such as images. While blobs can hold objects up to 4 GB in size, typically it is not recommended to store very large objects within the database system. In many situations it is better to store the large file on the file system, and store the path to the file in the database system in a `VARCHAR` field.

Functions

See Blob Functions.

Boolean Type

Name	Aliases	Description
BOOLEAN	BOOL	Logical Boolean (`true` / `false`)

The BOOLEAN type represents a statement of truth ("true" or "false"). In SQL, the BOOLEAN field can also have a third state "unknown" which is represented by the SQL NULL value.

Select the three possible values of a BOOLEAN column:

```
SELECT true, false, NULL::BOOLEAN;
```

Boolean values can be explicitly created using the literals `true` and `false`. However, they are most often created as a result of comparisons or conjunctions. For example, the comparison `i > 10` results in a Boolean value. Boolean values can be used in the WHERE and HAVING clauses of a SQL statement to filter out tuples from the result. In this case, tuples for which the predicate evaluates to `true` will pass the filter, and tuples for which the predicate evaluates to `false` or NULL will be filtered out. Consider the following example:

Create a table with the values 5, 15 and NULL:

```
CREATE TABLE integers (i INTEGER);
INSERT INTO integers VALUES (5), (15), (NULL);
```

Select all entries where `i > 10`:

```
SELECT * FROM integers WHERE i > 10;
```

In this case 5 and NULL are filtered out (`5 > 10` is `false` and `NULL > 10` is NULL):

i
15

Conjunctions

The AND / OR conjunctions can be used to combine Boolean values.

Below is the truth table for the AND conjunction (i.e., `x AND y`).

X	X AND true	X AND false	X AND NULL
true	true	false	NULL
false	false	false	false
NULL	NULL	false	NULL

Below is the truth table for the OR conjunction (i.e., `x OR y`).

X	X OR true	X OR false	X OR NULL
true	true	true	true
false	true	false	NULL
NULL	true	NULL	NULL

Expressions

See Logical Operators and Comparison Operators.

Date Types

Name	Aliases	Description
DATE		Calendar date (year, month, day)

A date specifies a combination of year, month and day. DuckDB follows the SQL standard's lead by counting dates exclusively in the Gregorian calendar, even for years before that calendar was in use. Dates can be created using the DATE keyword, where the data must be formatted according to the ISO 8601 format (YYYY-MM-DD).

```
SELECT DATE '1992-09-20';
```

Special Values

There are also three special date values that can be used on input:

Input string	Description
epoch	1970-01-01 (Unix system day zero)
infinity	Later than all other dates
-infinity	Earlier than all other dates

The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but `epoch` is simply a notational shorthand that will be converted to the date value when read.

```
SELECT
    '-infinity'::DATE AS negative,
    'epoch'::DATE AS epoch,
    'infinity'::DATE AS positive;
```

negative	epoch	positive
-infinity	1970-01-01	infinity

Functions

See Date Functions.

Enum Data Type

Name	Description
ENUM	Dictionary representing all possible string values of a column

The enum type represents a dictionary data structure with all possible unique values of a column. For example, a column storing the days of the week can be an enum holding all possible days. Enums are particularly interesting for string columns with low cardinality (i.e., fewer distinct values). This is because the column only stores a numerical reference to the string in the enum dictionary, resulting in immense savings in disk storage and faster query performance.

Creating Enums

You can create an enum using hardcoded values:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
-- This statement will fail since enums cannot hold NULL values:
-- CREATE TYPE mood AS ENUM ('sad', NULL);
-- This statement will fail since enum values must be unique:
-- CREATE TYPE mood AS ENUM ('sad', 'sad');
```

You can create enums in a specific schema:

```
CREATE SCHEMA my_schema;
CREATE TYPE my_schema.mood AS ENUM ('sad', 'ok', 'happy');
```

Anonymous enums can be created on the fly during casting:

```
SELECT 'clubs'::ENUM ('spades', 'hearts', 'diamonds', 'clubs');
```

You can also create an enum using a `SELECT` statement that returns a single column of VARCHARs. The set of values from the select statement will be deduplicated automatically, and `NULL` values will be ignored:

```
CREATE TYPE region AS ENUM (SELECT region FROM sales_data);
```

If you are importing data from a file, you can create an enum for a VARCHAR column before importing:

```
CREATE TYPE region AS ENUM (SELECT region FROM 'sales_data.csv');
CREATE TABLE sales_data (amount INTEGER, region region);
COPY sales_data FROM 'sales_data.csv';
```

Using Enums

Enum values are case-sensitive, so `'maltese'` and `'Maltese'` are considered different values:

```
CREATE TYPE breed AS ENUM ('maltese', 'Maltese');
-- Will return false
SELECT 'maltese'::breed = 'Maltese'::breed;
-- Will error
SELECT 'MALTESE'::breed;
```

After an enum has been created, it can be used anywhere a standard built-in type is used. For example, we can create a table with a column that references the enum.

```
CREATE TABLE person (
    name TEXT,
    current_mood mood
);
INSERT INTO person VALUES
```

```
    ('Pedro', 'happy'),
    ('Mark', NULL),
    ('Pagliacci', 'sad'),
    ('Mr. Mackey', 'ok');
```

The following query will fail since the mood type does not have a `quackity-quack` value.

```
INSERT INTO person VALUES ('Hannes', 'quackity-quack');
```

Enums vs. Strings

DuckDB enums are automatically cast to VARCHAR types whenever necessary. This characteristic allows for comparisons between different enums, or an enum and a VARCHAR column.

It also allows for an enum to be used in any VARCHAR function. For example:

```
SELECT current_mood, regexp_matches(current_mood, '.*a.*') AS contains_a FROM person;
```

current_mood	contains_a
happy	true
NULL	NULL
sad	true
ok	false

When comparing two different enum types, DuckDB will cast both to strings and perform a string comparison:

```
CREATE TYPE new_mood AS ENUM ('happy', 'anxious');
SELECT * FROM person
WHERE current_mood = 'happy'::new_mood;
-- Equivalent to `WHERE current_mood::VARCHAR = 'happy'::VARCHAR`
```

name	current_mood
Pedro	happy

When comparing an enum to a VARCHAR, DuckDB will cast the enum to VARCHAR and perform a string comparison:

```
SELECT * FROM person
WHERE current_mood = name;
-- Equivalent to `WHERE current_mood::VARCHAR = name`
-- No rows returned
```

When comparing against a constant string, DuckDB will perform an optimization and `try_cast(<constant string>, enum_type)`{:.language-sql .highlight} so that physically we are doing an integer comparison instead of a string comparison (but logically it is still a string comparison):

```
SELECT * FROM person
WHERE current_mood = 'sad';
-- Equivalent to `WHERE current_mood::VARCHAR = 'sad'`
```

name	current_mood
Pagliacci	sad

> **Warning.** This means that comparing against a random (non-equivalent) string always results in `false` (and does not error):

```
SELECT * FROM person
WHERE current_mood = 'bogus';
-- Equivalent to `WHERE current_mood::VARCHAR = 'bogus'`
-- No rows returned
```

If you want to enforce type-safety, cast to the enum explicitly:

```
SELECT * FROM person
WHERE current_mood = 'bogus'::mood;
-- Conversion Error: Could not convert string 'bogus' to ENUM('sad', 'ok', 'happy')
```

Ordering of Enums

Enum values are ordered according to their order in the enum's definition. For example:

```
CREATE TYPE priority AS ENUM ('low', 'medium', 'high');
SELECT 'low'::priority < 'high'::priority AS comp;
-- note that 'low'::VARCHAR < 'high'::VARCHAR is false!
```

comp
true

```
SELECT unnest(['medium'::priority, 'high'::priority, 'low'::priority]) AS m
ORDER BY m;
```

m
low
medium
high

> **Warning.** If you compare an enum to a non-enum (e.g., a VARCHAR or a different enum type), the enum will first be cast to a string (as described in the previous section), and the comparison will be done lexicographically as with strings:

```
CREATE TABLE tasks (name TEXT, priority_level priority);
INSERT INTO tasks VALUES ('a', 'low'), ('b', 'medium'), ('c', 'high');
-- WARNING!
-- Equivalent to `WHERE priority_level::VARCHAR >= 'medium'`
SELECT * FROM tasks
WHERE priority_level >= 'medium';
-- Misses the 'high' priority task!
```

name	priority_level
b	medium

So, if you want to e.g. "get all priorities at or above `medium`" then explicitly cast to the enum type:

```
SELECT * FROM tasks
WHERE priority_level >= 'medium'::priority;
```

name	priority_level
b	medium
c	high

Functions

See Enum Functions.

For example, show the available values in the mood enum using the `enum_range` function:

```
SELECT enum_range(NULL::mood) AS my_enum_range;
```

my_enum_range
[sad, ok, happy]

Enum Removal

Enum types are stored in the catalog, and a catalog dependency is added to each table that uses them. It is possible to drop an enum from the catalog using the following command:

```
DROP TYPE <enum_name>;
```

Currently, it is possible to drop enums that are used in tables without affecting the tables.

> **Warning.** This behavior of the enum removal feature is subject to change. In future releases, it is expected that any dependent columns must be removed before dropping the enum, or the enum must be dropped with the additional CASCADE parameter.

Geometry Data Type

Name	Description
GEOMETRY	Geospatial entity

The GEOMETRY data type is used to store and manipulate geometric objects, such as points, lines, and polygons.

The GEOMETRY type was part of the `spatial` extension but became a built-in data type in DuckDB 1.5. Most of the benefits of having GEOMETRY as a built-in type (e.g., storage optimizations, statistics, etc.) are therefore only available in databases using storage version v1.5 and above. However, almost all of the associated functions for working with geometries (e.g., calculating distances, areas, intersections) are still part of `spatial`.

Types of Geometries

Conceptually, the GEOMETRY type follows the core data model defined in the Simple Features standard, which is widely used in geospatial databases and GIS software. A GEOMETRY value can therefore represent 7 types of shapes:

Geometry Type	Description
Point	A single location in space, defined by its coordinates (e.g., longitude and latitude).
LineString	A sequence of points connected by straight lines, representing a path or route.
Polygon	A set of closed rings defined by a sequence of points, representing an area such as a country border or a building footprint. The first ring is the "shell", and "interior" rings represent holes in the polygon.
MultiPoint	A collection of points.
MultiLineString	A collection of LineStrings.
MultiPolygon	A collection of Polygons.
GeometryCollection	A collection of different geometry types, allowing for complex geometries that combine points, lines, and polygons or even other nested geometry collections.

The textual representation of geometries uses "Well-Known Text" (WKT) format. Geometries can be cast to and from WKT strings, so you can use string literals to create geometries directly in SQL statements.

In the following example, we create a GEOMETRY column with the 7 different types of supported geometries:

```sql
CREATE TABLE geometries (
    id INTEGER,
    geom GEOMETRY
);

INSERT INTO geometries VALUES
  (1, 'POINT (30 10)'),
  (2, 'LINESTRING (30 10, 10 30, 40 40)'),
  (3, 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))'),
  (4, 'MULTIPOINT ((10 40), (40 30), (20 20), (30 10))'),
  (5, 'MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20))'),
  (6, 'MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10,15 5)))'),
  (7, 'GEOMETRYCOLLECTION (POINT(40 10), LINESTRING(10 10,20 20,10 40), POLYGON((40 40,20 45,45 30,40
40)))');

SELECT * FROM geometries;
----
```

id int32	geom geometry
1	POINT (30 10)
2	LINESTRING (30 10, 10 30, 40 40)
3	POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
4	MULTIPOINT (10 40, 40 30, 20 20, 30 10)
5	MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20))
6	MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))

```
│     7 │ GEOMETRYCOLLECTION (POINT (40 10), LINESTRING (10 10, 20 20, 10 40), POLYGON ((40 40, 20 45, 45
30, 40 40))) │
└───────┴──────────────────────────────────────────────────────────────────────────────────────────────────
```

Multi-Dimensional Geometries

The GEOMETRY type is primarily used to model shapes in two dimensions (e.g. X/Y or `longitude/latitude`), but it also supports shapes with additional vertex dimensions such as Z for elevation or M for "measure", or both.

The vertex dimensions of a GEOMETRY value must be consistent across all vertices. For example, if one vertex has X, Y, and Z coordinates, then all other vertices in that geometry must also have X, Y, and Z coordinates. This means that you cannot have a mix of 2D and 3D vertices within the same geometry. This also applies for collections of geometries, such as MULTIPOINT or GEOMETRYCOLLECTION, where all geometries within the collection must have the same vertex dimensions.

Functions that operate on GEOMETRY values typically ignore any additional dimensions beyond the X and Y unless explicitly specified, but they can still be stored and can be retrieved if needed.

In the following example, we create a GEOMETRY table with 2D, 3D(Z), 3D(M) and 4D(ZM) points:

```
CREATE TABLE points (
    id INTEGER,
    geom GEOMETRY
);

INSERT INTO points VALUES
  (1, 'POINT (30 10)'),
  (2, 'POINT Z (30 10 5)'),
  (3, 'POINT M (30 10 1)'),
  (4, 'POINT ZM (30 10 5 1)');

SELECT * FROM points;
```

```
┌─────────┬──────────────────────┐
│   id    │         geom         │
│  int32  │       geometry       │
├─────────┼──────────────────────┤
│       1 │ POINT (30 10)        │
│       2 │ POINT Z (30 10 5)    │
│       3 │ POINT M (30 10 1)    │
│       4 │ POINT ZM (30 10 5 1) │
└─────────┴──────────────────────┘
```

```
-- But we cannot mix different vertex dimensions within the same geometry!
INSERT INTO points VALUES
  (5, 'GEOMETRYCOLLECTION (POINT (30 10), POINT Z (30 10 5))');
```

```
Invalid Input Error:
Failed to parse geometry: Geometry has inconsistent Z/M dimensions at offset 43
```

Empty Geometries

Geometries can also be "empty" (e.g., `POINT EMPTY`, `LINESTRING EMPTY`, `MULTIPOLYGON EMPTY`, etc.) which means they don't contain any vertices. Empty geometries are still valid geometries and can be used in spatial operations, but they are mostly useful for representing the result of topological operations that don't have a valid geometrical representation (e.g., the intersection of two non-overlapping geometries is an empty geometry).

Geometry Storage

Internally `GEOMETRY` values are stored as a sequence of bytes, similarly to DuckDB's `BLOB` types. The exact binary format is not yet stabilized and may change in a future release, but as of DuckDB storage version v1.5 it is based on little-endian Well-Known Binary (WKB), which is a standard binary encoding for geometries. In older storage versions, geometries were stored in a different custom binary format used by the `spatial` extension, but this conversion is performed automatically at the storage layer and is not visible to the execution engine or the user.

Shredding and Compression

The `GEOMETRY` type supports a storage optimization called "shredding", which improves compression for geometry columns where all values share the same geometry type and vertex dimensions.

When a row group qualifies, DuckDB splits the geometry segment within the row group into primitive `STRUCT`, `LIST`, and `DOUBLE` segments that can be compressed independently using lightweight algorithms - far more efficiently than storing variable-size binary blobs.

The shredded layout depends on the geometry type:

- `POINT` - STRUCT(X DOUBLE, Y DOUBLE) (and/or Z, M)
- `LINESTRING` - STRUCT(X DOUBLE, Y DOUBLE)[]
- `POLYGON` - STRUCT(X DOUBLE, Y DOUBLE)[][]
- `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` - same as above, with one additional level of list nesting

Row groups are not shredded if they contain `GEOMETRYCOLLECTION`s, any `EMPTY` geometries, or multiple geometry sub-types.

Additionally, row groups are not shredded if they fall below the minimum size threshold (default: ~25% of the maximum row group size, i.e., 30,000 rows).

This threshold is configurable via the `geometry_minimum_shredding_size` setting. Set it to `0` to always shred, or `-1` to disable shredding entirely.

```
-- Disable shredding for geometry columns
SET geometry_minimum_shredding_size = -1;

-- Always shred geometry columns regardless of row group size
SET geometry_minimum_shredding_size = 0;
```

The primary benefit of shredding is significantly improved compression, but in the future we plan to add ways to expose the shredded representation directly to the execution engine without having to "reassemble" the geometry back into binary again.

The following example illustrates the effects of shredding on the storage footprint of a `GEOMETRY` column.

```
-- Attach a persistent database
-- (geometry shredding requires storage version v1.5.0 or later, which is the default in DuckDB 2.0)
ATTACH 'geometry_db.db' AS geometry_db;

USE geometry_db;

-- Disable shredding completely and create a table with 1 million 2D points
SET geometry_minimum_shredding_size = -1;

CREATE OR REPLACE TABLE points AS SELECT printf('POINT (%d %d)', x, y)::GEOMETRY AS geom
FROM range(0, 1000) AS rx(x), range(0, 1000) AS ry(y);

-- Checkpoint the database to persist the data and storage layout to disk
CHECKPOINT;

-- Attach a second database
ATTACH 'shredded_db.db' AS shredded_db;

USE shredded_db;
```

```
-- This time, set the minimum shredding size to 0 to always shred geometry columns,
-- and create the same table with 1 million 2D points
SET geometry_minimum_shredding_size = 0;

CREATE OR REPLACE TABLE points AS SELECT printf('POINT (%d %d)', x, y)::GEOMETRY AS geom
FROM range(0, 1000) AS rx(x), range(0, 1000) AS ry(y);

-- Checkpoint to persist the data and storage layout to disk, and apply shredding
CHECKPOINT;

-- Now check the storage layout and memory usage of the geometry column in both attached databases
SELECT database_name, database_size FROM pragma_database_size();
----
```

database_name varchar	database_size varchar
shredded_db	2.2 MiB
geometry_db	6.5 MiB
memory	0 bytes

```
-- Almost 3x smaller storage thanks to shredding!

-- We can inspect what type of segments are used to store the geometry column
-- in each database using the `pragma_storage_info` function.

-- The geometry column in `geometry_db` is stored as regular GEOMETRY segments
SELECT DISTINCT(segment_type) FROM pragma_storage_info('geometry_db.points');
----
```

segment_type varchar
GEOMETRY
VALIDITY

```
-- While the geometry column in `shredded_db` is decomposed into primitive DOUBLE segments,
-- which can be compressed much more efficiently!
SELECT DISTINCT(segment_type) FROM pragma_storage_info('shredded_db.points');
----
```

segment_type varchar
VALIDITY
DOUBLE

Geometry Statistics

GEOMETRY columns contain geometry-specific statistics that track the bounding box of the geometries in each row group, as well as the set of geometry types and vertex dimensions that are present within the row group.

You can inspect the statistics of a column using the `stats()` function:

```
CREATE TABLE geometries AS SELECT 'POINT Z (30 10 5)'::GEOMETRY AS geom;

SELECT stats(geom) AS geom_stats FROM geometries;
```

The result is a VARIANT value:

```
{'approx_unique': 1, 'extent': {'x_max': 30.0, 'x_min': 30.0, 'y_max': 10.0, 'y_min': 10.0, 'z_max': 5.0,
'z_min': 5.0}, 'has_empty_geom': false, 'has_empty_part': false, 'has_no_null': true, 'has_non_empty_geom':
true, 'has_non_empty_part': true, 'has_null': false}
```

These statistics can be used by the query optimizer to skip row groups that don't match the geometry type or vertex dimensions required by a query, or to speed up spatial predicates by first checking if the bounding box of the geometries in the row group overlaps with the bounding box of the query geometry.

Currently, only the && operator, which is used to check if the bounding box of a geometry intersects the bounding box of another geometry, can take advantage of geometry statistics when used in a `WHERE` clause. There is ongoing work to add support for more statistics-based optimizations to the functions in the `spatial` extension, such as `ST_Intersects`, `ST_Distance`, etc.

Persisting geometry statistics is only possible in storage versions v1.5 and above, and so if you are using an older storage version, the geometry statistics will turn into "unknown" statistics when checkpointing. In other words, the bounding box will be set to an infinitely large bounding box and all geometry types and vertex dimensions will be marked as maybe present, which means that the execution engine will not be able to do any optimizations based on the geometry statistics.

Coordinate Reference Systems

As far as the execution engine is concerned, geometries are considered to exist in a Cartesian coordinate system. In practice, however, most geospatial data is associated with a specific **Coordinate Reference System** (CRS) that defines how the coordinates relate to real-world locations on the Earth's surface.

A helpful analogy is to think of CRSs as the equivalent of "time zones", but for geospatial data. Just like how time zones define how local time relates to a standard reference time (e.g., UTC), CRSs define how the coordinates of a geometry relate to a standard reference system (e.g., WGS 84). CRSs are usually either geographic (e.g., WGS 84, which uses latitude and longitude) or projected (e.g., UTM, which uses linear units like meters).

When working with geospatial data, it's important to be aware of the CRS associated with different datasets. Performing spatial operations on geometries in different CRSs without proper transformation will most likely lead to incorrect results.

How are Coordinate Reference Systems Stored in DuckDB?

To avoid these kinds of mistakes, DuckDB makes it possible to explicitly associate a CRS with a `GEOMETRY` column.

This is done by passing a CRS "identifier" as a parameter of the `GEOMETRY` type. For example, a column of type `GEOMETRY('OGC:CRS84')` stores geometries that are associated with the "OGC CRS84" coordinate reference system.

CRS identifiers in DuckDB are always strings. `OGC:CRS84` is the identifier for a common geographic coordinate system spanning the whole globe where the X coordinate represents longitude and the Y coordinate represents latitude. DuckDB only knows this because the identifier `OGC:CRS84` is registered as a *known* CRS in the system catalog.

> **Warning.** Storing `GEOMETRY` columns that carry a CRS identifier requires storage version v1.5.0 or newer. Writing such a column into a database created with an older storage version fails with an error such as `Invalid Input Error: GEOMETRY columns with coordinate reference system identifiers are not supported in storage versions prior v1.5.0`. Create or attach the database with `ATTACH 'file.db' (STORAGE_VERSION 'v1.5.0');`.

By default, only a handful of common CRSs are registered as known, but extensions can also register additional known CRSs. In particular, the `spatial` extension registers over 7000 CRSs from the EPSG Geodetic Parameter Dataset, which is arguably the most widely used database of coordinate reference systems.

You can list all available CRSs known to DuckDB using the `duckdb_coordinate_systems()` function:

```
SELECT * FROM duckdb_coordinate_systems();
```

```
┌───────────────┬───────────────┬─────────────┬─────────────┬──────────┬──────────┬───────────┬───────────┬──────────┐
│ database_name │ database_oid  │ schema_name │ schema_oid  │ crs_oid  │ crs_name │ auth_name │ auth_code │
projjson                │              wkt2_2019              │
│    varchar    │     int64     │   varchar   │    int64    │  int64   │ varchar  │  varchar  │  varchar  │
varchar                 │              varchar                │
├───────────────┼───────────────┼─────────────┼─────────────┼──────────┼──────────┼───────────┼───────────┼──────────
```

```
| system        |            0 | main        |          0 |    1354 | OGC:CRS83 | OGC       | CRS83     |
{"$schema":"https://proj.org/schemas… | GEOGCRS["NAD83 (CRS83)",DATUM["North… |
| system        |            0 | main        |          0 |    1353 | OGC:CRS84 | OGC       | CRS84     |
{"$schema":"https://proj.org/schemas… | GEOGCRS["WGS 84 (CRS84)",ENSEMBLE["W… |
```

Handling Unknown Coordinate Reference Systems

As mentioned above, only coordinate systems that are registered in the system catalog (and therefore "known" to DuckDB) can be used when creating `GEOMETRY` columns. If you try to create a `GEOMETRY` column with an unknown CRS identifier, either manually or by importing an external geospatial dataset, the statement will fail with an error.

```
SELECT 'POINT(1 2)'::GEOMETRY('DUCK:1337') AS my_point;
----
Binder Error:
Encountered unrecognized coordinate system 'DUCK:1337' when trying to create GEOMETRY type
The coordinate system definition may be incomplete or invalid ...
```

This restriction exists because DuckDB needs the complete CRS definition, not just an identifier, to perform coordinate transformations and to export to formats that embed CRS metadata, such as GeoParquet. Without a system catalog entry, there is no way to resolve an identifier to its full definition.

You can set the `ignore_unknown_crs` configuration option to `true` to simply skip any unknown CRSs and create `GEOMETRY` columns without CRS instead.

```
-- Ignore any unknown CRS identifiers
SET ignore_unknown_crs = true;

select 'POINT(1 2)'::GEOMETRY('DUCK:1337') AS my_point;
----
┌──────────────┐
│   my_point   │
│   geometry   │
├──────────────┤
│ POINT (1 2)  │
└──────────────┘
```

`-- The geometry is created, but the CRS is dropped from the type!`

Alternatively, if you are trying to define a `GEOMETRY` column yourself, you can provide a complete CRS definition in WKT or PROJJSON format instead of a shorthand identifier as the CRS parameter. However, as complete CRS definitions are usually very large, this gets unwieldy very quickly and is not recommended for interactive use.

It is currently not possible to define a custom CRS from within SQL, or to persist custom CRS definitions in a database such that DuckDB can use them to resolve CRS identifiers for geometry columns, but this is something we are considering for the future.

Working with Geometries in Different Coordinate Reference Systems

One benefit of tracking CRSs as part of the type system is that it prevents a lot of common mistakes that can occur when working with geometries from different coordinate systems. Most spatial functions that operate on multiple `GEOMETRY` values verify that all input expressions have the same CRS before performing the operation. Similarly, `GEOMETRY` columns can only be implicitly cast to and from other `GEOMETRY` columns if the source or the target don't have a CRS specified.

To convert a geometry from one CRS to another, you can use the `ST_Transform(geom, crs)` function from the `spatial` extension.

```
LOAD spatial;
-- required in DuckDB 2.0: assume [longitude, latitude] axis order
SET geometry_always_xy = true;

SELECT ST_Transform('POINT(4.897070 52.377956)'::GEOMETRY('OGC:CRS84'), 'EPSG:3857') AS transformed;
```

```
┌──────────────────────────────────────────────┐
│                  transformed                 │
│          geometry('epsg:3857')               │
├──────────────────────────────────────────────┤
│ POINT (545139.3387790163 6868755.38408516)   │
└──────────────────────────────────────────────┘
```

You can also use the `ST_SetCRS(geom, crs)` function to assign a CRS to a geometry that doesn't have one, or to reassign a CRS without transforming coordinates (e.g., when the data is already in the correct coordinate system but lacks the correct CRS).

```
SELECT ST_SetCRS('POINT(4.897070 52.377956)'::GEOMETRY, 'OGC:CRS84') AS with_crs;
```

```
┌──────────────────────────────┐
│           with_crs           │
│  geometry('ogc:crs84')       │
├──────────────────────────────┤
│ POINT (4.89707 52.377956)    │
└──────────────────────────────┘
```

Or if you want to remove the CRS from a geometry, you can either just cast to `GEOMETRY`, or set the CRS to `''`:

```
SELECT 'POINT(4.897070 52.377956)'::GEOMETRY('OGC:CRS84')::GEOMETRY AS no_crs;
```

```
┌──────────────────────────────┐
│            no_crs            │
│           geometry           │
├──────────────────────────────┤
│ POINT (4.89707 52.377956)    │
└──────────────────────────────┘
```

```
SELECT ST_SetCRS('POINT(4.897070 52.377956)'::GEOMETRY('OGC:CRS84'), '') AS no_crs;
```

```
┌──────────────────────────────┐
│            no_crs            │
│           geometry           │
├──────────────────────────────┤
│ POINT (4.89707 52.377956)    │
└──────────────────────────────┘
```

You can of course also use `ST_CRS(geom)` to retrieve the CRS of a geometry:

```
SELECT ST_CRS('POINT(4.897070 52.377956)'::GEOMETRY('OGC:CRS84')) AS crs;
```

```
┌──────────────┐
│     crs      │
│   varchar    │
├──────────────┤
│ OGC:CRS84    │
└──────────────┘
```

Functions

- See geometry functions for the list of built-in geometry functions.
- See the documentation of the `spatial` extension for the large set of additional geometry functions provided by the extension, including functions for calculating areas, distances, intersections, unions, and much more.

Interval Type

`INTERVAL`s represent periods of time that can be added to or subtracted from `DATE`, `TIMESTAMP`, `TIMESTAMPTZ`, or `TIME` values.

Name	Description
INTERVAL	Period of time

An INTERVAL can be constructed by providing amounts together with units. Units that aren't *months*, *days*, or *microseconds* are converted to equivalent amounts in the next smaller of these three basis units.

```
SELECT
    INTERVAL 1 YEAR, -- single unit using YEAR keyword; stored as 12 months
    INTERVAL (random() * 10) YEAR, -- parentheses necessary for variable amounts;
                                   -- stored as integer number of months
    INTERVAL '1 month 1 day', -- string type necessary for multiple units; stored as (1 month, 1 day)
    '16 months'::INTERVAL, -- string cast supported; stored as 16 months
    '48:00:00'::INTERVAL, -- HH::MM::SS string supported; stored as (48 * 60 * 60 * 1e6 microseconds)
;
```

> **Warning.** Decimal values are truncated to integers when used with unit keywords (unless the unit is SECONDS or MILLISECONDS).
>
> ```
> SELECT INTERVAL '1.5' YEARS;
> -- Returns 12 months; equivalent to `to_years(CAST(trunc(1.5) AS INTEGER))`
> ```
>
> For more precision, include the unit in the string or use a more granular unit; e.g., INTERVAL '1.5 years' or INTERVAL 18 MONTHS.

Three independent basis units are necessary because a month does not correspond to a fixed amount of days (February has fewer days than March) and a day doesn't correspond to a fixed amount of microseconds (days can be 25 hours or 23 hours long because of daylight saving time). The division into components makes the INTERVAL class suitable for adding or subtracting specific time units to a date. For example, we can generate a table with the first day of every month using the following SQL query:

```
SELECT DATE '2000-01-01' + INTERVAL (i) MONTH
FROM range(12) t(i);
```

When INTERVALs are deconstructed via the datepart function, the *months* component is additionally split into years and months, and the *microseconds* component is split into hours, minutes and microseconds. The *days* component is not split into additional units. To demonstrate this, the following query generates an INTERVAL called period by summing random amounts of the three basis units. It then extracts the aforementioned six parts from period, adds them back together, and confirms that the result is always equal to the original period.

```
SELECT
    period = list_reduce(
        [INTERVAL (datepart(part, period) || part) FOR part IN
             ['year', 'month', 'day', 'hour', 'minute', 'microsecond']
        ],
        lambda i1, i2: i1 + i2
    ) -- always true
FROM (
    VALUES (
        INTERVAL (random() * 123_456_789_123) MICROSECONDS
        + INTERVAL (random() * 12_345) DAYS
        + INTERVAL (random() * 12_345) MONTHS
    )
) _(period);
```

> **Warning.** The *microseconds* component is split only into hours, minutes and microseconds, rather than hours, minutes, *seconds* and microseconds.

The following table describes how these parts are extracted by datepart in formulas, as a function of the three basis units.

Part	Formula
year	#months // 12
month	#months % 12
day	#days
hour	#microseconds // (60 * 60 * 1_000_000)
minute	(#microseconds // (60 * 1_000_000)) % 60
microsecond	#microseconds % (60 * 1_000_000)

Additionally, `datepart` may be used to extract centuries, decades, quarters, seconds and milliseconds from `INTERVAL`s. However, these parts are not required when reassembling the original `INTERVAL`. In fact, if the previous query additionally extracted any of these additional parts, then the sum of the extracted parts would generally be larger than the original `period`.

Part	Formula
century	datepart('year', interval) // 100
decade	datepart('year', interval) // 10
quarter	datepart('month', interval) // 3 + 1
second	datepart('microsecond', interval) // 1_000_000
millisecond	datepart('microsecond', interval) // 1_000

> All units use 0-based indexing, except for quarters, which use 1-based indexing.

For example:

```
SELECT
    datepart('decade', INTERVAL 12 YEARS), -- returns 1
    datepart('year', INTERVAL 12 YEARS), -- returns 12
    datepart('second', INTERVAL 1_234 MILLISECONDS), -- returns 1
    datepart('microsecond', INTERVAL 1_234 MILLISECONDS), -- returns 1_234_000
;
```

Arithmetic with Timestamps, Dates and Intervals

`INTERVAL`s can be added to and subtracted from `TIMESTAMP`s, `TIMESTAMPTZ`s, `DATE`s, and `TIME`s using the `+` and `-` operators.

```
SELECT
    DATE '2000-01-01' + INTERVAL 1 YEAR,
    TIMESTAMP '2000-01-01 01:33:30' - INTERVAL '1 month 13 hours',
    TIME '02:00:00' - INTERVAL '3 days 23 hours', -- wraps; equals TIME '03:00:00'
;
```

> Adding an `INTERVAL` to a `DATE` returns a `TIMESTAMP` even when the `INTERVAL` has no microseconds component. The result is the same as if the `DATE` was cast to a `TIMESTAMP` (which sets the time component to `00:00:00`) before adding the `INTERVAL`.

Conversely, subtracting two `TIMESTAMP`s or two `TIMESTAMPTZ`s from one another creates an `INTERVAL` describing the difference between the timestamps with only the *days and microseconds* components. For example:

```
SELECT
    TIMESTAMP '2000-02-06 12:00:00' - TIMESTAMP '2000-01-01 11:00:00', -- 36 days 1 hour
    TIMESTAMP '2000-02-01' + (TIMESTAMP '2000-02-01' - TIMESTAMP '2000-01-01'), -- '2000-03-03', NOT
    '2000-03-01'
;
```

Subtracting two DATEs from one another does not create an INTERVAL but rather returns the number of days between the given dates as integer value.

> **Warning.** Extracting a part of the INTERVAL difference between two TIMESTAMPs is not equivalent to computing the number of partition boundaries between the two TIMESTAMPs for the corresponding unit, as computed by the datediff function:

```
SELECT
    datediff('day', TIMESTAMP '2020-01-01 01:00:00', TIMESTAMP '2020-01-02 00:00:00'), -- 1
    datepart('day', TIMESTAMP '2020-01-02 00:00:00' - TIMESTAMP '2020-01-01 01:00:00'), -- 0
;
```

Equality and Comparison

For equality and ordering comparisons only, the total number of microseconds in an INTERVAL is computed by converting the days basis unit to `24 * 60 * 60 * 1e6` microseconds and the months basis unit to 30 days, or `30 * 24 * 60 * 60 * 1e6` microseconds.

As a result, INTERVALs can compare equal even when they are functionally different, and the ordering of INTERVALs is not always preserved when they are added to dates or timestamps.

For example:

- `INTERVAL 30 DAYS = INTERVAL 1 MONTH`
- but `DATE '2020-01-01' + INTERVAL 30 DAYS != DATE '2020-01-01' + INTERVAL 1 MONTH`.

and

- `INTERVAL '30 days 12 hours' > INTERVAL 1 MONTH`
- but `DATE '2020-01-01' + INTERVAL '30 days 12 hours' < DATE '2020-01-01' + INTERVAL 1 MONTH`.

Functions

See the Date Part Functions page for a list of available date parts for use with an INTERVAL.

See the Interval Operators page for functions that operate on intervals.

List Type

A LIST column encodes lists of values. Fields in the column can have values with different lengths, but they must all have the same underlying type. LISTs are typically used to store arrays of numbers, but can contain any uniform data type, including other LISTs and STRUCTs.

LISTs are similar to PostgreSQL's ARRAY type. DuckDB uses the LIST terminology, but some `array_` functions are provided for PostgreSQL compatibility.

See the data types overview for a comparison between nested data types.

> For storing fixed-length lists, DuckDB uses the ARRAY type.

Creating Lists

Lists can be created using the `list_value(expr, ...)` function or the equivalent bracket notation `[expr, ...]`. The expressions can be constants or arbitrary expressions. To create a list from a table column, use the `list` aggregate function.

List of integers:

```
SELECT [1, 2, 3];
```

List of strings with a NULL value:

```
SELECT ['duck', 'goose', NULL, 'heron'];
```

List of lists with NULL values:

```
SELECT [['duck', 'goose', 'heron'], NULL, ['frog', 'toad'], []];
```

Create a list with the `list_value` function:

```
SELECT list_value(1, 2, 3);
```

Create a table with an INTEGER list column and a VARCHAR list column:

```
CREATE TABLE list_table (int_list INTEGER[], varchar_list VARCHAR[]);
```

Retrieving from Lists

Retrieving one or more values from a list can be accomplished using brackets and slicing notation, or through list functions like `list_extract`. Multiple equivalent functions are provided as aliases for compatibility with systems that refer to lists as arrays. For example, the function `array_slice`.

Example	Result
SELECT ['a', 'b', 'c'][3]	'c'
SELECT ['a', 'b', 'c'][-1]	'c'
SELECT ['a', 'b', 'c'][2 + 1]	'c'
SELECT list_extract(['a', 'b', 'c'], 3)	'c'
SELECT ['a', 'b', 'c'][1:2]	['a', 'b']
SELECT ['a', 'b', 'c'][:2]	['a', 'b']
SELECT ['a', 'b', 'c'][-2:]	['b', 'c']
SELECT list_slice(['a', 'b', 'c'], 2, 3)	['b', 'c']

Comparison and Ordering

The LIST type can be compared using all the comparison operators. These comparisons can be used in logical expressions such as WHERE and HAVING clauses, and return BOOLEAN values.

The LIST ordering is defined positionally using the following rules, where `min_len = min(len(l1), len(l2))`.

- **Equality.** `l1` and `l2` are equal, if `len(l1) = len(l2)` and for each `i` in `[1, min_len]`: `l1[i] = l2[i]`.
- **Less Than**. For the first index `i` in `[1, min_len]` where `l1[i] != l2[i]`: If `l1[i] < l2[i]`, `l1` is less than `l2`. If all values are equal, then the longer list sorts last.

NULL values are compared following PostgreSQL's semantics: At the top level, comparisons to NULL yield NULL; At lower nesting levels NULLs are greater than other values.

Here are some queries returning `true` for the comparison.

```
SELECT [1, 2] < [1, 3] AS result;
```

```
SELECT [[1], [2, 4, 5]] < [[2]] AS result;
```

```
SELECT [ ] < [1] AS result;
```

```
SELECT [1, 2] < [1, NULL, 4] AS result;
```

These queries return `false`.

```
SELECT [ ] < [ ] AS result;
```

```
SELECT [1, 2] < [1] AS result;
```

These queries return `NULL`.

```
SELECT NULL = [1] AS result;
```

Functions

See List Functions.

Literal Types

DuckDB has special literal types for representing `NULL`, integer and string literals in queries. These have their own binding and conversion rules.

> Prior to DuckDB version 0.10.0, integer and string literals behaved identically to the `INTEGER` and `VARCHAR` types.

Null Literals

The `NULL` literal is denoted with the keyword `NULL`. The `NULL` literal can be implicitly converted to any other type.

Integer Literals

Integer literals are denoted as a sequence of one or more decimal digits. At runtime, these result in values of the `INTEGER_LITERAL` type. `INTEGER_LITERAL` types can be implicitly converted to any integer type in which the value fits. For example, the integer literal `42` can be implicitly converted to a `TINYINT`, but the integer literal `1000` cannot be.

> DuckDB does not support hexadecimal or binary literals directly. However, strings or string literals in hexadecimal or binary notation with `0x` or `0b` prefixes respectively, can be cast to integer types, e.g., `'0xFF'::INT = 255` or `'0b101'::INT = 5`.

Other Numeric Literals

Non-integer numeric literals can be denoted with decimal notation, using the period character (`.`) to separate the integer part and the decimal part of the number. Either the integer part or the decimal part may be omitted:

```
SELECT 1.5;          -- 1.5
SELECT .50;          -- 0.5
SELECT 2.;           -- 2.0
```

Non-integer numeric literals can also be denoted using *E notation*. In E notation, an integer or decimal literal is followed by an exponential part, which is denoted by `e` or `E`, followed by a literal integer indicating the exponent. The exponential part indicates that the preceding value should be multiplied by 10 raised to the power of the exponent:

```
SELECT 1e2;           -- 100
SELECT 6.02214e23;    -- Avogadro's constant
SELECT 1e-10;         -- 1 ångström
```

Underscores in Numeric Literals

DuckDB's SQL dialect allows using the underscore character _ in numeric literals as an optional separator. The rules for using underscores are as follows:

- Underscores are allowed in integer, decimal, hexadecimal and binary notation. Hexadecimal notation uses the `0x` prefix and binary notation uses the `0b` prefix, as described in the Integer Literals section; for example, `'0xFF'::INTEGER` and `'0b101'::INTEGER`.
- Underscores cannot be the first or last character in a literal.
- Underscores have to have an integer/numeric part on either side of them, i.e., there cannot be multiple underscores in a row and underscores cannot appear immediately before or after a decimal or exponent.

Examples:

```
SELECT 100_000_000;          -- 100000000
SELECT '0xFF_FF'::INTEGER;   -- 65535
SELECT 1_2.1_2E0_1;          -- 121.2
SELECT '0b0_1_0_1'::INTEGER; -- 5
```

String Literals

String literals are delimited using single quotes (`'`, apostrophe) and result in `STRING_LITERAL` values. Note that double quotes (`"`) cannot be used as string delimiter character: instead, double quotes are used to delimit quoted identifiers.

Implicit String Literal Concatenation

Consecutive single-quoted string literals separated only by whitespace that contains at least one newline are implicitly concatenated:

```
SELECT 'Hello'
    ' '
    'World' AS greeting;
```

is equivalent to:

```
SELECT 'Hello'
    || ' '
    || 'World' AS greeting;
```

They both return the following result:

greeting
Hello World

Note that implicit concatenation only works if there is at least one newline between the literals. Using adjacent string literals separated by whitespace without a newline results in a syntax error:

```
SELECT 'Hello' ' ' 'World' AS greeting;
```

```
Parser Error:
syntax error at or near "' '"

LINE 1: SELECT 'Hello' ' ' 'World' AS greeting;
                       ^
```

Also note that implicit concatenation only works with single-quoted string literals, and does not work with other kinds of string values.

Implicit String Conversion

`STRING_LITERAL` instances can be implicitly converted to *any* other type.

For example, we can compare string literals with dates:

```
SELECT d > '1992-01-01' AS result
FROM (VALUES (DATE '1992-01-01')) t(d);
```

result
false

However, we cannot compare VARCHAR values with dates.

```
SELECT d > '1992-01-01'::VARCHAR
FROM (VALUES (DATE '1992-01-01')) t(d);
```

```
Binder Error:
Cannot compare values of type DATE and type VARCHAR - an explicit cast is required
```

Escape String Literals

To escape a single quote (apostrophe) character in a string literal, use `''`. For example, `SELECT '''' AS s` returns `'`.

To enable some common escape sequences, such as `\n` for the newline character, prefix a string literal with `e` (or `E`).

```
SELECT e'Hello\nworld' AS msg;
```

msg varchar
Hello\nworld

The following backslash escape sequences are supported:

Escape sequence	Name	ASCII code
`\b`	backspace	8
`\f`	form feed	12
`\n`	newline	10
`\r`	carriage return	13
`\t`	tab	9

Dollar-Quoted String Literals

DuckDB supports dollar-quoted string literals, which are surrounded by double-dollar symbols (`$$`):

```
SELECT $$Hello
world$$ AS msg;
```

msg varchar
Hello\nworld

```
SELECT $$The price is $9.95$$ AS msg;
```

msg
The price is $9.95

Even more, you can insert alphanumeric tags in the double-dollar symbols to allow for the use of regular double-dollar symbols *within* the string literal:

```
SELECT $tag$ this string can contain newlines,
'single quotes',
"double quotes",
and $$dollar quotes$$ $tag$ AS msg;
```

msg varchar
this string can contain newlines,\n'single quotes',\n"double quotes",\nand $$dollar quotes$$

Implicit concatenation only works for single-quoted string literals, not with dollar-quoted ones.

Bit String and Hex String Literals

> New Starting with DuckDB 2.0, bit string and hex string literals are supported. In earlier versions, `B'0101'` and `X'CAFE'` were parsed as the VARCHAR strings `'b0101'` and `'xCAFE'`, respectively.

A string literal prefixed with `B` is a bit string literal, which produces a `BITSTRING` value. It may only contain the characters `0` and `1`.

A string literal prefixed with `X` is a hex string literal, which produces a `BLOB` value. It must contain an even number of hexadecimal digits.

```
SELECT B'0101' AS b, X'CAFE' AS x;
```

b bit	x blob
0101	\xCA\xFE

Map Type

MAPs are similar to STRUCTs in that they are an ordered list of key-value pairs. However, MAPs do not need to have the same keys present for each row, and thus are suitable for use cases where the schema is unknown beforehand or varies per row.

MAPs must have a single type for all keys, and a single type for all values. Keys and values can be any type, and the type of the keys does not need to match the type of the values (e.g., a MAP of VARCHAR to INT is valid). MAPs may not have duplicate keys. MAPs return NULL if a key is not found rather than throwing an error as structs do.

In contrast, STRUCTs must have string keys, but each value may have a different type. See the data types overview for a comparison between nested data types.

To construct a MAP, use the bracket syntax preceded by the MAP keyword.

Creating Maps

A map with VARCHAR keys and INTEGER values. This returns {key1=10, key2=20, key3=30}:

```
SELECT MAP {'key1': 10, 'key2': 20, 'key3': 30};
```

Alternatively use the map_from_entries function. This returns {key1=10, key2=20, key3=30}:

```
SELECT map_from_entries([('key1', 10), ('key2', 20), ('key3', 30)]);
```

A map can be also created using two lists: keys and values. This returns {key1=10, key2=20, key3=30}:

```
SELECT MAP(['key1', 'key2', 'key3'], [10, 20, 30]);
```

A map can also use INTEGER keys and NUMERIC values. This returns {1=42.001, 5=-32.100}:

```
SELECT MAP {1: 42.001, 5: -32.1};
```

Keys and/or values can also be nested types. This returns {[a, b]=[1.1, 2.2], [c, d]=[3.3, 4.4]}:

```
SELECT MAP {['a', 'b']: [1.1, 2.2], ['c', 'd']: [3.3, 4.4]};
```

Create a table with a map column that has INTEGER keys and DOUBLE values:

```
CREATE TABLE tbl (col MAP(INTEGER, DOUBLE));
```

Retrieving from Maps

MAP values can be retrieved using the map_extract_value function or bracket notation:

```
SELECT MAP {'key1': 5, 'key2': 43}['key1'];
```

```
5
```

If the key has the wrong type, an error is thrown. If it has the correct type but is merely not contained in the map, a NULL value is returned:

```
SELECT MAP {'key1': 5, 'key2': 43}['key3'];
```

```
NULL
```

The map_extract function (and its synonym element_at) can be used to retrieve a value wrapped in a list; it returns an empty list if the key is not contained in the map:

```
SELECT map_extract(MAP {'key1': 5, 'key2': 43}, 'key1');
```

```
[5]
```

```
SELECT map_extract(MAP {'key1': 5, 'key2': 43}, 'key3');
```

```
[]
```

Comparison Operators

Nested types can be compared using all the comparison operators. These comparisons can be used in logical expressions for both WHERE and HAVING clauses, as well as for creating Boolean values.

The ordering is defined positionally in the same way that words can be ordered in a dictionary. NULL values compare greater than all other values and are considered equal to each other.

At the top level, NULL nested values obey standard SQL NULL comparison rules: comparing a NULL nested value to a non-NULL nested value produces a NULL result. Comparing nested value *members*, however, uses the internal nested value rules for NULLs, and a NULL nested value member will compare above a non-NULL nested value member.

Functions

See Map Functions.

NULL Values

`NULL` values are special values that are used to represent missing data in SQL. Columns of any type can contain `NULL` values. Logically, a `NULL` value can be seen as "the value of this field is unknown".

A `NULL` value can be inserted to any field that does not have the `NOT NULL` qualifier:

```
CREATE TABLE integers (i INTEGER);
INSERT INTO integers VALUES (NULL);
```

`NULL` values have special semantics in many parts of the query as well as in many functions:

> Any comparison with a `NULL` value returns `NULL`, including `NULL = NULL`.

You can use `IS NOT DISTINCT FROM` to perform an equality comparison where `NULL` values compare equal to each other. Use `IS (NOT) NULL` to check if a value is `NULL`.

```
SELECT NULL = NULL;
```

```
NULL
```

```
SELECT NULL IS NOT DISTINCT FROM NULL;
```

```
true
```

```
SELECT NULL IS NULL;
```

```
true
```

NULL and Functions

A function that has an input argument as `NULL` **usually** returns `NULL`.

```
SELECT cos(NULL);
```

```
NULL
```

The `coalesce` function is an exception to this: it takes any number of arguments, and returns for each row the first argument that is not `NULL`. If all arguments are `NULL`, `coalesce` also returns `NULL`.

```
SELECT coalesce(NULL, NULL, 1);
```

```
1
```

```
SELECT coalesce(10, 20);
```

```
10
```

```
SELECT coalesce(NULL, NULL);
```

```
NULL
```

The `ifnull` function is a two-argument version of `coalesce`.

```
SELECT ifnull(NULL, 'default_string');
```

```
default_string
```

```
SELECT ifnull(1, 'default_string');
```

```
1
```

NULL and AND / OR

NULL values have special behavior when used with AND and OR. For details, see the Boolean Type documentation.

NULL and IN / NOT IN

The behavior of `... IN <something with a NULL>`{:.language-sql .highlight} is different from `... IN <something with no NULLs>`{:.language-sql .highlight}. For details, see the IN documentation.

NULL and Aggregate Functions

NULL values are ignored in most aggregate functions.

Aggregate functions that do not ignore NULL values include: `first`, `last`, `list` and `array_agg`. To exclude NULL values from these results, use a `FILTER` clause with a predicate that excludes the relevant null rows, such as `list(x) FILTER (WHERE x IS NOT NULL)`. A different filter predicate does not remove NULL values from rows that pass it.

```
CREATE TABLE integers (i INTEGER);
INSERT INTO integers VALUES (1), (10), (NULL);
```

```
SELECT min(i) FROM integers;
```

```
1
```

```
SELECT max(i) FROM integers;
```

```
10
```

Numeric Types

Fixed-Width Integer Types

The types TINYINT, SMALLINT, INTEGER, BIGINT and HUGEINT store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error. The types UTINYINT, USMALLINT, UINTEGER, UBIGINT and UHUGEINT store whole unsigned numbers. Attempts to store negative numbers or values outside of the allowed range will result in an error.

Name	Aliases	Min	Max	Size in bytes
TINYINT	INT1	-2^7	2^7 - 1	1
SMALLINT	INT2, INT16, SHORT	-2^15	2^15 - 1	2
INTEGER	INT4, INT32, INT, SIGNED	-2^31	2^31 - 1	4
BIGINT	INT8, INT64, LONG	-2^63	2^63 - 1	8
HUGEINT	INT128	-2^127	2^127 - 1	16
UTINYINT	UINT8	0	2^8 - 1	1
USMALLINT	UINT16	0	2^16 - 1	2
UINTEGER	UINT32	0	2^32 - 1	4
UBIGINT	UINT64	0	2^64 - 1	8
UHUGEINT	UINT128	0	2^128 - 1	16

> `INT8` is a 64-bit integer, and is not the signed equivalent of `UINT8`, an unsigned, 8-bit integer. The type aliases `INT1`, `INT2`, `INT4` and `INT8` for signed integers were inherited from PostgreSQL, where digits in these names indicate their size in *bytes*, whereas the type aliases for their unsigned equivalents, `UINT8`, `UINT16`, `UINT32` and `UINT64`, indicate their size in *bits* following the C/C++ convention.

The type integer is the common choice, as it offers the best balance between range, storage size, and performance. The `SMALLINT` type is generally only used if disk space is at a premium. The `BIGINT` and `HUGEINT` types are designed to be used when the range of the integer type is insufficient.

Variable-Length Integers

The previously mentioned integer types all have in common that the numbers in the minimum and maximum range all have the same storage size, `UTINYINT` is 1 byte, `SMALLINT` is 2 bytes, etc. But sometimes you need numbers that are even bigger than what is supported by a `HUGEINT`! In these situations, you can use the `BIGNUM` type, which stores positive numbers in a similar fashion as other integer types, but uses three additional bytes to store the required size and a sign bit. A number with `N` decimal digits requires approximately `0.415 * N + 3` bytes when stored in a `BIGNUM`.

Unlike variable-length integer implementations in other systems, there are limits to `BIGNUM`: the maximal and minimal representable values are approximately `±4.27e20201778`. Those are numbers with 20,201,779 decimal digits and storing a single such number requires 8 megabytes.

Fixed-Point Decimals

The data type `DECIMAL(WIDTH, SCALE)` (also available under the alias `NUMERIC(WIDTH, SCALE)`) represents an exact fixed-point decimal value. When creating a value of type `DECIMAL`, the `WIDTH` and `SCALE` can be specified to define which size of decimal values can be held in the field. The `WIDTH` field determines how many digits can be held, and the `scale` determines the number of digits after the decimal point. For example, the type `DECIMAL(3, 2)` can fit the value `1.23`, but cannot fit the value `12.3` or the value `1.234`. The default `WIDTH` and `SCALE` is `DECIMAL(18, 3)`, if none are specified. If only the `WIDTH` is specified, the `SCALE` defaults to `0` – for example, `DECIMAL(18)` is equivalent to `DECIMAL(18, 0)` and rounds to whole numbers (e.g., `1.5::DECIMAL(18)` is 2).

Specifying the `WIDTH` and the `SCALE`

Both parameters are optional, and a trailing comma after the `WIDTH` is accepted. The following spellings therefore all bind to a concrete decimal type:

```
SELECT
    typeof(1.5::DECIMAL)      AS "DECIMAL",
    typeof(1.5::DECIMAL())    AS "DECIMAL()",
    typeof(1.5::DECIMAL(10))  AS "DECIMAL(10)",
    typeof(1.5::DECIMAL(10,)) AS "DECIMAL(10,)";
```

DECIMAL	DECIMAL()	DECIMAL(10)	DECIMAL(10,)
DECIMAL(18,3)	DECIMAL(18,3)	DECIMAL(10,0)	DECIMAL(10,0)

The `NUMERIC` alias accepts the same spellings and yields the same types.

Note that a bare `DECIMAL` is not an unconstrained decimal type as it is in PostgreSQL and Oracle. DuckDB always binds it to the concrete type `DECIMAL(18, 3)`, which rounds to 3 fractional digits and holds at most 15 digits before the decimal point.

The `WIDTH` must be between 1 and 38, and the `SCALE` must be between 0 and 38 and may not exceed the `WIDTH`. Values outside these bounds are rejected by the binder instead of being clamped:

```
SELECT 1::DECIMAL(39, 2);
```

```
Binder Error:
DECIMAL type width must be between 1 and 38
```

```
SELECT 1::DECIMAL(18, 20);
```

```
Binder Error:
DECIMAL type scale cannot be greater than width
```

Unlike PostgreSQL and Oracle, DuckDB does not support a negative `SCALE`:

```
SELECT 1::DECIMAL(18, -1);
```

```
Binder Error:
Type "DECIMAL" does not accept type parameters (INTEGER, INTEGER)
```

Arithmetic and Internal Representation

Addition, subtraction and multiplication of two fixed-point decimals returns another fixed-point decimal with the required `WIDTH` and `SCALE` to contain the exact result, or throws an error if the required `WIDTH` would exceed the maximal supported `WIDTH`, which is currently 38.

Division of fixed-point decimals does not typically produce numbers with finite decimal expansion. Therefore, DuckDB uses approximate floating-point arithmetic for all divisions that involve fixed-point decimals and accordingly returns floating-point data types.

Internally, decimals are represented as integers depending on their specified `WIDTH`.

Width	Internal	Size (bytes)
1-4	`INT16`	2
5-9	`INT32`	4
10-18	`INT64`	8
19-38	`INT128`	16

Performance can be impacted by using too large decimals when not required. In particular, decimal values with a width above 19 are slow, as arithmetic involving the `INT128` type is much more expensive than operations involving the `INT32` or `INT64` types. It is therefore recommended to stick with a `WIDTH` of 18 or below, unless there is a good reason for why this is insufficient.

Floating-Point Types

The data types `FLOAT` and `DOUBLE` precision are variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Name	Aliases	Description
`FLOAT`	`FLOAT4`, `REAL`	Single precision floating-point number (4 bytes)
`DOUBLE`	`FLOAT8`	Double precision floating-point number (8 bytes)

Like for fixed-point data types, conversion from literals or casts from other datatypes to floating-point types stores inputs that cannot be represented exactly as approximations. However, it can be harder to predict what inputs are affected by this. For example, it is not surprising that `1.3::DECIMAL(1, 0) - 0.7::DECIMAL(1, 0) != 0.6::DECIMAL(1, 0)` but it may be surprising that `1.3::FLOAT - 0.7::FLOAT != 0.6::FLOAT`.

Additionally, whereas multiplication, addition and subtraction of fixed-point decimal data types is exact, these operations are only approximate on floating-point binary data types.

For more complex mathematical operations, however, floating-point arithmetic is used internally and more precise results can be obtained if intermediate steps are *not* cast to fixed point formats of the same width as in- and outputs. For example, `(10::FLOAT / 3::FLOAT)::FLOAT * 3 = 10` whereas `(10::DECIMAL(18, 3) / 3::DECIMAL(18, 3))::DECIMAL(18, 3) * 3 = 9.999`.

In general:

- If you require exact storage of numbers with a known number of decimal digits and require exact additions, subtractions and multiplications (such as for monetary amounts), use the DECIMAL data type or its `NUMERIC` alias instead.
- If you want to do fast or complicated calculations, the floating-point data types may be more appropriate. However, if you use the results for anything important, you should evaluate your implementation carefully for corner cases (ranges, infinities, underflows, invalid operations) that may be handled differently from what you expect and you should familiarize yourself with common floating-point pitfalls. The article "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg and the floating point series on Bruce Dawson's blog provide excellent starting points.

On most platforms, the `FLOAT` type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The `DOUBLE` type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Positive numbers outside of these ranges (and negative numbers outside the mirrored ranges) may cause errors on some platforms but will usually be converted to zero or infinity, respectively.

In addition to ordinary numeric values, the floating-point types have several special values representing IEEE 754 special values:

- `Infinity`: infinity
- `-Infinity`: negative infinity
- NaN: not a number

On machines with the required CPU/FPU support, DuckDB follows the IEEE 754 specification regarding these special values, with one exception:

- NaN compares equal to NaN and greater than any other floating point number.

To insert these values as literals in a SQL command, you must put quotes around them, you may abbreviate `Infinity` as `Inf`, and you may use any capitalization. For example:

```
SELECT
    sqrt(2) > '-inf',
    'nan' > sqrt(2);
```

(sqrt(2) > '-inf')	('nan' > sqrt(2))
true	true

Universally Unique Identifiers (UUIDs)

DuckDB supports universally unique identifiers (UUIDs) through the `UUID` type. These use 128 bits and are represented internally as `HUGEINT` values. When printed, they are shown with lowercase hexadecimal characters, separated by dashes as follows: `<12345678>-<1234>-<1234>-<1234>-<1234567890ab>{:.language-sql .highlight}` (using 36 characters in total including the dashes). For example, `4ac7a9e9-607c-4c8a-84f3-843f0191e3fd` is a valid UUID.

DuckDB supports generating UUIDv4 and UUIDv7 identifiers. To retrieve the version of a UUID value, use the `uuid_extract_version` function.

UUIDv4

To generate a UUIDv4 value, use the `uuid()` function or its aliases the `uuidv4()` and `gen_random_uuid()` functions.

UUIDv7

To generate a UUIDv7 value, use the `uuidv7()` function. To retrieve the timestamp from a UUIDv7 value, use the `uuid_extract_timestamp` function:

```
SELECT uuid_extract_timestamp(uuidv7()) AS ts;
```

ts
2025-04-19 15:51:20.07+00

Rounding

Casting from floating-point or fixed-point types to integers means that the source numbers are rounded.

The following default rounding logic applies:

- Casting from `FLOAT` and `DOUBLE` to integers of any size: round to the nearest integer, with ties (halfs) rounded to the nearest even number (banker's rounding).

  ```
  SELECT 2.5::FLOAT::INTEGER AS result;  -- round to even
  ```

result int32
2

- Casting from `DECIMAL` to integers of any size: round to the nearest integer, with ties (halfs) rounded away from zero.

  ```
  SELECT 2.5::DECIMAL::INTEGER AS result;  -- round away from zero
  SELECT 2.5::INTEGER AS result;           -- number literals are treated as DECIMALs
  ```

result int32
3

If non-default rounding logic is required, functions like `ceil`, `floor`, `round` and `round_even` should be used.

Functions

See Numeric Functions and Operators.

Struct Data Type

Conceptually, a `STRUCT` column contains an ordered list of columns called "entries". The entries are referenced by name using strings. This document refers to those entry names as keys. Each row in the `STRUCT` column must have the same keys. The names of the struct entries are part of the *schema*. Each row in a `STRUCT` column must have the same layout. The names of the struct entries are case-insensitive.

`STRUCT`s are typically used to nest multiple columns into a single column, and the nested column can be of any type, including other `STRUCT`s and `LIST`s.

STRUCTs are similar to PostgreSQL's ROW type. The key difference is that DuckDB STRUCTs require the same keys in each row of a STRUCT column. This allows DuckDB to provide significantly improved performance by fully utilizing its vectorized execution engine, and also enforces type consistency for improved correctness. Structs with unnamed entries are represented by the TUPLE type, which can be created using the `row` function or the `(expr, ...)` notation. See an example below and the STRUCT functions documentation for details.

See the data types overview for a comparison between nested data types.

Creating Structs

Structs can be created using the `struct_pack(name := expr, ...)` function, the equivalent array notation `{'name': expr, ...}`, or using a row variable. To create a struct with unnamed entries, use a tuple.

Create a struct using the `struct_pack` function. Note the lack of single quotes around the keys and the use of the `:=` operator:

```
SELECT struct_pack(key1 := 'value1', key2 := 42) AS s;
```

Create a struct using the array notation:

```
SELECT {'key1': 'value1', 'key2': 42} AS s;
```

Create a struct using a row variable:

```
SELECT d AS s FROM (SELECT 'value1' AS key1, 42 AS key2) d;
```

Create a struct of integers:

```
SELECT {'x': 1, 'y': 2, 'z': 3} AS s;
```

Create a struct of strings with a `NULL` value:

```
SELECT {'yes': 'duck', 'maybe': 'goose', 'huh': NULL, 'no': 'heron'} AS s;
```

Create a struct with a different type for each key:

```
SELECT {'key1': 'string', 'key2': 1, 'key3': 12.345} AS s;
```

Create a struct of structs with `NULL` values:

```
SELECT {
        'birds': {'yes': 'duck', 'maybe': 'goose', 'huh': NULL, 'no': 'heron'},
        'aliens': NULL,
        'amphibians': {'yes': 'frog', 'maybe': 'salamander', 'huh': 'dragon', 'no': 'toad'}
    } AS s;
```

> New As of DuckDB 2.0, STRUCT types may be empty. The bare type name `STRUCT` denotes an empty struct, so both `NULL::STRUCT` and `CREATE TABLE t (s STRUCT)` are valid, and `struct_pack()` with no arguments returns `{}` (whereas `row()` with no arguments returns the empty tuple `()`). Note that `STRUCT()` with empty parentheses remains a syntax error, and `unnest` and `struct_extract` raise an error on an empty struct. Persisting an empty struct into a pre-2.0 database format is not supported.

```
SELECT struct_pack() AS s;
```

Adding or Updating Fields of Structs

To add new fields or update existing ones, you can use `struct_update`:

```
SELECT struct_update({'a': 1, 'b': 2}, b := 3, c := 4) AS s;
```

Alternatively, `struct_insert` also allows adding new fields but not updating existing ones.

Retrieving from Structs

Retrieving a value from a struct can be accomplished using dot notation, bracket notation, or through struct functions like `struct_extract`.

Use dot notation to retrieve the value at a key's location. In the following query, the subquery generates a struct column `a`, which we then query with `a.x`.

```
SELECT a.x FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
```

If a key contains a space, simply wrap it in double quotes (").

```
SELECT a."x space" FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
```

Bracket notation may also be used. Note that this uses single quotes (') since the goal is to specify a certain string key and only constant expressions may be used inside the brackets (no expressions):

```
SELECT a['x space'] FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
```

The `struct_extract` function is also equivalent. This returns 1:

```
SELECT struct_extract({'x space': 1, 'y': 2, 'z': 3}, 'x space');
```

`unnest` / `STRUCT.*`

Rather than retrieving a single key from a struct, the `unnest` special function can be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys:

```
SELECT unnest(a)
FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
```

x	y	z
1	2	3

The same can be achieved with the star notation (`*`), which additionally allows modifications of the returned columns:

```
SELECT a.* EXCLUDE ('y')
FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
```

x	z
1	3

> **Warning.** The star notation is currently limited to top-level struct columns and non-aggregate expressions.

Dot Notation Order of Operations

Referring to structs with dot notation can be ambiguous with referring to schemas and tables. In general, DuckDB looks for columns first, then for struct keys within columns. DuckDB resolves references in these orders, using the first match to occur:

No Dots

```
SELECT part1
FROM tbl;
```

1. `part1` is a column

One Dot

```
SELECT part1.part2
FROM tbl;
```

1. `part1` is a table, `part2` is a column
2. `part1` is a column, `part2` is a property of that column

Two (or More) Dots

```
SELECT part1.part2.part3
FROM tbl;
```

1. `part1` is a schema, `part2` is a table, `part3` is a column
2. `part1` is a table, `part2` is a column, `part3` is a property of that column
3. `part1` is a column, `part2` is a property of that column, `part3` is a property of that column

Any extra parts (e.g., `.part4.part5`, etc.) are always treated as properties

Tuples

> New The TUPLE type was introduced in DuckDB 2.0. In earlier versions, unnamed structs were STRUCT values with empty keys.

A TUPLE is a struct whose entries have no names. Its entries are referenced by their (1-based) position. Tuples are created using the `row` function or, if there are at least two expressions, using parentheses:

```
SELECT row(x, x + 1, y) AS t FROM (SELECT 1 AS x, 'a' AS y) AS s;
```

```
┌──────────────────────────────────┐
│                t                 │
│ tuple(integer, integer, varchar) │
├──────────────────────────────────┤
│ (1, 2, a)                        │
└──────────────────────────────────┘
```

If using multiple expressions, the `row` function is optional. The following query returns the same result as the previous one:

```
SELECT (x, x + 1, y) AS t FROM (SELECT 1 AS x, 'a' AS y);
```

A single-element tuple can be created with `row(x)` or with a trailing comma, `(x,)`.

The type of a tuple is written as `TUPLE(<type_1>, ..., <type_n>)`:

```
SELECT typeof((1, 'a')) AS type;
```

```
TUPLE(INTEGER, VARCHAR)
```

Accessing Tuple Entries

Tuple entries are accessed by position using the bracket notation or the `struct_extract` and `struct_extract_at` functions. Dot notation is not supported, as the entries have no names:

```
SELECT t[1] AS first, struct_extract_at(t, 2) AS second
FROM (SELECT (42, 'duck') AS t);
```

```
┌─────────┬─────────┐
│  first  │ second  │
│  int32  │ varchar │
├─────────┼─────────┤
│      42 │ duck    │
└─────────┴─────────┘
```

Tuples in Tables

Tuples can be stored in tables, either by declaring a `TUPLE` column or by using `CREATE TABLE ... AS`:

```
CREATE TABLE t2 AS SELECT row('a') AS t;
SELECT column_name, column_type FROM (DESCRIBE t2);
```

column_name varchar	column_type varchar
t	TUPLE(VARCHAR)

When written to file formats without a tuple type, such as Parquet, tuples are stored as structs with the generated keys `element1`, `element2`, etc.

Casting between Tuples and Structs

Tuples can be cast to structs with the same number of entries, and vice versa. The entries are matched by position:

```
SELECT
    (1, 'a')::STRUCT(id INTEGER, name VARCHAR) AS s,
    {'id': 1, 'name': 'a'}::TUPLE(INTEGER, VARCHAR) AS t;
```

s struct(id integer, "name" varchar)	t tuple(integer, varchar)
{'id': 1, 'name': a}	(1, a)

This allows inserting tuples into STRUCT columns:

```
CREATE TABLE t1 (s STRUCT(v VARCHAR, i INTEGER));
INSERT INTO t1 VALUES (row('a', 42));
SELECT * FROM t1;
```

The table will contain a single entry:

```
{'v': a, 'i': 42}
```

The following produces the same result as above:

```
CREATE TABLE t1 AS (
    SELECT row('a', 42)::STRUCT(v VARCHAR, i INTEGER) AS s
);
```

Casting between a tuple and a struct with a different number of entries fails.

Casting between Structs

When casting between structs, the names of at least one field have to match. Therefore, the following query will fail:

```
SELECT a::STRUCT(y INTEGER) AS b
FROM
    (SELECT {'x': 42} AS a);
```

```
Binder Error:
STRUCT to STRUCT cast must have at least one matching member
```

A workaround for this is to use `struct_pack` instead:

```
SELECT struct_pack(y := a.x) AS b
FROM
    (SELECT {'x': 42} AS a);
```

Comparison and Ordering

The `STRUCT` type can be compared using all the comparison operators. These comparisons can be used in logical expressions such as `WHERE` and `HAVING` clauses, and return `BOOLEAN` values.

Comparisons are done in lexicographical order over the entries, following PostgreSQL's `IS NOT DISTINCT FROM` semantics: corresponding entries are compared position by position, and two `NULL` entries in the same position are considered equal (not distinct) so the comparison moves on to the next position. For ordering purposes, a `NULL` entry is treated as larger than any non-`NULL` value.

Specifically:

- If every entry of `s1` is not distinct from the corresponding entry of `s2` (treating two `NULL`s as equal), then `s1` and `s2` compare equal.
- else, for the first index `i` where `s1.value[i]` is distinct from `s2.value[i]`, if `s1.value[i] < s2.value[i] OR s2.value[i] IS NULL`, then `s1` is less than `s2`, and vice versa.

Structs of different types are implicitly cast to a struct type with the union of the involved keys, following the rules for combination casting.

The following queries return `true`:

```
SELECT {'k1': 0, 'k2': 0} < {'k1': 1, 'k2': 0};
```

```
SELECT {'k1': 'hello'} < {'k1': 'world'};
```

```
SELECT {'k1': 0, 'k2': 0} < {'k1': 0, 'k2': NULL};
```

```
SELECT {'k1': 0} < {'k2': 0};
```

```
SELECT  {'k1': 0, 'k2': 0} < {'k2': 0, 'k3': 0};
```

```
SELECT {'k1': 1, 'k2': 0} > {'k3': 0, 'k1': 0};
```

The following queries return `false`:

```
SELECT {'k1': 1, 'k2': 0} < {'k1': 0, 'k2': 1};
```

```
SELECT {'k1': [0]} < {'k1': [0, 0]};
```

```
SELECT {'k1': 1} > {'k2': 0};
```

```
SELECT {'k1': 1, 'k2': 0} < {'k3': 0, 'k1': 0};
```

```
SELECT  {'k1': 1, 'k2': 0} > {'k2': 0, 'k3': 0};
```

Updating the Schema

Starting with DuckDB 1.3.0, it's possible to update the sub-schema of structs using the `ALTER TABLE` clause.

To follow the examples, initialize the `test` table as follows:

```
CREATE TABLE test (s STRUCT(i INTEGER, j INTEGER));
INSERT INTO test VALUES (ROW(1, 1)), (ROW(2, 2));
```

Adding a Field

Add field `k INTEGER` to struct `s` in table `test`:

```
ALTER TABLE test ADD COLUMN s.k INTEGER;
FROM test;
```

```
┌──────────────────────────────────────────┐
│                    s                     │
│ struct(i integer, j integer, k integer)  │
├──────────────────────────────────────────┤
│ {'i': 1, 'j': 1, 'k': NULL}              │
│ {'i': 2, 'j': 2, 'k': NULL}              │
└──────────────────────────────────────────┘
```

Dropping a Field

Drop field `i` from struct `s` in table `test`:

```
ALTER TABLE test DROP COLUMN s.i;
FROM test;
```

```
┌──────────────────────────────┐
│              s               │
│ struct(j integer, k integer) │
├──────────────────────────────┤
│ {'j': 1, 'k': NULL}          │
│ {'j': 2, 'k': NULL}          │
└──────────────────────────────┘
```

Renaming a Field

Renaming field `j` of struct `s` to `v1` in table `test`:

```
ALTER TABLE test RENAME s.j TO v1;
FROM test;
```

```
┌──────────────────────────────┐
│              s               │
│ struct(v1 integer, k integer)│
├──────────────────────────────┤
│ {'v1': 1, 'k': NULL}         │
│ {'v1': 2, 'k': NULL}         │
└──────────────────────────────┘
```

Functions

See Struct Functions.

Text Types

In DuckDB, strings can be stored in the VARCHAR field. The field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Name	Aliases	Description
VARCHAR	CHAR, BPCHAR, STRING, TEXT	Variable-length character string
VARCHAR(n)	CHAR(n), BPCHAR(n), STRING(n), TEXT(n)	Variable-length character string. The maximum length n has no effect and is only provided for compatibility

Specifying a Length Limit

Specifying the length for the VARCHAR, STRING and TEXT types is not required and has no effect on the system. Specifying the length will not improve performance or reduce storage space of the strings in the database. These variants are supported for compatibility with other systems that do require a length to be specified for strings.

If you wish to restrict the number of characters in a VARCHAR column for data integrity reasons the CHECK constraint should be used, for example:

```
CREATE TABLE strings (
    val VARCHAR CHECK (length(val) <= 10) -- val has a maximum length of 10
);
```

The VARCHAR field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Specifying a Compression Type

You can specify a compression type for a string with the `USING COMPRESSION` clause. For example, to apply zstd compression, run:

```
CREATE TABLE tbl (s VARCHAR USING COMPRESSION zstd);
```

Text Type Values

Values of the text type are character strings, also known as string values or simply strings. At runtime, string values are constructed in one of the following ways:

- referencing columns whose declared or implied type is the text data type
- string literals
- casting expressions to a text type
- applying a string operator, or invoking a function that returns a text type value

Strings with Special Characters

To use special characters in a string, use escape string literals or dollar-quoted string literals. Alternatively, you can use concatenation and the `chr` character function:

```
SELECT 'Hello' || chr(10) || 'world' AS msg;
```

msg
varchar
Hello\nworld

Functions

See Text Functions and Pattern Matching.

Time Types

The `TIME` and `TIMETZ` types specify the hour, minute, second, microsecond of a day.

Name	Aliases	Description
`TIME`	`TIME WITHOUT TIME ZONE`	Time of day
`TIMETZ`	`TIME WITH TIME ZONE`	Time of day, with time zone offset
`TIME_NS`		Time of day, nanosecond precision

Instances can be created using the type names as a keyword, where the data must be formatted according to the ISO 8601 format (`hh:mm:ss[.zzzzzz[zzz]][+-TT[:tt]]`).

```
SELECT TIME '1992-09-20 11:30:00.123456';
```

```
11:30:00.123456
```

```
SELECT TIMETZ '1992-09-20 11:30:00.123456';
```

```
11:30:00.123456+00
```

```
SELECT TIMETZ '1992-09-20 11:30:00.123456-02:00';
```

```
13:30:00.123456+00
```

```
SELECT TIMETZ '1992-09-20 11:30:00.123456+05:30';
```

```
06:00:00.123456+00
```

```
SELECT '15:30:00.123456789'::TIME_NS;
```

```
15:30:00.123456789
```

TIME_NS values can also be read from Parquet when the type is TIME with unit NANOS.

> **Warning.** The TIME type should only be used in rare cases, where the date part of the timestamp can be disregarded. Most applications should use the TIMESTAMP types to represent their timestamps.

Timestamp Types

Timestamps represent points in time. As such, they combine DATE and TIME information. They can be created using the type name followed by a string formatted according to the ISO 8601 format, YYYY-MM-DD hh:mm:ss[.zzzzzzzzz][+-TT[:tt]], which is also the format we use in this documentation. Decimal places beyond the supported precision are ignored.

Timestamp Types

Name	Aliases	Description
TIMESTAMP_NS		Naive timestamp with nanosecond precision
TIMESTAMP	DATETIME, TIMESTAMP WITHOUT TIME ZONE	Naive timestamp with microsecond precision
TIMESTAMP_MS		Naive timestamp with millisecond precision
TIMESTAMP_S		Naive timestamp with second precision
TIMESTAMPTZ	TIMESTAMP WITH TIME ZONE	Time zone aware timestamp with microsecond precision
TIMESTAMPTZ_NS		Time zone aware timestamp with nanosecond precision

> New The TIMESTAMPTZ_NS type was introduced in DuckDB 2.0. It provides time zone aware timestamps with nanosecond precision. External columns with nanosecond precision and WITH TIME ZONE semantics, e.g., Parquet timestamp columns with isAdjustedToUTC=true, are now read as TIMESTAMPTZ_NS and retain their full precision. In earlier versions these columns were converted to TIMESTAMP WITH TIME ZONE and lost precision.

```
SELECT TIMESTAMP_NS '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456789
```

```
SELECT TIMESTAMP '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456
```

```
SELECT TIMESTAMP_MS '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123
```

```
SELECT TIMESTAMP_S '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00
```

```
SELECT TIMESTAMPTZ '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456+00
```

```
SELECT TIMESTAMPTZ '1992-09-20 12:30:00.123456789+01:00';
```

```
1992-09-20 11:30:00.123456+00
```

Unlike `TIMESTAMPTZ`, the `TIMESTAMPTZ_NS` type retains nanosecond precision:

```
SELECT TIMESTAMPTZ_NS '1992-09-20 11:30:00.123456789+00';
```

```
1992-09-20 11:30:00.123456789+00
```

DuckDB distinguishes timestamps `WITHOUT TIME ZONE` and `WITH TIME ZONE`. The time zone aware types are `TIMESTAMPTZ` (microsecond precision) and `TIMESTAMPTZ_NS` (nanosecond precision).

Despite the name, a `TIMESTAMP WITH TIME ZONE` does not store time zone information. Instead, it only stores the `INT64` number of non-leap microseconds since the Unix epoch `1970-01-01 00:00:00+00`, and thus unambiguously identifies a point in absolute time, or *instant*. The reason for the labels *time zone aware* and `WITH TIME ZONE` is that timestamp arithmetic, *binning*, and string formatting for this type are performed in a configured time zone, which defaults to the system time zone and is just `UTC+00:00` in the examples above.

The corresponding `TIMESTAMP WITHOUT TIME ZONE` stores the same `INT64`, but arithmetic, binning and string formatting follow the straightforward rules of Coordinated Universal Time (UTC) without offsets or time zones. Accordingly, `TIMESTAMP`s could be interpreted as UTC timestamps, but more commonly they are used to represent *local* observations of time recorded in an unspecified time zone, and operations on these types can be interpreted as simply manipulating tuple fields following nominal temporal logic. It is a common data cleaning problem to disambiguate such observations, which may also be stored in raw strings without time zone specification or UTC offsets, into unambiguous `TIMESTAMP WITH TIME ZONE` instants. One possible solution to this is to append UTC offsets to strings, followed by an explicit cast to `TIMESTAMP WITH TIME ZONE`. Alternatively, a `TIMESTAMP WITHOUT TIME ZONE` may be created first and then be combined with a time zone specification to obtain a time zone aware `TIMESTAMP WITH TIME ZONE`.

Conversion between Strings and Naïve / Time Zone-Aware Timestamps

The conversion between strings *without* UTC offsets or IANA time zone names and `WITHOUT TIME ZONE` types is unambiguous and straightforward. The conversion between strings *with* UTC offsets or time zone names and `WITH TIME ZONE` types is also unambiguous, but requires the `ICU` extension to handle time zone names.

When strings *without* UTC offsets or time zone names are converted to a `WITH TIME ZONE` type, the string is interpreted in the configured time zone. When strings with UTC offsets are passed to a `WITHOUT TIME ZONE` type, the offsets or time zone specifications are ignored. Starting with DuckDB 2.0, time zone names in strings passed to a `WITHOUT TIME ZONE` type are also ignored (in earlier versions, names other than UTC raised an error).

Finally, when `WITH TIME ZONE` and `WITHOUT TIME ZONE` types are converted to each other via explicit or implicit casts, the translation uses the configured time zone. To use an alternative time zone, the `timezone` function provided by the `ICU` extension may be used:

```
SELECT
    timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40') AS aware1,
    timezone('America/Denver', TIMESTAMPTZ '2001-02-16 04:38:40') AS naive1,
    timezone('UTC', TIMESTAMP '2001-02-16 20:38:40+00:00') AS aware2,
    timezone('UTC', TIMESTAMPTZ '2001-02-16 04:38:40 Europe/Berlin') AS naive2;
```

aware1	naive1	aware2	naive2
2001-02-17 04:38:40+01	2001-02-15 20:38:40	2001-02-16 21:38:40+01	2001-02-16 03:38:40

Note that `TIMESTAMP`s are displayed without time zone specification in the results, following ISO 8601 rules for local times, while time-zone aware `TIMESTAMPTZ`s are displayed with the UTC offset of the configured time zone, which is `'Europe/Berlin'` in the example. The UTC offsets of `'America/Denver'` and `'Europe/Berlin'` at all involved instants are `-07:00` and `+01:00`, respectively.

Special Values

Three special strings can be used to create timestamps:

Input string	Description
`epoch`	1970-01-01 00:00:00[+00] (Unix system time zero)
`infinity`	Later than all other timestamps
`-infinity`	Earlier than all other timestamps

The values `infinity` and `-infinity` are special cased and are displayed unchanged, whereas the value `epoch` is simply a notational shorthand that is converted to the corresponding timestamp value when read.

```
SELECT '-infinity'::TIMESTAMP, 'epoch'::TIMESTAMP, 'infinity'::TIMESTAMP;
```

Negative	Epoch	Positive
-infinity	1970-01-01 00:00:00	infinity

Functions

See Timestamp Functions.

Time Zones

To understand time zones and the `WITH TIME ZONE` types, it helps to start with two concepts: *instants* and *temporal binning*.

Instants

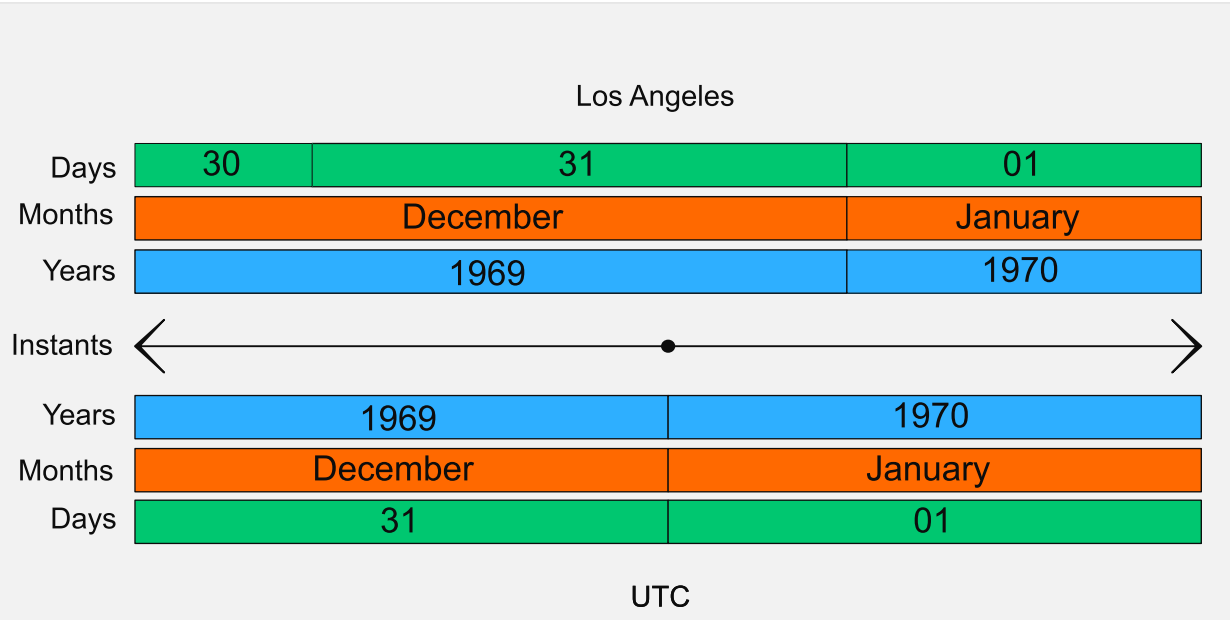
An instant is a point in absolute time, usually given as a count of some time increment from a fixed point in time (called the *epoch*). This is similar to how positions on the earth's surface are given using latitude and longitude relative to the equator and the Greenwich Meridian. In DuckDB, the fixed point is the Unix epoch `1970-01-01 00:00:00+00:00`, and the increment is in seconds, milliseconds, microseconds, or nanoseconds, depending on the specific data type.

Temporal Binning

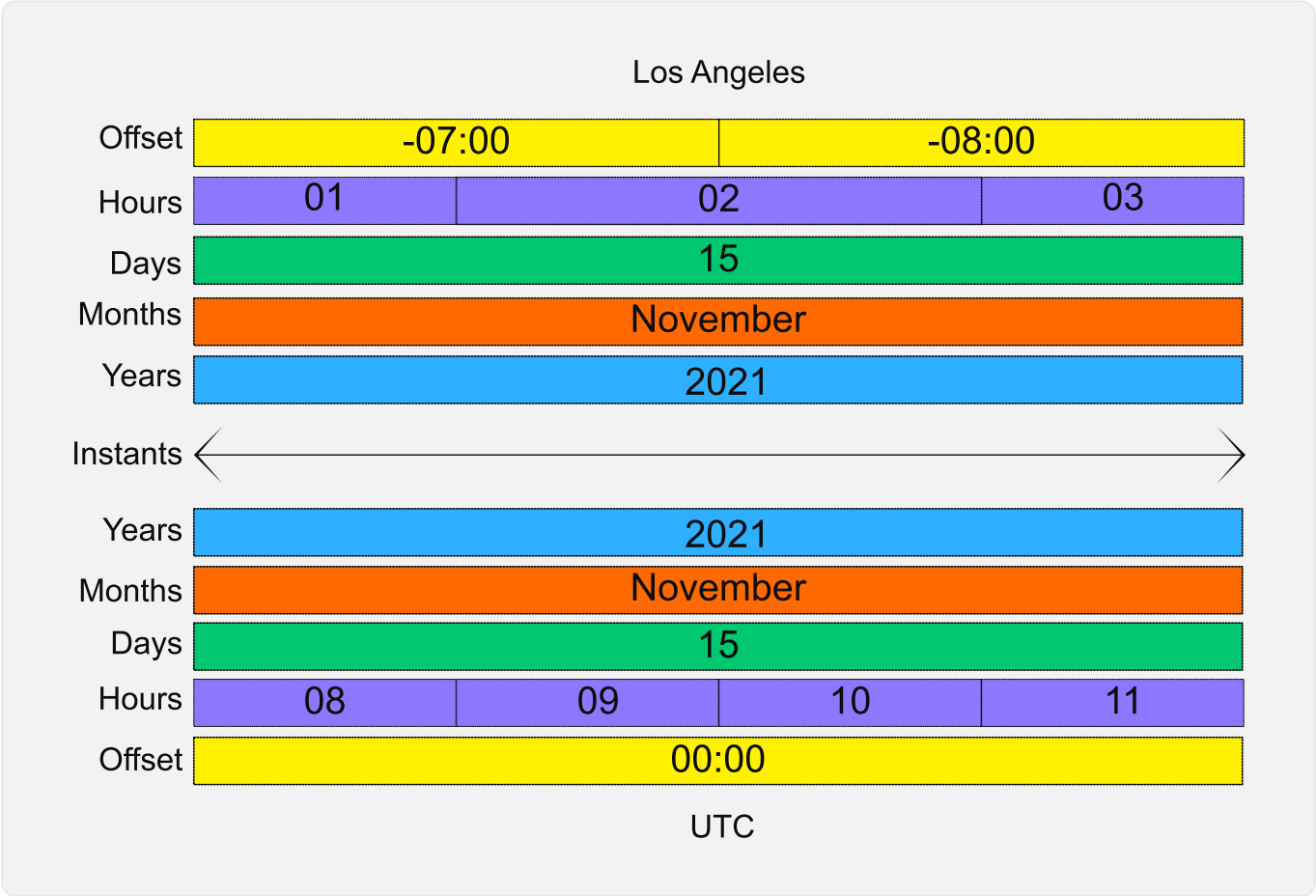
Binning is a common practice with continuous data: A range of possible values is broken up into contiguous subsets and the binning operation maps actual values to the *bin* they fall into. *Temporal binning* is simply applying this practice to instants; for example, by binning instants into years, months and days.



Temporal binning rules are complex, and generally come in two sets: *time zones* and *calendars*. For most tasks, the calendar will just be the widely used Gregorian calendar, but time zones apply locale-specific rules and can vary widely. For example, here is what binning for the `'America/Los_Angeles'` time zone looks like near the epoch:



The most common temporal binning problem occurs when daylight saving time changes. The example below contains a daylight saving time change where the ”hour” bin is two hours long. To distinguish the two hours, another range of bins containing the offset from UTC is needed:



Time Zone Support

The `TIMESTAMPTZ` type can be binned into calendar and clock bins using a suitable extension. The built-in `icu` extension implements all the binning and arithmetic functions using time zone data built from the IANA Time Zone Database and its own calendar implementation.

> Despite its name, the `icu` extension no longer depends on the International Components for Unicode (ICU) library: starting with DuckDB 2.0, it implements time zones, calendars and collations natively.

To set the time zone to use, first load the `icu` extension. The `icu` extension comes pre-bundled with several DuckDB clients (including the CLI, Python, R, JDBC and ODBC) and is autoloaded on first use, so this step can be skipped in those cases. In other cases you might first need to install and load the `icu` extension.

```
INSTALL icu;
LOAD icu;
```

Next, use the `SET TimeZone` command:

```
SET TimeZone = 'America/Los_Angeles';
```

Time binning operations for `TIMESTAMPTZ` will then be implemented using the given time zone.

A list of available time zones can be pulled from the `pg_timezone_names()` table function:

```
SELECT
    name,
    abbrev,
    utc_offset
FROM pg_timezone_names()
ORDER BY
    name;
```

You can also find a reference table of available time zones.

Calendar Support

The ICU extension also supports non-Gregorian calendars using the `SET Calendar` command. Note that the `INSTALL` and `LOAD` steps are only required if the DuckDB client does not bundle the ICU extension.

```
INSTALL icu;
LOAD icu;
SET Calendar = 'japanese';
```

Time binning operations for `TIMESTAMPTZ` will then be implemented using the given calendar. In this example, the `era` part will now report the Japanese imperial era number.

A list of available calendars can be pulled from the `icu_calendar_names()` table function:

```
SELECT name
FROM icu_calendar_names()
ORDER BY 1;
```

Settings

The current values of the `TimeZone` and `Calendar` settings are determined by the `icu` extension when it starts up. They can be queried from in the `duckdb_settings()` table function:

```
SELECT *
FROM duckdb_settings()
WHERE name = 'TimeZone';
```

name	value	description	input_type
TimeZone	Europe/Amsterdam	The current time zone	VARCHAR

```
SELECT *
FROM duckdb_settings()
WHERE name = 'Calendar';
```

name	value	description	input_type
Calendar	gregorian	The current calendar	VARCHAR

> If you find that your binning operations are not behaving as you expect, check the `TimeZone` and `Calendar` values and adjust them if needed.

Time Zone Reference List

An up-to-date version of this list can be pulled from the `pg_timezone_names()` table function:

```
SELECT name, abbrev
FROM pg_timezone_names()
ORDER BY name;
```

name	abbrev
ACT	ACT
AET	AET
AGT	AGT
ART	ART
AST	AST
Africa/Abidjan	Iceland
Africa/Accra	Iceland
Africa/Addis_Ababa	EAT
Africa/Algiers	Africa/Algiers
Africa/Asmara	EAT
Africa/Asmera	EAT
Africa/Bamako	Iceland
Africa/Bangui	Africa/Bangui
Africa/Banjul	Iceland
Africa/Bissau	Africa/Bissau
Africa/Blantyre	CAT
Africa/Brazzaville	Africa/Brazzaville
Africa/Bujumbura	CAT
Africa/Cairo	ART
Africa/Casablanca	Africa/Casablanca
Africa/Ceuta	Africa/Ceuta
Africa/Conakry	Iceland
Africa/Dakar	Iceland
Africa/Dar_es_Salaam	EAT
Africa/Djibouti	EAT
Africa/Douala	Africa/Douala
Africa/El_Aaiun	Africa/El_Aaiun
Africa/Freetown	Iceland
Africa/Gaborone	CAT
Africa/Harare	CAT
Africa/Johannesburg	Africa/Johannesburg
Africa/Juba	Africa/Juba
Africa/Kampala	EAT
Africa/Khartoum	Africa/Khartoum
Africa/Kigali	CAT
Africa/Kinshasa	Africa/Kinshasa
Africa/Lagos	Africa/Lagos
Africa/Libreville	Africa/Libreville
Africa/Lome	Iceland

name	abbrev
Africa/Luanda	Africa/Luanda
Africa/Lubumbashi	CAT
Africa/Lusaka	CAT
Africa/Malabo	Africa/Malabo
Africa/Maputo	CAT
Africa/Maseru	Africa/Maseru
Africa/Mbabane	Africa/Mbabane
Africa/Mogadishu	EAT
Africa/Monrovia	Africa/Monrovia
Africa/Nairobi	EAT
Africa/Ndjamena	Africa/Ndjamena
Africa/Niamey	Africa/Niamey
Africa/Nouakchott	Iceland
Africa/Ouagadougou	Iceland
Africa/Porto-Novo	Africa/Porto-Novo
Africa/Sao_Tome	Africa/Sao_Tome
Africa/Timbuktu	Iceland
Africa/Tripoli	Libya
Africa/Tunis	Africa/Tunis
Africa/Windhoek	Africa/Windhoek
America/Adak	America/Adak
America/Anchorage	AST
America/Anguilla	PRT
America/Antigua	PRT
America/Araguaina	America/Araguaina
America/Argentina/Buenos_Aires	AGT
America/Argentina/Catamarca	America/Argentina/Catamarca
America/Argentina/ComodRivadavia	America/Argentina/ComodRivadavia
America/Argentina/Cordoba	America/Argentina/Cordoba
America/Argentina/Jujuy	America/Argentina/Jujuy
America/Argentina/La_Rioja	America/Argentina/La_Rioja
America/Argentina/Mendoza	America/Argentina/Mendoza
America/Argentina/Rio_Gallegos	America/Argentina/Rio_Gallegos
America/Argentina/Salta	America/Argentina/Salta
America/Argentina/San_Juan	America/Argentina/San_Juan
America/Argentina/San_Luis	America/Argentina/San_Luis
America/Argentina/Tucuman	America/Argentina/Tucuman
America/Argentina/Ushuaia	America/Argentina/Ushuaia
America/Aruba	PRT

name	abbrev
America/Asuncion	America/Asuncion
America/Atikokan	EST
America/Atka	America/Atka
America/Bahia	America/Bahia
America/Bahia_Banderas	America/Bahia_Banderas
America/Barbados	America/Barbados
America/Belem	America/Belem
America/Belize	America/Belize
America/Blanc-Sablon	PRT
America/Boa_Vista	America/Boa_Vista
America/Bogota	America/Bogota
America/Boise	America/Boise
America/Buenos_Aires	AGT
America/Cambridge_Bay	America/Cambridge_Bay
America/Campo_Grande	America/Campo_Grande
America/Cancun	America/Cancun
America/Caracas	America/Caracas
America/Catamarca	America/Catamarca
America/Cayenne	America/Cayenne
America/Cayman	EST
America/Chicago	CST
America/Chihuahua	America/Chihuahua
America/Ciudad_Juarez	America/Ciudad_Juarez
America/Coral_Harbour	EST
America/Cordoba	America/Cordoba
America/Costa_Rica	America/Costa_Rica
America/Coyhaique	America/Coyhaique
America/Creston	MST
America/Cuiaba	America/Cuiaba
America/Curacao	PRT
America/Danmarkshavn	America/Danmarkshavn
America/Dawson	America/Dawson
America/Dawson_Creek	America/Dawson_Creek
America/Denver	Navajo
America/Detroit	America/Detroit
America/Dominica	PRT
America/Edmonton	America/Edmonton
America/Eirunepe	America/Eirunepe
America/El_Salvador	America/El_Salvador

name	abbrev
America/Ensenada	America/Ensenada
America/Fort_Nelson	America/Fort_Nelson
America/Fort_Wayne	IET
America/Fortaleza	America/Fortaleza
America/Glace_Bay	America/Glace_Bay
America/Godthab	America/Godthab
America/Goose_Bay	America/Goose_Bay
America/Grand_Turk	America/Grand_Turk
America/Grenada	PRT
America/Guadeloupe	PRT
America/Guatemala	America/Guatemala
America/Guayaquil	America/Guayaquil
America/Guyana	America/Guyana
America/Halifax	America/Halifax
America/Havana	Cuba
America/Hermosillo	America/Hermosillo
America/Indiana/Indianapolis	IET
America/Indiana/Knox	America/Indiana/Knox
America/Indiana/Marengo	America/Indiana/Marengo
America/Indiana/Petersburg	America/Indiana/Petersburg
America/Indiana/Tell_City	America/Indiana/Tell_City
America/Indiana/Vevay	America/Indiana/Vevay
America/Indiana/Vincennes	America/Indiana/Vincennes
America/Indiana/Winamac	America/Indiana/Winamac
America/Indianapolis	IET
America/Inuvik	America/Inuvik
America/Iqaluit	America/Iqaluit
America/Jamaica	Jamaica
America/Jujuy	America/Jujuy
America/Juneau	America/Juneau
America/Kentucky/Louisville	America/Kentucky/Louisville
America/Kentucky/Monticello	America/Kentucky/Monticello
America/Knox_IN	America/Knox_IN
America/Kralendijk	PRT
America/La_Paz	America/La_Paz
America/Lima	America/Lima
America/Los_Angeles	PST
America/Louisville	America/Louisville
America/Lower_Princes	PRT

name	abbrev
America/Maceio	America/Maceio
America/Managua	America/Managua
America/Manaus	America/Manaus
America/Marigot	PRT
America/Martinique	America/Martinique
America/Matamoros	America/Matamoros
America/Mazatlan	America/Mazatlan
America/Mendoza	America/Mendoza
America/Menominee	America/Menominee
America/Merida	America/Merida
America/Metlakatla	America/Metlakatla
America/Mexico_City	America/Mexico_City
America/Miquelon	America/Miquelon
America/Moncton	America/Moncton
America/Monterrey	America/Monterrey
America/Montevideo	America/Montevideo
America/Montreal	America/Montreal
America/Montserrat	PRT
America/Nassau	America/Nassau
America/New_York	America/New_York
America/Nipigon	America/Nipigon
America/Nome	America/Nome
America/Noronha	America/Noronha
America/North_Dakota/Beulah	America/North_Dakota/Beulah
America/North_Dakota/Center	America/North_Dakota/Center
America/North_Dakota/New_Salem	America/North_Dakota/New_Salem
America/Nuuk	America/Nuuk
America/Ojinaga	America/Ojinaga
America/Panama	EST
America/Pangnirtung	America/Pangnirtung
America/Paramaribo	America/Paramaribo
America/Phoenix	MST
America/Port-au-Prince	America/Port-au-Prince
America/Port_of_Spain	PRT
America/Porto_Acre	America/Porto_Acre
America/Porto_Velho	America/Porto_Velho
America/Puerto_Rico	PRT
America/Punta_Arenas	America/Punta_Arenas
America/Rainy_River	America/Rainy_River

name	abbrev
America/Rankin_Inlet	America/Rankin_Inlet
America/Recife	America/Recife
America/Regina	America/Regina
America/Resolute	America/Resolute
America/Rio_Branco	America/Rio_Branco
America/Rosario	America/Rosario
America/Santa_Isabel	America/Santa_Isabel
America/Santarem	America/Santarem
America/Santiago	America/Santiago
America/Santo_Domingo	America/Santo_Domingo
America/Sao_Paulo	BET
America/Scoresbysund	America/Scoresbysund
America/Shiprock	Navajo
America/Sitka	America/Sitka
America/St_Barthelemy	PRT
America/St_Johns	CNT
America/St_Kitts	PRT
America/St_Lucia	PRT
America/St_Thomas	PRT
America/St_Vincent	PRT
America/Swift_Current	America/Swift_Current
America/Tegucigalpa	America/Tegucigalpa
America/Thule	America/Thule
America/Thunder_Bay	America/Thunder_Bay
America/Tijuana	America/Tijuana
America/Toronto	America/Toronto
America/Tortola	PRT
America/Vancouver	America/Vancouver
America/Virgin	PRT
America/Whitehorse	America/Whitehorse
America/Winnipeg	America/Winnipeg
America/Yakutat	America/Yakutat
America/Yellowknife	America/Yellowknife
Antarctica/Casey	Antarctica/Casey
Antarctica/Davis	Antarctica/Davis
Antarctica/DumontDUrville	Antarctica/DumontDUrville
Antarctica/Macquarie	Antarctica/Macquarie
Antarctica/Mawson	Antarctica/Mawson
Antarctica/McMurdo	NZ

name	abbrev
Antarctica/Palmer	Antarctica/Palmer
Antarctica/Rothera	Antarctica/Rothera
Antarctica/South_Pole	NZ
Antarctica/Syowa	Antarctica/Syowa
Antarctica/Troll	Antarctica/Troll
Antarctica/Vostok	Antarctica/Vostok
Arctic/Longyearbyen	Arctic/Longyearbyen
Asia/Aden	Asia/Aden
Asia/Almaty	Asia/Almaty
Asia/Amman	Asia/Amman
Asia/Anadyr	Asia/Anadyr
Asia/Aqtau	Asia/Aqtau
Asia/Aqtobe	Asia/Aqtobe
Asia/Ashgabat	Asia/Ashgabat
Asia/Ashkhabad	Asia/Ashkhabad
Asia/Atyrau	Asia/Atyrau
Asia/Baghdad	Asia/Baghdad
Asia/Bahrain	Asia/Bahrain
Asia/Baku	Asia/Baku
Asia/Bangkok	Asia/Bangkok
Asia/Barnaul	Asia/Barnaul
Asia/Beirut	Asia/Beirut
Asia/Bishkek	Asia/Bishkek
Asia/Brunei	Asia/Brunei
Asia/Calcutta	IST
Asia/Chita	Asia/Chita
Asia/Choibalsan	Asia/Choibalsan
Asia/Chongqing	CTT
Asia/Chungking	CTT
Asia/Colombo	Asia/Colombo
Asia/Dacca	BST
Asia/Damascus	Asia/Damascus
Asia/Dhaka	BST
Asia/Dili	Asia/Dili
Asia/Dubai	Asia/Dubai
Asia/Dushanbe	Asia/Dushanbe
Asia/Famagusta	Asia/Famagusta
Asia/Gaza	Asia/Gaza
Asia/Harbin	CTT

name	abbrev
Asia/Hebron	Asia/Hebron
Asia/Ho_Chi_Minh	VST
Asia/Hong_Kong	Hongkong
Asia/Hovd	Asia/Hovd
Asia/Irkutsk	Asia/Irkutsk
Asia/Istanbul	Turkey
Asia/Jakarta	Asia/Jakarta
Asia/Jayapura	Asia/Jayapura
Asia/Jerusalem	Israel
Asia/Kabul	Asia/Kabul
Asia/Kamchatka	Asia/Kamchatka
Asia/Karachi	PLT
Asia/Kashgar	Asia/Kashgar
Asia/Kathmandu	Asia/Kathmandu
Asia/Katmandu	Asia/Katmandu
Asia/Khandyga	Asia/Khandyga
Asia/Kolkata	IST
Asia/Krasnoyarsk	Asia/Krasnoyarsk
Asia/Kuala_Lumpur	Singapore
Asia/Kuching	Asia/Kuching
Asia/Kuwait	Asia/Kuwait
Asia/Macao	Asia/Macao
Asia/Macau	Asia/Macau
Asia/Magadan	Asia/Magadan
Asia/Makassar	Asia/Makassar
Asia/Manila	Asia/Manila
Asia/Muscat	Asia/Muscat
Asia/Nicosia	Asia/Nicosia
Asia/Novokuznetsk	Asia/Novokuznetsk
Asia/Novosibirsk	Asia/Novosibirsk
Asia/Omsk	Asia/Omsk
Asia/Oral	Asia/Oral
Asia/Phnom_Penh	Asia/Phnom_Penh
Asia/Pontianak	Asia/Pontianak
Asia/Pyongyang	Asia/Pyongyang
Asia/Qatar	Asia/Qatar
Asia/Qostanay	Asia/Qostanay
Asia/Qyzylorda	Asia/Qyzylorda
Asia/Rangoon	Asia/Rangoon

name	abbrev
Asia/Riyadh	Asia/Riyadh
Asia/Saigon	VST
Asia/Sakhalin	Asia/Sakhalin
Asia/Samarkand	Asia/Samarkand
Asia/Seoul	ROK
Asia/Shanghai	CTT
Asia/Singapore	Singapore
Asia/Srednekolymsk	Asia/Srednekolymsk
Asia/Taipei	ROC
Asia/Tashkent	Asia/Tashkent
Asia/Tbilisi	Asia/Tbilisi
Asia/Tehran	Iran
Asia/Tel_Aviv	Israel
Asia/Thimbu	Asia/Thimbu
Asia/Thimphu	Asia/Thimphu
Asia/Tokyo	JST
Asia/Tomsk	Asia/Tomsk
Asia/Ujung_Pandang	Asia/Ujung_Pandang
Asia/Ulaanbaatar	Asia/Ulaanbaatar
Asia/Ulan_Bator	Asia/Ulan_Bator
Asia/Urumqi	Asia/Urumqi
Asia/Ust-Nera	Asia/Ust-Nera
Asia/Vientiane	Asia/Vientiane
Asia/Vladivostok	Asia/Vladivostok
Asia/Yakutsk	Asia/Yakutsk
Asia/Yangon	Asia/Yangon
Asia/Yekaterinburg	Asia/Yekaterinburg
Asia/Yerevan	NET
Atlantic/Azores	Atlantic/Azores
Atlantic/Bermuda	Atlantic/Bermuda
Atlantic/Canary	Atlantic/Canary
Atlantic/Cape_Verde	Atlantic/Cape_Verde
Atlantic/Faeroe	Atlantic/Faeroe
Atlantic/Faroe	Atlantic/Faroe
Atlantic/Jan_Mayen	Atlantic/Jan_Mayen
Atlantic/Madeira	Atlantic/Madeira
Atlantic/Reykjavik	Iceland
Atlantic/South_Georgia	Atlantic/South_Georgia
Atlantic/St_Helena	Iceland

name	abbrev
Atlantic/Stanley	Atlantic/Stanley
Australia/ACT	AET
Australia/Adelaide	Australia/Adelaide
Australia/Brisbane	Australia/Brisbane
Australia/Broken_Hill	Australia/Broken_Hill
Australia/Canberra	AET
Australia/Currie	Australia/Currie
Australia/Darwin	ACT
Australia/Eucla	Australia/Eucla
Australia/Hobart	Australia/Hobart
Australia/LHI	Australia/LHI
Australia/Lindeman	Australia/Lindeman
Australia/Lord_Howe	Australia/Lord_Howe
Australia/Melbourne	Australia/Melbourne
Australia/NSW	AET
Australia/North	ACT
Australia/Perth	Australia/Perth
Australia/Queensland	Australia/Queensland
Australia/South	Australia/South
Australia/Sydney	AET
Australia/Tasmania	Australia/Tasmania
Australia/Victoria	Australia/Victoria
Australia/West	Australia/West
Australia/Yancowinna	Australia/Yancowinna
BET	BET
BST	BST
Brazil/Acre	Brazil/Acre
Brazil/DeNoronha	Brazil/DeNoronha
Brazil/East	BET
Brazil/West	Brazil/West
CAT	CAT
CET	CET
CNT	CNT
CST	CST
CST6CDT	CST6CDT
CTT	CTT
Canada/Atlantic	Canada/Atlantic
Canada/Central	Canada/Central
Canada/East-Saskatchewan	Canada/East-Saskatchewan

name	abbrev
Canada/Eastern	Canada/Eastern
Canada/Mountain	Canada/Mountain
Canada/Newfoundland	CNT
Canada/Pacific	Canada/Pacific
Canada/Saskatchewan	Canada/Saskatchewan
Canada/Yukon	Canada/Yukon
Chile/Continental	Chile/Continental
Chile/EasterIsland	Chile/EasterIsland
Cuba	Cuba
EAT	EAT
ECT	ECT
EET	EET
EST	EST
EST5EDT	EST5EDT
Egypt	ART
Eire	Eire
Etc/GMT	GMT
Etc/GMT+0	GMT
Etc/GMT+1	Etc/GMT+1
Etc/GMT+10	Etc/GMT+10
Etc/GMT+11	Etc/GMT+11
Etc/GMT+12	Etc/GMT+12
Etc/GMT+2	Etc/GMT+2
Etc/GMT+3	Etc/GMT+3
Etc/GMT+4	Etc/GMT+4
Etc/GMT+5	Etc/GMT+5
Etc/GMT+6	Etc/GMT+6
Etc/GMT+7	Etc/GMT+7
Etc/GMT+8	Etc/GMT+8
Etc/GMT+9	Etc/GMT+9
Etc/GMT-0	GMT
Etc/GMT-1	Etc/GMT-1
Etc/GMT-10	Etc/GMT-10
Etc/GMT-11	Etc/GMT-11
Etc/GMT-12	Etc/GMT-12
Etc/GMT-13	Etc/GMT-13
Etc/GMT-14	Etc/GMT-14
Etc/GMT-2	Etc/GMT-2
Etc/GMT-3	Etc/GMT-3

name	abbrev
Etc/GMT-4	Etc/GMT-4
Etc/GMT-5	Etc/GMT-5
Etc/GMT-6	Etc/GMT-6
Etc/GMT-7	Etc/GMT-7
Etc/GMT-8	Etc/GMT-8
Etc/GMT-9	Etc/GMT-9
Etc/GMT0	GMT
Etc/Greenwich	GMT
Etc/UCT	UCT
Etc/UTC	UCT
Etc/Universal	UCT
Etc/Zulu	UCT
Europe/Amsterdam	CET
Europe/Andorra	Europe/Andorra
Europe/Astrakhan	Europe/Astrakhan
Europe/Athens	EET
Europe/Belfast	GB
Europe/Belgrade	Europe/Belgrade
Europe/Berlin	Europe/Berlin
Europe/Bratislava	Europe/Bratislava
Europe/Brussels	CET
Europe/Bucharest	Europe/Bucharest
Europe/Budapest	Europe/Budapest
Europe/Busingen	Europe/Busingen
Europe/Chisinau	Europe/Chisinau
Europe/Copenhagen	Europe/Copenhagen
Europe/Dublin	Eire
Europe/Gibraltar	Europe/Gibraltar
Europe/Guernsey	GB
Europe/Helsinki	Europe/Helsinki
Europe/Isle_of_Man	GB
Europe/Istanbul	Turkey
Europe/Jersey	GB
Europe/Kaliningrad	Europe/Kaliningrad
Europe/Kiev	Europe/Kiev
Europe/Kirov	Europe/Kirov
Europe/Kyiv	Europe/Kyiv
Europe/Lisbon	WET
Europe/Ljubljana	Europe/Ljubljana

name	abbrev
Europe/London	GB
Europe/Luxembourg	CET
Europe/Madrid	Europe/Madrid
Europe/Malta	Europe/Malta
Europe/Mariehamn	Europe/Mariehamn
Europe/Minsk	Europe/Minsk
Europe/Monaco	ECT
Europe/Moscow	W-SU
Europe/Nicosia	Europe/Nicosia
Europe/Oslo	Europe/Oslo
Europe/Paris	ECT
Europe/Podgorica	Europe/Podgorica
Europe/Prague	Europe/Prague
Europe/Riga	Europe/Riga
Europe/Rome	Europe/Rome
Europe/Samara	Europe/Samara
Europe/San_Marino	Europe/San_Marino
Europe/Sarajevo	Europe/Sarajevo
Europe/Saratov	Europe/Saratov
Europe/Simferopol	Europe/Simferopol
Europe/Skopje	Europe/Skopje
Europe/Sofia	Europe/Sofia
Europe/Stockholm	Europe/Stockholm
Europe/Tallinn	Europe/Tallinn
Europe/Tirane	Europe/Tirane
Europe/Tiraspol	Europe/Tiraspol
Europe/Ulyanovsk	Europe/Ulyanovsk
Europe/Uzhgorod	Europe/Uzhgorod
Europe/Vaduz	Europe/Vaduz
Europe/Vatican	Europe/Vatican
Europe/Vienna	Europe/Vienna
Europe/Vilnius	Europe/Vilnius
Europe/Volgograd	Europe/Volgograd
Europe/Warsaw	Poland
Europe/Zagreb	Europe/Zagreb
Europe/Zaporozhye	Europe/Zaporozhye
Europe/Zurich	Europe/Zurich
Factory	Factory
GB	GB

name	abbrev
GB-Eire	GB
GMT	GMT
GMT+0	GMT
GMT-0	GMT
GMT0	GMT
Greenwich	GMT
HST	HST
Hongkong	Hongkong
IET	IET
IST	IST
Iceland	Iceland
Indian/Antananarivo	EAT
Indian/Chagos	Indian/Chagos
Indian/Christmas	Indian/Christmas
Indian/Cocos	Indian/Cocos
Indian/Comoro	EAT
Indian/Kerguelen	Indian/Kerguelen
Indian/Mahe	Indian/Mahe
Indian/Maldives	Indian/Maldives
Indian/Mauritius	Indian/Mauritius
Indian/Mayotte	EAT
Indian/Reunion	Indian/Reunion
Iran	Iran
Israel	Israel
JST	JST
Jamaica	Jamaica
Japan	JST
Kwajalein	Kwajalein
Libya	Libya
MET	CET
MIT	MIT
MST	MST
MST7MDT	MST7MDT
Mexico/BajaNorte	Mexico/BajaNorte
Mexico/BajaSur	Mexico/BajaSur
Mexico/General	Mexico/General
NET	NET
NST	NZ
NZ	NZ

name	abbrev
NZ-CHAT	NZ-CHAT
Navajo	Navajo
PLT	PLT
PNT	MST
PRC	CTT
PRT	PRT
PST	PST
PST8PDT	PST8PDT
Pacific/Apia	MIT
Pacific/Auckland	NZ
Pacific/Bougainville	Pacific/Bougainville
Pacific/Chatham	NZ-CHAT
Pacific/Chuuk	Pacific/Chuuk
Pacific/Easter	Pacific/Easter
Pacific/Efate	Pacific/Efate
Pacific/Enderbury	Pacific/Enderbury
Pacific/Fakaofo	Pacific/Fakaofo
Pacific/Fiji	Pacific/Fiji
Pacific/Funafuti	Pacific/Funafuti
Pacific/Galapagos	Pacific/Galapagos
Pacific/Gambier	Pacific/Gambier
Pacific/Guadalcanal	SST
Pacific/Guam	Pacific/Guam
Pacific/Honolulu	HST
Pacific/Johnston	HST
Pacific/Kanton	Pacific/Kanton
Pacific/Kiritimati	Pacific/Kiritimati
Pacific/Kosrae	Pacific/Kosrae
Pacific/Kwajalein	Kwajalein
Pacific/Majuro	Pacific/Majuro
Pacific/Marquesas	Pacific/Marquesas
Pacific/Midway	Pacific/Midway
Pacific/Nauru	Pacific/Nauru
Pacific/Niue	Pacific/Niue
Pacific/Norfolk	Pacific/Norfolk
Pacific/Noumea	Pacific/Noumea
Pacific/Pago_Pago	Pacific/Pago_Pago
Pacific/Palau	Pacific/Palau
Pacific/Pitcairn	Pacific/Pitcairn

name	abbrev
Pacific/Pohnpei	SST
Pacific/Ponape	SST
Pacific/Port_Moresby	Pacific/Port_Moresby
Pacific/Rarotonga	Pacific/Rarotonga
Pacific/Saipan	Pacific/Saipan
Pacific/Samoa	Pacific/Samoa
Pacific/Tahiti	Pacific/Tahiti
Pacific/Tarawa	Pacific/Tarawa
Pacific/Tongatapu	Pacific/Tongatapu
Pacific/Truk	Pacific/Truk
Pacific/Wake	Pacific/Wake
Pacific/Wallis	Pacific/Wallis
Pacific/Yap	Pacific/Yap
Poland	Poland
Portugal	WET
ROC	ROC
ROK	ROK
SST	SST
Singapore	Singapore
SystemV/AST4	SystemV/AST4
SystemV/AST4ADT	SystemV/AST4ADT
SystemV/CST6	SystemV/CST6
SystemV/CST6CDT	SystemV/CST6CDT
SystemV/EST5	SystemV/EST5
SystemV/EST5EDT	SystemV/EST5EDT
SystemV/HST10	SystemV/HST10
SystemV/MST7	SystemV/MST7
SystemV/MST7MDT	SystemV/MST7MDT
SystemV/PST8	SystemV/PST8
SystemV/PST8PDT	SystemV/PST8PDT
SystemV/YST9	SystemV/YST9
SystemV/YST9YDT	SystemV/YST9YDT
Turkey	Turkey
UCT	UCT
US/Alaska	AST
US/Aleutian	US/Aleutian
US/Arizona	MST
US/Central	CST
US/East-Indiana	IET

name	abbrev
US/Eastern	US/Eastern
US/Hawaii	HST
US/Indiana-Starke	US/Indiana-Starke
US/Michigan	US/Michigan
US/Mountain	Navajo
US/Pacific	PST
US/Pacific-New	PST
US/Samoa	US/Samoa
UTC	UCT
Universal	UCT
VST	VST
W-SU	W-SU
WET	WET
Zulu	UCT

Union Type

A `UNION` *type* (not to be confused with the SQL `UNION` operator) is a nested type capable of holding one of multiple "alternative" values, much like the `union` in C. The main difference is that these `UNION` types are *tagged unions* and thus always carry a discriminator "tag" which signals which alternative it is currently holding, even if the inner value itself is null. `UNION` types are thus more similar to C++17's `std::variant`, Rust's `Enum` or the "sum type" present in most functional languages.

`UNION` types must always have at least one member, and while they can contain multiple members of the same type, the tag names must be unique. `UNION` types can have at most 255 members.

Under the hood, `UNION` types are implemented on top of `STRUCT` types, and simply keep the "tag" as the first entry.

`UNION` values can be created with the `union_value(tag := expr)` function or by casting from a member type.

Example

Create a table with a `UNION` column:

```
CREATE TABLE tbl1 (u UNION(num INTEGER, str VARCHAR));
INSERT INTO tbl1 VALUES (1), ('two'), (union_value(str := 'three'));
```

Any type can be implicitly cast to a `UNION` containing the type. Any `UNION` can also be implicitly cast to another `UNION` if the source `UNION` members are a subset of the target's (if the cast is unambiguous).

`UNION` uses the member types' VARCHAR cast functions when casting to VARCHAR:

```
SELECT u FROM tbl1;
```

u
1
two

u
three

Select all the `str` members:

```
SELECT union_extract(u, 'str') AS str
FROM tbl1;
```

str
NULL
two
three

Alternatively, you can use 'dot syntax' similarly to STRUCTs.

```
SELECT u.str
FROM tbl1;
```

str
NULL
two
three

Select the currently active tag from the `UNION` as an `ENUM`.

```
SELECT union_tag(u) AS t
FROM tbl1;
```

t
num
str
str

Union Casts

Compared to other nested types, `UNION`s allow a set of implicit casts to facilitate unintrusive and natural usage when working with their members as "subtypes". However, these casts have been designed with two principles in mind, to avoid ambiguity and to avoid casts that could lead to loss of information. This prevents `UNION`s from being completely "transparent", while still allowing `UNION` types to have a "supertype" relationship with their members.

Thus `UNION` types can't be implicitly cast to any of their member types in general, since the information in the other members not matching the target type would be "lost". If you want to coerce a `UNION` into one of its members, you should use the `union_extract` function explicitly instead.

The only exception to this is when casting a `UNION` to `VARCHAR`, in which case the members will all use their corresponding `VARCHAR` casts. Since everything can be cast to `VARCHAR`, this is "safe" in a sense.

Casting to Unions

A type can always be implicitly cast to a `UNION` if it can be implicitly cast to one of the `UNION` member types.

- If there are multiple candidates, the built in implicit casting priority rules determine the target type. For example, a `FLOAT` → `UNION(i INTEGER, v VARCHAR)` cast will always cast the `FLOAT` to the `INTEGER` member before `VARCHAR`.
- If the cast still is ambiguous, i.e., there are multiple candidates with the same implicit casting priority, an error is raised. This usually happens when the `UNION` contains multiple members of the same type, e.g., a `FLOAT` → `UNION(i INTEGER, num INTEGER)` is always ambiguous.

So how do we disambiguate if we want to create a `UNION` with multiple members of the same type? By using the `union_value` function, which takes a keyword argument specifying the tag. For example, `union_value(num := 2::INTEGER)` will create a `UNION` with a single member of type `INTEGER` with the tag `num`. This can then be used to disambiguate in an explicit (or implicit, read on below!) `UNION` to `UNION` cast, like `CAST(union_value(b := 2) AS UNION(a INTEGER, b INTEGER))`.

Casting between Unions

`UNION` types can be cast between each other if the source type is a "subset" of the target type. In other words, all the tags in the source `UNION` must be present in the target `UNION`, and all the types of the matching tags must be implicitly castable between source and target. In essence, this means that `UNION` types are covariant with respect to their members.

Ok	Source	Target	Comments
✅	`UNION(a A, b B)`	`UNION(a A, b B, c C)`	
✅	`UNION(a A, b B)`	`UNION(a A, b C)`	if B can be implicitly cast to C
❌	`UNION(a A, b B, c C)`	`UNION(a A, b B)`	
❌	`UNION(a A, b B)`	`UNION(a A, b C)`	if B can't be implicitly cast to C
❌	`UNION(A, B, D)`	`UNION(A, B, C)`	

Comparison and Sorting

Since `UNION` types are implemented on top of `STRUCT` types internally, they can be used with all the comparison operators as well as in both `WHERE` and `HAVING` clauses with the same semantics as `STRUCT`s. The "tag" is always stored as the first struct entry, which ensures that the `UNION` types are compared and ordered by "tag" first.

Functions

See Union Functions.

Typecasting

Typecasting is an operation that converts a value in one particular data type to the closest corresponding value in another data type. Like other SQL engines, DuckDB supports both implicit and explicit typecasting.

Explicit Casting

Explicit typecasting is performed by using a `CAST` expression. For example, `CAST(col AS VARCHAR)` or `col::VARCHAR` explicitly cast the column `col` to `VARCHAR`. See the cast page for more information.

Implicit Casting

In many situations, the system will add casts by itself. This is called *implicit* casting and happens, for example, when a function is called with an argument that does not match the type of the function but can be cast to the required type.

Implicit casts can only be added for a number of type combinations, and is generally only possible when the cast cannot fail. For example, an implicit cast can be added from `INTEGER` to `DOUBLE` – but not from `DOUBLE` to `INTEGER`.

Consider the function `sin(DOUBLE)`. This function takes as input argument a column of type `DOUBLE`, however, it can be called with an integer as well: `sin(1)`. The integer is converted into a double before being passed to the `sin` function.

> **Tip.** To check whether a type can be implicitly cast to another type, use the `can_cast_implicitly` function.

Combination Casting

When values of different types need to be combined to an unspecified joint parent type, the system will perform implicit casts to an automatically selected parent type. For example, `list_value(1::INT64, 1::UINT64)` creates a list of type `INT128[]`. The implicit casts performed in this situation are sometimes more lenient than regular implicit casts. For example, a `BOOL` value may be cast to `INT` (with `true` mapping to `1` and `false` to `0`) even though this is not possible for regular implicit casts.

This *combination casting* occurs for comparisons (= / < / >), set operations (`UNION` / `EXCEPT` / `INTERSECT`), and nested type constructors (`list_value` / `[...]` / `MAP`).

Casting Operations Matrix

Values of a particular data type cannot always be cast to any arbitrary target data type. The only exception is the `NULL` value – which can always be converted between types. The following matrix describes which conversions are supported. When implicit casting is allowed, it implies that explicit casting is also possible.

The matrix is generated from the documented DuckDB version by `scripts/generate_typecasting_matrix.py`; do not edit it by hand. Implicit casts are added by the system only where the conversion is generally lossless, so they are also *safe* to reverse, with the notable exception of casts from wide integer or `DECIMAL` types to `FLOAT` or `DOUBLE`, which can lose precision. Explicit-only casts may lose information or fail for particular values.

In the matrix below, rows are the source type and columns are the target type. `I` marks a cast the system also performs *implicitly*; `E` marks a cast that is only available *explicitly* (with `CAST` or `::`); an empty cell marks an unsupported cast. Even where a cast is supported, it may still fail at runtime for a particular value (for example, out of range or unparseable input).

Column abbreviations: `BOOL` (BOOLEAN), `I8` (TINYINT), `I16` (SMALLINT), `I32` (INTEGER), `I64` (BIGINT), `I128` (HUGEINT), `U8` (UTINYINT), `U16` (USMALLINT), `U32` (UINTEGER), `U64` (UBIGINT), `U128` (UHUGEINT), `F32` (FLOAT), `F64` (DOUBLE), `DEC` (DECIMAL(18, 3)), `STR` (VARCHAR), `BLOB` (BLOB), `BIT` (BIT), `UUID` (UUID), `DATE` (DATE), `TIME` (TIME), `TS` (TIMESTAMP), `TSTZ` (TIMESTAMP WITH TIME ZONE), `TTZ` (TIME WITH TIME ZONE), `IVL` (INTERVAL).

From To	BOOL	I8	I16	I32	I64	I128	U8	U16	U32	U64	U128	F32	F64	DEC	STR	BLOB	BIT	UUID	DATE	TIME	TS	TSTZ	TTZ	IVL
BOOLEAN		E	E	E	E	E	E	E	E	E	E	E	E	E	E		E							
TINYINT	I		I	I	I	I	E	E	E	E	E	I	I	I	E		E							
SMALLINT	E	I		I	I	I	E	E	E	E	E	I	I	I	E		E							
INTEGER	E	E	I		I	I	E	E	E	E	E	I	I	I	E		E							
BIGINT	E	E	E	I		I	E	E	E	E	E	I	I	I	E		E							
HUGEINT	E	E	E	E	I		E	E	E	E	E	I	I	I	E		E							
UTINYINT	E	I	I	I	I	I		I	I	I	I	I	I	I	E		E							
USMALLINT	E	I	I	I	I	I	I		I	I	I	I	I	I	E		E							
UINTEGER	E	I	I	I	I	I	I	I		I	I	I	I	I	E		E							
UBIGINT	E	I	I	I	I	I	I	I	I		I	I	I	I	E		E							
UHUGEINT	E	I	I	I	I	I	I	I	I	I		I	I	I	E		E							
FLOAT	E	E	E	E	E	E	E	E	E	E	E		E	E	E									
DOUBLE	E	E	E	E	E	E	E	E	E	E	E	I		E	E									
DECIMAL	E	E	E	E	E	E	E	E	E	E	E	E	E		E									
VARCHAR	E	E	E	E	E	E	E	E	E	E	E	E	E	E		E	E	E	E	E	E	E	E	E
BLOB															E									
BIT	E	E	E	E	E	E	E	E	E	E	E			E	E									
UUID															E									
DATE															E						E			
TIME															E						E		E	
TS															E				E	E		E	E	
TSTZ															E				E	E	E		E	
TTZ															E					E				
IVL															E									

From To	BOOL	I8	I16	I32	I64	I128	U8	U16	U32	U64	U128	F32	F64	DEC	STR	BLOB	BIT	UUID	DATE	TIME	TS	TSTZ	TTZ	IVL
USMALLINT	E	I	E	I	I	I	E	I	I	I	I	I	I	I	E		E							
UINTEGER	E	I	E	E	I	I	E	E	I	I	I	I	I	I	E		E							
UBIGINT	E	I	E	E	E	I	E	E	E	I	I	I	I	I	E		E							
UHUGEINT	E	I	E	E	E	E	E	E	E	E	I	I	I	I	E		E							
FLOAT	E	E	E	E	E	E	E	E	E	E	E	I	I	E	E		E							
DOUBLE	E	E	E	E	E	E	E	E	E	E	E	I	I	E	E		E							
DECIMAL(18, 3)	E	E	E	E	E	E	E	E	E	E	I	I	I	E										
VARCHAR	E	E	E	E	E	E	E	E	E	E	E	E	E	E	I	E	E	E	E	E	E	E	E	E
BLOB															E	I	E	E						
BIT	E	E	E	E	E	E	E	E	E	E	E	E	E		E	E	I							
UUID															E	E		I						
DATE															E				I		I	I		
TIME															E					I			E	
TIMESTAMP															E				E	E	I	I	E	
TIMESTAMP WITH TIME ZONE															E				E		I	I	E	
TIME WITH TIME ZONE															E					E			I	
INTERVAL															E									I

Even though a casting operation is supported based on the source and target data type, a cast operation may still fail at runtime if it would result in an invalid target value, for example due to overflows.

> **Deprecated.** Prior to version 0.10.0, DuckDB allowed any type to be implicitly cast to VARCHAR during function binding. Version 0.10.0 introduced a breaking change which no longer allows implicit casts to VARCHAR. The `old_implicit_casting` configuration option setting can be used to revert to the old behavior. However, please note that this flag will be deprecated in the future.

Lossy Casts

Casting operations that result in loss of precision are allowed. For example, it is possible to explicitly cast a numeric type with fractional digits – such as `DECIMAL`, `FLOAT` or `DOUBLE` – to an integral type like `INTEGER` or `BIGINT`. The number will be rounded.

```
SELECT CAST(3.1 AS INTEGER);  -- 3
SELECT CAST(3.5 AS INTEGER);  -- 4
SELECT CAST(-1.7 AS INTEGER); -- -2
```

The applicable rounding rules can be found on the `NUMERIC type page`.

Overflows

Casting operations that would result in a value overflow throw an error. For example, the value `999` is too large to be represented by the `TINYINT` data type. Therefore, an attempt to cast that value to that type results in a runtime error:

```
SELECT CAST(999 AS TINYINT);
```

```
Conversion Error:
Type INT32 with value 999 can't be cast because the value is out of range for the destination type INT8
```

So even though the cast operation from `INTEGER` to `TINYINT` is supported, it is not possible for this particular value. `TRY_CAST` can be used to convert the value into `NULL` instead of throwing an error.

Varchar

The `VARCHAR` type acts as a universal target: any arbitrary value of any arbitrary type can always be cast to the VARCHAR type. This type is also used for displaying values in the shell.

```
SELECT CAST(42.5 AS VARCHAR);
```

Casting from VARCHAR to another data type is supported, but can raise an error at runtime if DuckDB cannot parse and convert the provided text to the target data type.

```
SELECT CAST('NotANumber' AS INTEGER);
```

In general, casting to VARCHAR is a lossless operation and any type can be cast back to the original type after being converted into text.

```
SELECT CAST(CAST([1, 2, 3] AS VARCHAR) AS INTEGER[]);
```

Literal Types

Integer literals (such as `42`) and string literals (such as `'string'`) have special implicit casting rules. See the literal types page for more information.

Lists / Arrays

Lists can be explicitly cast to other lists using the same casting rules. The cast is applied to the children of the list. For example, if we convert an `INTEGER[]` list to a VARCHAR`[]` list, the child `INTEGER` elements are individually cast to VARCHAR and a new list is constructed.

```
SELECT CAST([1, 2, 3] AS VARCHAR[]);
```

Arrays

Arrays follow the same casting rules as lists. In addition, arrays can be implicitly cast to lists of the same type. For example, an `INTEGER[3]` array can be implicitly cast to an `INTEGER[]` list.

Structs

Structs can be cast to other structs as long as they share at least one field.

> The rationale behind this requirement is to help avoid unintended errors. If two structs do not have any fields in common, then the cast was likely not intended.

```
SELECT CAST({'a': 42} AS STRUCT(a VARCHAR));
```

Fields that exist in the target struct, but that do not exist in the source struct, default to `NULL`.

```
SELECT CAST({'a': 42} AS STRUCT(a VARCHAR, b VARCHAR));
```

Fields that only exist in the source struct are ignored.

```
SELECT CAST({'a': 42, 'b': 43} AS STRUCT(a VARCHAR));
```

The names of the struct can also be in a different order. The fields of the struct will be reshuffled based on the names of the structs.

```
SELECT CAST({'a': 42, 'b': 84} AS STRUCT(b VARCHAR, a VARCHAR));
```

For combination casting, the fields of the resulting struct are the superset of all fields of the input structs. This logic also applies recursively to potentially nested structs.

```
SELECT {'outer1': {'inner1': 42, 'inner2': 42}} AS c
UNION
SELECT {'outer1': {'inner2': 'hello', 'inner3': 'world'}, 'outer2': '100'} AS c;
```

```
SELECT [{'a': 42}, {'b': 84}];
```

Unions

Union casting rules can be found on the `UNION type page`.

Variant Type

The `VARIANT` type stores typed, binary data where each row is self-contained with its own type information. This differs from the JSON type, which is physically stored as text. Because type metadata is embedded per-value, `VARIANT` provides better compression and query performance than JSON for semi-structured data.

The `VARIANT` type is inspired by Snowflake's semi-structured `VARIANT` data type. It is available in Parquet since 2025 and also supported by DuckDB's Parquet reader.

Examples

Storing Different Types in the Same Column

A `VARIANT` column can hold values of different types across rows:

```
CREATE TABLE events (id INTEGER, data VARIANT);
INSERT INTO events VALUES
    (1, 42::VARIANT),
    (2, 'hello world'::VARIANT),
    (3, [1, 2, 3]::VARIANT),
    (4, {'name': 'Alice', 'age': 30}::VARIANT);
```

```
SELECT * FROM events;
```

id int32	data variant
1	42
2	hello world
3	[1, 2, 3]
4	{'age': 30, 'name': Alice}

Note that the keys of `VARIANT` objects are sorted: `{'name': 'Alice', 'age': 30}::VARIANT` is returned as `{'age': 30, 'name': Alice}`.

Checking the Type of a Value

Use `variant_typeof` to inspect the underlying type of each row:

```
SELECT id, data, variant_typeof(data) AS vtype
FROM events;
```

id int32	data variant	vtype varchar
1	42	INT32
2	hello world	VARCHAR
3	[1, 2, 3]	ARRAY(3)
4	{'age': 30, 'name': Alice}	OBJECT(age, name)

Extracting Fields from Nested Variants

Fields can be extracted from nested `VARIANT` values using dot notation or the `variant_extract` function:

```
SELECT data.name FROM events WHERE id = 4;
```

```
SELECT variant_extract(data, 'name') AS name FROM events WHERE id = 4;
```

name variant
Alice

Storage

> New Shredding `VARIANT` columns in DuckDB's storage was introduced in DuckDB 2.0.

When storing `VARIANT` columns in a DuckDB database file, DuckDB 2.0 automatically detects the common structure of the values and *shreds* them, i.e., stores the fields that share a type as separate, typed columns. Shredded data compresses well and can be scanned efficiently.

Shredding is only applied to row groups with at least `variant_minimum_shredding_size` rows (by default, 30,000 rows, roughly one fourth of the default row group size). Setting it to `-1` disables shredding:

```
SET variant_minimum_shredding_size = -1;
```

Parquet Support

DuckDB supports reading and writing `VARIANT` types from Parquet files, including *shredding,* a technique that stores nested data as flat values for more efficient access.

Writing VARIANT to Parquet

When writing `VARIANT` columns to Parquet, DuckDB can automatically shred (decompose) the variant data into typed columns based on the structure of the first row group. This auto-shredding improves read performance by enabling predicate pushdown and efficient column access.

To explicitly provide a schema for shredding, use the `SHREDDING` copy option:

```
COPY events TO 'events.parquet' (
    FORMAT parquet,
    SHREDDING {'data': 'STRUCT(name VARCHAR, age INTEGER)'}
);
```

Reading Snowflake VARIANT from Parquet

DuckDB can read shredded `VARIANT` Parquet files produced by Snowflake, automatically reconstructing the variant values from the shredded columns.

Functions

See the Variant Functions page for the functions that operate on `VARIANT` values.

Expressions

Expressions

An expression is a combination of values, operators and functions. Expressions are highly composable, and range from very simple to arbitrarily complex. They can be found in many different parts of SQL statements. This section describes the different types of operators and functions that can be used within expressions.

CASE Expression

The CASE expression performs a switch based on a condition. The basic form is identical to the ternary condition used in many programming languages (`CASE WHEN cond THEN a ELSE b END` is equivalent to `cond ? a : b`). With a single condition this can be expressed with `IF(cond, a, b)`.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE WHEN i > 2 THEN 1 ELSE 0 END AS test
FROM integers;
```

i	test
1	0
2	0
3	1

This is equivalent to:

```
SELECT i, IF(i > 2, 1, 0) AS test
FROM integers;
```

The `WHEN cond THEN expr` part of the CASE expression can be chained, whenever any of the conditions returns true for a single tuple, the corresponding expression is evaluated and returned.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE WHEN i = 1 THEN 10 WHEN i = 2 THEN 20 ELSE 0 END AS test
FROM integers;
```

i	test
1	10
2	20
3	0

The `ELSE` clause of the CASE expression is optional. If no `ELSE` clause is provided and none of the conditions match, the CASE expression will return `NULL`.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE WHEN i = 1 THEN 10 END AS test
FROM integers;
```

i	test
1	10
2	NULL
3	NULL

It is also possible to provide an individual expression after the CASE but before the WHEN. When this is done, the CASE expression is effectively transformed into a `switch` statement.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE i WHEN 1 THEN 10 WHEN 2 THEN 20 WHEN 3 THEN 30 END AS test
FROM integers;
```

i	test
1	10
2	20
3	30

This is equivalent to:

```
SELECT i, CASE WHEN i = 1 THEN 10 WHEN i = 2 THEN 20 WHEN i = 3 THEN 30 END AS test
FROM integers;
```

SWITCH Expression

The SWITCH expression is syntactic sugar for the CASE expression. It takes an expression, a MAP of values to results, and an optional default value.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, SWITCH(i, MAP {1: 'one', 2: 'two', 3: 'three'}) AS test
FROM integers;
```

i	test
1	one
2	two
3	three

A default value can be provided as the third argument, which is returned when none of the map keys match:

```
SELECT i, SWITCH(i, MAP {1: 'one', 2: 'two'}, 'other') AS test
FROM integers;
```

i	test
1	one
2	two
3	other

Casting

Casting refers to the operation of converting a value in a particular data type to the corresponding value in another data type. Casting can occur either implicitly or explicitly. The syntax described here performs an explicit cast. More information on casting can be found on the typecasting page.

Explicit Casting

The standard SQL syntax for explicit casting is `CAST(expr AS TYPENAME)`, where `TYPENAME` is a name (or alias) of one of DuckDB's data types. DuckDB also supports the shorthand `expr::TYPENAME`, which is also present in PostgreSQL.

```
SELECT CAST(i AS VARCHAR) AS i
FROM generate_series(1, 3) tbl(i);
```

i
1
2
3

```
SELECT i::DOUBLE AS i
FROM generate_series(1, 3) tbl(i);
```

i
1.0
2.0
3.0

Casting Rules

Not all casts are possible. For example, it is not possible to convert an `INTEGER` to a `DATE`. Casts may also throw errors when the cast could not be successfully performed. For example, trying to cast the string `'hello'` to an `INTEGER` will result in an error being thrown.

```
SELECT CAST('hello' AS INTEGER);
```

```
Conversion Error:
Could not convert string 'hello' to INT32
```

The exact behavior of the cast depends on the source and destination types. For example, when casting from VARCHAR to any other type, the string will be attempted to be converted.

Casting between numeric types might have rounding implications. The applicable rounding rules can be found on the `NUMERIC` type page.

TRY_CAST

TRY_CAST can be used when the preferred behavior is not to throw an error, but instead to return a NULL value. TRY_CAST will never throw an error, and will instead return NULL if a cast is not possible.

```
SELECT TRY_CAST('hello' AS INTEGER) AS i;
```

i
NULL

`cast_to_type` Function

The `cast_to_type` function allows generating a cast from an expression to the type of another column. For example:

```
SELECT cast_to_type('42', NULL::INTEGER) AS result;
```

res
int32
42

This function is primarily useful in macros, as it allows you to maintain types. This helps with making generic macros that operate on different types. For example, the following macro adds to a number if the input is an `INTEGER`:

```
CREATE TABLE tbl (i INT, s VARCHAR);
INSERT INTO tbl VALUES (42, 'hello world');

CREATE MACRO conditional_add(col, nr) AS
    CASE
        WHEN typeof(col) == 'INTEGER' THEN cast_to_type(col::INTEGER + nr, col)
        ELSE col
    END;
SELECT conditional_add(COLUMNS(*), 100) FROM tbl;
```

i	s
int32	varchar
142	hello world

Note that the CASE statement needs to return the same type in all code paths. We can perform the addition on any input column by adding a cast to the desired type – but we need to cast the result of the addition back to the source type to make the binding work.

Collations

Collations provide rules for how text should be sorted or compared in the execution engine. Collations are useful for localization, as the rules for how text should be ordered are different for different languages or for different countries. These orderings are often incompatible with one another. For example, in English the letter `y` comes between `x` and `z`. However, in Lithuanian the letter `y` comes between the `i` and `j`. For that reason, different collations are supported. The user must choose which collation they want to use when performing sorting and comparison operations.

By default, the `BINARY` collation is used. That means that strings are ordered and compared based only on their binary contents. This makes sense for standard ASCII characters (i.e., the letters A-Z and numbers 0-9), but generally does not make much sense for special unicode characters. It is, however, by far the fastest method of performing ordering and comparisons. Hence it is recommended to stick with the `BINARY` collation unless required otherwise.

> The `BINARY` collation is also available under the aliases `C` and `POSIX`.

> **Warning.** Collation support in DuckDB has some known limitations and has several planned improvements.

Using Collations

In the stand-alone installation of DuckDB three collations are included: `NOCASE`, `NOACCENT` and `NFC`. The `NOCASE` collation compares characters as equal regardless of their casing. The `NOACCENT` collation compares characters as equal regardless of their accents. The `NFC` collation performs NFC-normalized comparisons, see Unicode normalization for more information.

```
SELECT 'hello' = 'hElLO';
```

```
false
```

```
SELECT 'hello' COLLATE NOCASE = 'hElLO';
```

```
true
```

```
SELECT 'hello' = 'hëllo';
```

```
false
```

```
SELECT 'hello' COLLATE NOACCENT = 'hëllo';
```

```
true
```

Collations can be combined by chaining them using the dot operator. Note, however, that not all collations can be combined together. In general, the `NOCASE` collation can be combined with any other collator, but most other collations cannot be combined.

```
SELECT 'hello' COLLATE NOCASE = 'hElLÖ';
```

```
false
```

```
SELECT 'hello' COLLATE NOACCENT = 'hElLÖ';
```

```
false
```

```
SELECT 'hello' COLLATE NOCASE.NOACCENT = 'hElLÖ';
```

```
true
```

Default Collations

The collations we have seen so far have all been specified *per expression*. It is also possible to specify a default collator, either on the global database level or on a base table column. The `PRAGMA default_collation` can be used to specify the global default collator. This is the collator that will be used if no other one is specified.

```
SET default_collation = NOCASE;
SELECT 'hello' = 'HeLlo';
```

```
true
```

Collations can also be specified per-column when creating a table. When that column is then used in a comparison, the per-column collation is used to perform that comparison.

```
CREATE TABLE names (name VARCHAR COLLATE NOACCENT);
INSERT INTO names VALUES ('hännes');
```

```
SELECT name
FROM names
WHERE name = 'hannes';
```

```
hännes
```

Be careful here, however, as different collations cannot be combined. This can be problematic when you want to compare columns that have a different collation specified.

```
SELECT name
FROM names
WHERE name = 'hannes' COLLATE NOCASE;
```

```
ERROR: Cannot combine types with different collation!
```

```
CREATE TABLE other_names (name VARCHAR COLLATE NOCASE);
INSERT INTO other_names VALUES ('HÄNNES');
```

```
SELECT names.name AS name, other_names.name AS other_name
FROM names, other_names
WHERE names.name = other_names.name;
```

```
ERROR: Cannot combine types with different collation!
```

We need to manually overwrite the collation:

```
SELECT names.name AS name, other_names.name AS other_name
FROM names, other_names
WHERE names.name COLLATE NOACCENT.NOCASE = other_names.name COLLATE NOACCENT.NOCASE;
```

name	other_name
hännes	HÄNNES

ICU Collations

The collations we have seen so far are not region-dependent, and do not follow any specific regional rules. If you wish to follow the rules of a specific region or language, you will need to use one of the ICU collations. For that, you need to load the ICU extension.

Loading this extension will add a number of language and region specific collations to your database. These can be queried using the `PRAGMA collations` command, or by querying the `pragma_collations` function.

```
PRAGMA collations;
SELECT list(collname) FROM pragma_collations();
```

```
[af, am, ar, ar_sa, as, az, be, bg, bn, bo, br, bs, ca, ceb, chr, cs, cy, da, de, de_at, dsb, dz, ee, el,
en, en_us, eo, es, et, fa, fa_af, ff, fi, fil, fo, fr, fr_ca, fy, ga, gl, gu, ha, haw, he, he_il, hi, hr,
hsb, hu, hy, icu_noaccent, id, id_id, ig, is, it, ja, ka, kk, kl, km, kn, ko, kok, ku, ky, lb, lkt, ln, lo,
lt, lv, mk, ml, mn, mr, ms, mt, my, nb, nb_no, ne, nfc, nl, nn, noaccent, nocase, om, or, pa, pa_in, pl,
ps, pt, ro, ru, sa, se, si, sk, sl, smn, sq, sr, sr_ba, sr_me, sr_rs, sv, sw, ta, te, th, tk, to, tr, ug,
uk, ur, uz, vi, wae, wo, xh, yi, yo, yue, yue_cn, zh, zh_cn, zh_hk, zh_mo, zh_sg, zh_tw, zu]
```

These collations can then be used as the other collations would be used before. They can also be combined with the `NOCASE` collation. For example, to use the German collation rules you could use the following code snippet:

```
CREATE TABLE strings (s VARCHAR COLLATE DE);
INSERT INTO strings VALUES ('Gabel'), ('Göbel'), ('Goethe'), ('Goldmann'), ('Göthe'), ('Götz');
SELECT * FROM strings ORDER BY s;
```

```
"Gabel", "Göbel", "Goethe", "Goldmann", "Göthe", "Götz"
```

Comparisons

Comparison Operators

The table below shows the standard comparison operators. Whenever either of the input arguments is `NULL`, the output of the comparison is `NULL`.

Operator	Description	Example	Result
<	less than	`2 < 3`	`true`
>	greater than	`2 > 3`	`false`
<=	less than or equal to	`2 <= 3`	`true`
>=	greater than or equal to	`4 >= NULL`	`NULL`
= or ==	equal	`NULL = NULL`	`NULL`
<> or !=	not equal	`2 <> 2`	`false`

The table below shows the standard distinction operators. These operators treat `NULL` values as equal.

Operator	Description	Example	Result
`IS DISTINCT FROM`	not equal, including `NULL`	`2 IS DISTINCT FROM NULL`	`true`
`IS NOT DISTINCT FROM`	equal, including `NULL`	`NULL IS NOT DISTINCT FROM NULL`	`true`

Combination Casting

When performing comparison on different types, DuckDB performs Combination Casting. These casts were introduced to make interactive querying more convenient and are in line with the casts performed by several programming languages but are often not compatible with PostgreSQL's behavior. For example, the following expressions evaluate and return `true` in DuckDB but fail in PostgreSQL.

```
SELECT 1 = true;
SELECT 1 = '1.1';
```

> It is not possible to enforce stricter type-checking for DuckDB's comparison operators. If you require stricter type-checking, consider creating a macro with the `typeof` function or implementing a user-defined function.

`BETWEEN` and `IS [NOT] NULL`

Besides the standard comparison operators there are also the `BETWEEN` and `IS (NOT) NULL` operators. These behave much like operators, but have special syntax mandated by the SQL standard. They are shown in the table below.

Note that `BETWEEN` and `NOT BETWEEN` are only equivalent to the examples below in the cases where both `a`, `x` and `y` are of the same type, as `BETWEEN` will cast all of its inputs to the same type.

Predicate	Description
`a BETWEEN x AND y`	equivalent to `x <= a AND a <= y`
`a NOT BETWEEN x AND y`	equivalent to `x > a OR a > y`
`expression IS NULL`	`true` if expression is `NULL`, `false` otherwise
`expression ISNULL`	alias for `IS NULL` (non-standard)
`expression IS NOT NULL`	`false` if expression is `NULL`, `true` otherwise
`expression NOTNULL`	alias for `IS NOT NULL` (non-standard)

An explicit collation on any input to `BETWEEN` applies to the entire `BETWEEN` expression. This can differ from writing the comparisons separately, where the explicit collation applies only to the comparison that contains it:

```
SELECT
    'B' NOT BETWEEN ('A' COLLATE NOCASE) AND 'a' AS between_result,
    'B' < ('A' COLLATE NOCASE) OR 'B' > 'a' AS separate_result;
```

between_result	separate_result
true	false

> For the expression `BETWEEN x AND y`, `x` is used as the lower bound and `y` is used as the upper bound. Therefore, if `x > y`, the result will always be `false`.

IN Operator

The `IN` operator checks containment of the left expression inside the *collection* on the right hand side (RHS). Supported collections on the RHS are tuples, lists, and maps, as well as subqueries that return a single column. The single-column restriction applies only to subqueries; tuples, lists, and maps may themselves contain composite values.

`IN (val1, val2, ...)` (Tuple)

The `IN` operator on a tuple `(val1, val2, ...)` returns `true` if the expression is present in the RHS, `false` if the expression is not in the RHS and the RHS has no `NULL` values, or `NULL` if the expression is not in the RHS and the RHS has `NULL` values.

```
SELECT 'Math' IN ('CS', 'Math');
```

```
true
```

```
SELECT 'English' IN ('CS', 'Math');
```

```
false
```

```
SELECT 'Math' IN ('CS', 'Math', NULL);
```

```
true
```

```
SELECT 'English' IN ('CS', 'Math', NULL);
```

```
NULL
```

`IN [val1, val2, ...]` (List)

The `IN` operator works on lists according to the semantics used in Python. Unlike for the `IN tuple` operator, the presence of `NULL` values on the right hand side of the expression does not make a difference in the result:

```
SELECT 'Math' IN ['CS', 'Math', NULL];
```

```
true
```

```
SELECT 'English' IN ['CS', 'Math', NULL];
```

```
false
```

`IN` Map

The `IN` operator works on maps according to the semantics used in Python, i.e., it checks for the presence of keys (not values):

```
SELECT 'key1' IN MAP {'key1': 50, 'key2': 75};
```

```
true
```

```
SELECT 'key3' IN MAP {'key1': 50, 'key2': 75};
```

```
false
```

`IN` Subquery

The `IN` operator works with subqueries that return a single column. For example:

```
SELECT 42 IN (SELECT unnest([32, 42, 52]) AS x);
```

```
true
```

If the subquery returns more than one column, a Binder Error is thrown:

```
SELECT 42 IN (SELECT unnest([32, 42, 52]) AS x, 'a' AS y);
```

```
Binder Error:
Subquery returns 2 columns - expected 1
```

`IN` String

The `IN` operator can be used as a shorthand for the `contains` string function. For example:

```
SELECT 'Hello' IN 'Hello World';
```

```
true
```

`NOT IN`

`NOT IN` can be used to check if an element is not present in the set. `x NOT IN y` is equivalent to `NOT (x IN y)`.

Logical Operators

The following logical operators are available: `AND`, `OR` and `NOT`. SQL uses a three-valued logic system with `true`, `false` and `NULL`. Note that logical operators involving `NULL` do not always evaluate to `NULL`. For example, `NULL AND false` will evaluate to `false`, and `NULL OR true` will evaluate to `true`. Below are the complete truth tables.

Binary Operators: `AND` and `OR`

a	b	a AND b	a OR b
true	true	true	true
true	false	false	true
true	NULL	NULL	true
false	false	false	false
false	NULL	false	NULL
NULL	NULL	NULL	NULL

Unary Operator: `NOT`

a	NOT a
true	false
false	true

a	NOT a
NULL	NULL

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

Star Expression

Syntax

The * expression can be used in a SELECT statement to select all columns that are projected in the FROM clause.

```
SELECT *
FROM tbl;
```

TABLE.* and STRUCT.*

The * expression can be prepended by a table name to select only columns from that table.

```
SELECT tbl.*
FROM tbl
JOIN other_tbl USING (id);
```

Similarly, the * expression can also be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys. See the STRUCT data type and STRUCT functions pages for more details on working with structs.

For example:

```
SELECT st.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS st);
```

x	y	z
1	2	3

EXCLUDE Clause

EXCLUDE allows you to exclude specific columns from the * expression.

```
SELECT * EXCLUDE (col)
FROM tbl;
```

> New As of DuckDB 2.0, EXCEPT is accepted as a synonym for EXCLUDE, matching the SQL-standard and BigQuery spelling. The two keywords are interchangeable and may be mixed in the same query.
>
> ```
> SELECT * EXCEPT (col)
> FROM tbl;
> ```

REPLACE Clause

REPLACE allows you to replace specific columns by alternative expressions.

```
SELECT * REPLACE (col1 / 1_000 AS col1, col2 / 1_000 AS col2)
FROM tbl;
```

RENAME Clause

RENAME allows you to replace specific columns.

```
SELECT * RENAME (col1 AS height, col2 AS width)
FROM tbl;
```

Column Filtering via Pattern Matching Operators

The pattern matching operators `LIKE`, `GLOB`, `SIMILAR TO` and their variants allow you to select columns by matching their names to patterns.

```
SELECT * LIKE 'col%'
FROM tbl;
```

```
SELECT * GLOB 'col*'
FROM tbl;
```

```
SELECT * SIMILAR TO 'col.'
FROM tbl;
```

The NOT variants of these operators are also supported to exclude columns that match the pattern:

```
SELECT * NOT SIMILAR TO 'col.'
FROM tbl;
```

COLUMNS Expression

The COLUMNS expression is similar to the regular star expression, but additionally allows you to execute the same expression on the resulting columns.

```
CREATE TABLE numbers (id INTEGER, number INTEGER);
INSERT INTO numbers VALUES (1, 10), (2, 20), (3, NULL);
SELECT min(COLUMNS(*)), count(COLUMNS(*)) FROM numbers;
```

id	number	id	number
1	10	3	2

```
SELECT
    min(COLUMNS(* REPLACE (number + id AS number))),
    count(COLUMNS(* EXCLUDE (number)))
FROM numbers;
```

id	min((number + id))	id
1	11	3

COLUMNS expressions can also be combined, as long as they contain the same star expression:

```
SELECT COLUMNS(*) + COLUMNS(*) FROM numbers;
```

id	number
2	20
4	40

id	number
6	NULL

COLUMNS Expression in a WHERE Clause

COLUMNS expressions can also be used in WHERE clauses. The conditions are applied to all columns and are combined using the logical AND operator.

```
SELECT *
FROM (
    SELECT 'a', 'a'
    UNION ALL
    SELECT 'a', 'b'
    UNION ALL
    SELECT 'b', 'b'
) _(x, y)
WHERE COLUMNS(*) = 'a'; -- equivalent to: x = 'a' AND y = 'a'
```

x	y
a	a

To combine conditions using the logical OR operator, you can UNPACK the COLUMNS expression into the variadic `greatest` function.

```
SELECT *
FROM (
    SELECT 'a', 'a'
    UNION ALL
    SELECT 'a', 'b'
    UNION ALL
    SELECT 'b', 'b'
) _(x, y)
WHERE greatest(UNPACK(COLUMNS(*) = 'a')); -- equivalent to: x = 'a' OR y = 'a'
```

x	y
a	a
a	b

COLUMNS Expression in DISTINCT ON

COLUMNS expressions can be used in DISTINCT ON clauses to specify distinct columns by pattern:

```
SELECT DISTINCT ON (COLUMNS('x|y')) *
FROM (VALUES (1, 2, 'a'), (1, 2, 'b'), (3, 4, 'c')) t(x, y, z);
```

x	y	z
1	2	a
3	4	c

Regular Expressions in a COLUMNS Expression

COLUMNS expressions don't currently support the pattern matching operators, but they do support regular expression matching by simply passing a string constant in place of the star:

```
SELECT COLUMNS('(id|numbers?)') FROM numbers;
```

id	number
1	10
2	20
3	NULL

Renaming Columns with Regular Expressions in a COLUMNS Expression

The matches of capture groups in regular expressions can be used to rename matching columns. The capture groups are one-indexed; `\0` is the original column name.

For example, to select the first three letters of column names, run:

```
SELECT COLUMNS('(\w{3}).*') AS '\1' FROM numbers;
```

num
10
20
NULL

To remove a colon (`:`) character in the middle of a column name, run:

```
CREATE TABLE tbl ("Foo:Bar" INTEGER, "Foo:Baz" INTEGER, "Foo:Qux" INTEGER);
SELECT COLUMNS('(\w*):(\w*)') AS '\1\2' FROM tbl;
```

To add the original column name to the expression alias, run:

```
SELECT min(COLUMNS(*)) AS "min_\0" FROM numbers;
```

min_id	min_number
1	10

COLUMNS Lambda Function

COLUMNS also supports passing in a lambda function. The lambda function is evaluated for all columns present in the FROM clause. Only columns for which the lambda function evaluates to TRUE are retained, and discarded otherwise.

COLUMNS-clause lambdas have a mandatory parameter that receives the column name. Optionally, a second parameter may be declared to receive the (1-based) index of the column. COLUMNS-clause lambdas allow the execution of arbitrary expressions in order to select and rename columns.

```
SELECT COLUMNS(lambda c: c LIKE '%num%') FROM numbers;
```

number
10
20
NULL

COLUMNS List

COLUMNS also supports passing in a list of column names.

```
SELECT COLUMNS(['id', 'number']) FROM numbers;
```

id	number
1	10
2	20
3	NULL

Unpacking a COLUMNS Expression

By wrapping a COLUMNS expression in UNPACK, the columns expand into a parent expression, much like the iterable unpacking behavior in Python.

Without UNPACK, operations on the COLUMNS expression are applied to each column separately:

```
SELECT coalesce(COLUMNS(['a', 'b', 'c'])) AS result
FROM (SELECT NULL a, 42 b, true c);
```

result	result	result
NULL	42	true

With UNPACK, the COLUMNS expression is expanded into its parent expression, `coalesce` in the example above, which results in a single column:

```
SELECT coalesce(UNPACK(COLUMNS(['a', 'b', 'c']))) AS result
FROM (SELECT NULL AS a, 42 AS b, true AS c);
```

result
42

The UNPACK keyword may be replaced by `*`, matching Python syntax, when it is applied directly to the COLUMNS expression without any intermediate operations.

```
SELECT coalesce(*COLUMNS(*)) AS result
FROM (SELECT NULL a, 42 AS b, true AS c);
```

result
42

> **Warning.** In the following example, replacing UNPACK by * results in a syntax error:
>
> ```
> SELECT greatest(UNPACK(COLUMNS(*) + 1)) AS result
> FROM (SELECT 1 AS a, 2 AS b, 3 AS c);
> ```
>
> | result |
> |---|
> | 4 |

STRUCT.*

The * expression can also be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys. See the STRUCT data type and STRUCT functions pages for more details on working with structs.

For example:

```
SELECT st.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS st);
```

x	y	z
1	2	3

Subqueries

Subqueries are parenthesized query expressions that appear as part of a larger, outer query. Subqueries are usually based on `SELECT ... FROM`, but in DuckDB other query constructs such as `PIVOT` can also appear as a subquery.

Scalar Subquery

Scalar subqueries are subqueries that return a single value. They can be used anywhere where an expression can be used. If a scalar subquery returns more than a single value, an error is raised (unless `scalar_subquery_error_on_multiple_rows` is set to `false`, in which case a row is selected randomly).

Consider the following table:

Grades

grade	course
7	Math
9	Math
8	CS

```
CREATE TABLE grades (grade INTEGER, course VARCHAR);
INSERT INTO grades VALUES (7, 'Math'), (9, 'Math'), (8, 'CS');
```

We can run the following query to obtain the minimum grade:

```
SELECT min(grade) FROM grades;
```

min(grade)
7

By using a scalar subquery in the WHERE clause, we can figure out for which course this grade was obtained:

```
SELECT course FROM grades WHERE grade = (SELECT min(grade) FROM grades);
```

course
Math

ARRAY Subqueries

Subqueries that return multiple values can be wrapped with ARRAY to collect all results in a list.

```
SELECT ARRAY(SELECT grade FROM grades) AS all_grades;
```

all_grades
[7, 9, 8]

Subquery Comparisons: ALL, ANY and SOME

In the section on scalar subqueries, a scalar expression was compared directly to a subquery using the equality comparison operator (=). Such direct comparisons only make sense with scalar subqueries.

Scalar expressions can still be compared to single-column subqueries returning multiple rows by specifying a quantifier. Available quantifiers are ALL, ANY and SOME. The quantifiers ANY and SOME are equivalent.

ALL

The ALL quantifier specifies that the comparison as a whole evaluates to `true` when the individual comparison results of *the expression at the left hand side of the comparison operator* with each of the values from *the subquery at the right hand side of the comparison operator* **all** evaluate to `true`:

```
SELECT 6 <= ALL (SELECT grade FROM grades) AS adequate;
```

returns:

adequate
true

because 6 is less than or equal to each of the subquery results 7, 8 and 9.

However, the following query

```
SELECT 8 >= ALL (SELECT grade FROM grades) AS excellent;
```

returns

excellent
false

because 8 is not greater than or equal to the subquery result 9. And thus, because not all comparisons evaluate to `true`, `>= ALL` as a whole evaluates to `false`.

ANY

The ANY quantifier specifies that the comparison as a whole evaluates to `true` when at least one of the individual comparison results evaluates to `true`. For example:

```
SELECT 5 >= ANY (SELECT grade FROM grades) AS fail;
```

returns

fail
false

because no result of the subquery is less than or equal to 5.

The quantifier SOME may be used instead of ANY: ANY and SOME are interchangeable.

EXISTS

The EXISTS operator tests for the existence of any row inside the subquery. It returns either true when the subquery returns one or more records, and false otherwise. The EXISTS operator is generally the most useful as a *correlated* subquery to express semijoin operations. However, it can be used as an uncorrelated subquery as well.

For example, we can use it to figure out if there are any grades present for a given course:

```
SELECT EXISTS (FROM grades WHERE course = 'Math') AS math_grades_present;
```

math_grades_present
true

```
SELECT EXISTS (FROM grades WHERE course = 'History') AS history_grades_present;
```

history_grades_present
false

> The subqueries in the examples above make use of the fact that you can omit the `SELECT *` in DuckDB thanks to the FROM-first syntax. The `SELECT` clause is required in subqueries by other SQL systems but cannot fulfill any purpose in `EXISTS` and `NOT EXISTS` subqueries.

`NOT EXISTS`

The `NOT EXISTS` operator tests for the absence of any row inside the subquery. It returns either true when the subquery returns an empty result, and false otherwise. The `NOT EXISTS` operator is generally the most useful as a *correlated* subquery to express antijoin operations. For example, to find Person nodes without an interest:

```
CREATE TABLE Person (id BIGINT, name VARCHAR);
CREATE TABLE interest (PersonId BIGINT, topic VARCHAR);

INSERT INTO Person VALUES (1, 'Jane'), (2, 'Joe');
INSERT INTO interest VALUES (2, 'Music');

SELECT *
FROM Person
WHERE NOT EXISTS (FROM interest WHERE interest.PersonId = Person.id);
```

id	name
1	Jane

> DuckDB automatically detects when a `NOT EXISTS` query expresses an antijoin operation. There is no need to manually rewrite such queries to use `LEFT OUTER JOIN ... WHERE ... IS NULL`.

`IN` Operator

The `IN` operator checks containment of the left expression inside the result defined by the subquery or the set of expressions on the right hand side (RHS). The `IN` operator returns true if the expression is present in the RHS, false if the expression is not in the RHS and the RHS has no `NULL` values, or `NULL` if the expression is not in the RHS and the RHS has `NULL` values.

We can use the `IN` operator in a similar manner as we used the `EXISTS` operator:

```
SELECT 'Math' IN (SELECT course FROM grades) AS math_grades_present;
```

math_grades_present
true

Correlated Subqueries

All the subqueries presented here so far have been **uncorrelated** subqueries, where the subqueries themselves are entirely self-contained and can be run without the parent query. There exists a second type of subqueries called **correlated** subqueries. For correlated subqueries, the subquery uses values from the parent subquery.

Conceptually, the subqueries are run once for every single row in the parent query. Perhaps a simple way of envisioning this is that the correlated subquery is a **function** that is applied to every row in the source dataset.

For example, suppose that we want to find the minimum grade for every course. We could do that as follows:

```
SELECT *
FROM grades grades_parent
WHERE grade =
    (SELECT min(grade)
     FROM grades
     WHERE grades.course = grades_parent.course);
```

grade	course
7	Math
8	CS

The subquery uses a column from the parent query (`grades_parent.course`). Conceptually, we can see the subquery as a function where the correlated column is a parameter to that function:

```
SELECT min(grade)
FROM grades
WHERE course = ?;
```

Now when we execute this function for each of the rows, we can see that for `Math` this will return `7`, and for `CS` it will return `8`. We then compare it against the grade for that actual row. As a result, the row `(Math, 9)` will be filtered out, as `9 <> 7`.

Returning Each Row of the Subquery as a Struct

Using the name of a subquery in the `SELECT` clause (without referring to a specific column) turns each row of the subquery into a struct whose fields correspond to the columns of the subquery. For example:

```
SELECT t
FROM (SELECT unnest(generate_series(41, 43)) AS x, 'hello' AS y) t;
```

t
{'x': 41, 'y': hello}
{'x': 42, 'y': hello}
{'x': 43, 'y': hello}

TRY Expression

The TRY expression ensures that errors caused by the input rows in the child (scalar) expression result in `NULL` for those rows, instead of causing the query to throw an error.

> The TRY expression was inspired by the TRY_CAST expression.

Examples

The following calls return errors when invoked without the TRY expression. When they are wrapped into a TRY expression, they return `NULL`:

Casting

Without TRY

```
SELECT 'abc'::INTEGER;
```

```
Conversion Error:
Could not convert string 'abc' to INT32
```

With TRY

```
SELECT TRY('abc'::INTEGER);
```

```
NULL
```

Integer Division by Zero

Without TRY

```
SELECT 1 // 0;
```

```
Invalid Input Error:
Division by zero in expression (1 // 0). Use TRY(...) to return NULL for this expression, or SET error_on_
division_by_zero=false to return NULL for all divisions by zero.
```

With TRY

```
SELECT TRY(1 // 0);
```

```
NULL
```

Casting Multiple Rows

Without TRY

```
WITH cte AS (FROM (VALUES ('123'), ('test'), ('235')) t(a))
SELECT a::INTEGER AS x FROM cte;
```

```
Conversion Error:
Could not convert string 'test' to INT32
```

With TRY

```
WITH cte AS (FROM (VALUES ('123'), ('test'), ('235')) t(a))
SELECT TRY(a::INTEGER) AS x FROM cte;
```

x
123
NULL
235

Limitations

TRY cannot be used in combination with a volatile function, an aggregate function, or a scalar subquery. For example:

```
SELECT TRY(random())
```

```
Binder Error:
TRY can not be used in combination with a volatile function
```

Functions

Functions

Function Syntax

Function Chaining via the Dot Operator

DuckDB supports the dot syntax for function chaining. This allows the function call `fn(arg1, arg2, arg3, ...)` to be rewritten as `arg1.fn(arg2, arg3, ...)`. For example, take the following use of the `replace` function:

```
SELECT replace(goose_name, 'goose', 'duck') AS duck_name
FROM unnest(['African goose', 'Faroese goose', 'Hungarian goose', 'Pomeranian goose']) breed(goose_name);
```

This can be rewritten as follows:

```
SELECT goose_name.replace('goose', 'duck') AS duck_name
FROM unnest(['African goose', 'Faroese goose', 'Hungarian goose', 'Pomeranian goose']) breed(goose_name);
```

Using with Literals and Arrays

String literals and array access operations can be used with function chaining directly:

```
SELECT 'hello world'.replace(' ', '_');
```

```
SELECT m[1].map_entries()
FROM (VALUES ([MAP {'hello': 42}, MAP {'world': 42}])) t(m);
```

To apply function chaining to numeric literals, you must surround the literal with parentheses, e.g.:

```
SELECT (2).sqrt();
```

In the absence of these parentheses (e.g., `SELECT 2.sqrt();`), DuckDB returns a `Parser Error` for the function call:

```
Parser Error:
syntax error at or near "("
```

Limitations

Function chaining via the dot operator is limited to *scalar* functions and is not supported for *table* functions. For example, the following call returns a `Parser Error`:

```
SELECT * FROM ('my_file.parquet').read_parquet(); -- does not work
```

Query Functions

The `duckdb_functions()` table function shows the list of functions currently built into the system.

```
SELECT DISTINCT ON(function_name)
    function_name,
    function_type,
    return_type,
    parameters,
    parameter_types,
    description
FROM duckdb_functions()
WHERE function_type = 'scalar'
  AND function_name LIKE 'b%'
ORDER BY function_name;
```

function_name	function_type	return_type	parameters	parameter_types	description
bar	scalar	VARCHAR	[x, min, max, width]	[DOUBLE, DOUBLE, DOUBLE, DOUBLE]	Draws a band whose width is proportional to (x - min) and equal to width characters when x = max. width defaults to 80
base64	scalar	VARCHAR	[blob]	[BLOB]	Convert a blob to a base64 encoded string
bin	scalar	VARCHAR	[value]	[VARCHAR]	Converts the value to binary representation
bit_count	scalar	TINYINT	[x]	[TINYINT]	Returns the number of bits that are set
bit_length	scalar	BIGINT	[col0]	[VARCHAR]	Number of bits in a string
bit_position	scalar	INTEGER	[substring, bitstring]	[BIT, BIT]	Returns first starting index of the specified substring within bits, or zero if it is not present. The first (leftmost) bit is indexed 1
bitstring	scalar	BIT	[bitstring, length]	[VARCHAR, INTEGER]	Pads the bitstring until the specified length

Aggregate Functions

Examples

Produce a single row containing the sum of the `amount` column:

```
SELECT sum(amount)
FROM sales;
```

Produce one row per unique region, containing the sum of `amount` for each group:

```
SELECT region, sum(amount)
FROM sales
GROUP BY region;
```

Return only the regions that have a sum of `amount` higher than 100:

```
SELECT region
FROM sales
GROUP BY region
HAVING sum(amount) > 100;
```

Return the number of unique values in the `region` column:

```
SELECT count(DISTINCT region)
FROM sales;
```

Return two values, the total sum of `amount` and the sum of `amount` minus columns where the region is `north` using the `FILTER` clause:

```
SELECT sum(amount), sum(amount) FILTER (region != 'north')
FROM sales;
```

Returns a list of all regions in order of the `amount` column:

```
SELECT list(region ORDER BY amount DESC)
FROM sales;
```

Returns the amount of the first sale using the `first()` aggregate function:

```
SELECT first(amount ORDER BY date ASC)
FROM sales;
```

Downsample a series of 100 points to 5 points that preserve its visual shape using the `lttb()` aggregate function:

```
SELECT unnest(lttb(x, sin(x), 5 ORDER BY x)) AS point
FROM range(0, 100) t(x);
```

```
┌──────────────────────────────────────────┐
│                  point                   │
│      struct(x double, y double)          │
├──────────────────────────────────────────┤
│ {'x': 0.0, 'y': 0.0}                     │
│ {'x': 30.0, 'y': -0.9880316240928618}    │
│ {'x': 33.0, 'y': 0.9999118601072672}     │
│ {'x': 96.0, 'y': 0.9835877454343449}     │
│ {'x': 99.0, 'y': -0.9992068341863537}    │
└──────────────────────────────────────────┘
```

Syntax

Aggregates are functions that *combine* multiple rows into a single value. Aggregates are different from scalar functions and window functions because they change the cardinality of the result. Aggregate functions can be used in the `SELECT`, `HAVING`, and `ORDER BY` clauses of a SQL query.

`DISTINCT` Clause in Aggregate Functions

When the `DISTINCT` clause is provided, only distinct values are considered in the computation of the aggregate. This is typically used in combination with the `count` aggregate to get the number of distinct elements; but it can be used together with any aggregate function in the system. There are some aggregates that are insensitive to duplicate values (e.g., `min` and `max`) and for them this clause is parsed and ignored.

`ORDER BY` Clause in Aggregate Functions

An `ORDER BY` clause can be provided after the last argument of the function call. Note the lack of the comma separator before the clause.

```
SELECT <aggregate_function>(<arg>, <sep> ORDER BY <ordering_criteria>);
```

This clause ensures that the values being aggregated are sorted before applying the function. Most aggregate functions are order-insensitive, and for them this clause is parsed and discarded. However, there are some order-sensitive aggregates that can have non-deterministic results without ordering, e.g., `first`, `last`, `list` and `string_agg` / `group_concat` / `listagg`. These can be made deterministic by ordering the arguments.

For example:

```
CREATE TABLE tbl AS
    SELECT s FROM range(1, 4) r(s);

SELECT string_agg(s, ', ' ORDER BY s DESC) AS countdown
FROM tbl;
```

countdown
3, 2, 1

Handling NULL Values

All general aggregate functions ignore NULLs, except for `list` (`array_agg`), `first` (`arbitrary`) and `last`. To exclude NULLs from `list`, use a `FILTER` clause that excludes rows with null input, such as `list(x) FILTER (WHERE x IS NOT NULL)`. A filter on another column may still leave NULL values in the list. To ignore NULLs from `first`, you can use the `any_value` aggregate.

All general aggregate functions except `count` return NULL on empty groups. In particular, `list` does *not* return an empty list, `sum` does *not* return zero, and `string_agg` does *not* return an empty string in this case.

General Aggregate Functions

The table below shows the available general aggregate functions.

Function	Description
`any_value(arg)`	Returns the first non-null value from `arg`. This function is affected by ordering.
`arg_max(arg, val)`	Finds the row with the maximum `val` and calculates the `arg` expression at that row. Rows where the value of the `arg` or `val` expression is NULL are ignored. This function is affected by ordering.
`arg_max(arg, val, n)`	The generalized case of `arg_max` for n values: returns a `LIST` containing the `arg` expressions for the top n rows ordered by `val` descending. This function is affected by ordering.
`arg_max_null(arg, val)`	Finds the row with the maximum `val` and calculates the `arg` expression at that row. Rows where the `val` expression evaluates to NULL are ignored. This function is affected by ordering.
`arg_max_nulls_last(arg, val[, n])`	Finds the row with the maximum `val` and calculates the `arg` expression at that row. Unlike `arg_max`, NULL values of `arg` are not ignored and rows where `val` is NULL are sorted last, i.e., they are only selected if there are no other rows. If n is specified, returns a `LIST` containing the `arg` expressions for the top n rows ordered by `val` descending, with NULL values of `val` last. This function is affected by ordering.
`arg_min(arg, val)`	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the value of the `arg` or `val` expression is NULL are ignored. This function is affected by ordering.
`arg_min(arg, val, n)`	Returns a `LIST` containing the `arg` expressions for the ”bottom” n rows ordered by `val` ascending. This function is affected by ordering.
`arg_min_null(arg, val)`	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the `val` expression evaluates to NULL are ignored. This function is affected by ordering.
`arg_min_nulls_last(arg, val[, n])`	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Unlike `arg_min`, NULL values of `arg` are not ignored and rows where `val` is NULL are sorted last, i.e., they are only selected if there are no other rows. If n is specified, returns a `LIST` containing the `arg` expressions for the top n rows ordered by `val` ascending, with NULL values of `val` last. This function is affected by ordering.
`avg(arg)`	Calculates the average of all non-null values in `arg`. This function is affected by ordering.
`bit_and(arg)`	Returns the bitwise AND of all bits in a given expression.
`bit_or(arg)`	Returns the bitwise OR of all bits in a given expression.
`bit_xor(arg)`	Returns the bitwise XOR of all bits in a given expression.

Function	Description
`bitstring_agg(arg)`	Returns a bitstring whose length corresponds to the range of the non-null (integer) values, with bits set at the location of each (distinct) value.
`bool_and(arg)`	Returns `true` if every input value is `true`, otherwise `false`.
`bool_or(arg)`	Returns `true` if any input value is `true`, otherwise `false`.
`count()`	Returns the number of rows.
`count(arg)`	Returns the number of rows where `arg` is not `NULL`.
`countif(arg)`	Returns the number of rows where `arg` is `true`.
`decimal_average(arg)`	Computes the average of a `DECIMAL` column using exact integer arithmetic with banker's rounding, returning a `DECIMAL` with the same width and scale as the input.
`favg(arg)`	Calculates the average using a more accurate floating point summation (Kahan Sum). This function is affected by ordering.
`first(arg)`	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
`fsum(arg)`	Calculates the sum using a more accurate floating point summation (Kahan Sum). This function is affected by ordering.
`geometric_mean(arg)`	Calculates the geometric mean of all non-null values in `arg`. This function is affected by ordering.
`histogram(arg)`	Returns a `MAP` of key-value pairs representing buckets and counts.
`histogram(arg, boundaries)`	Returns a `MAP` of key-value pairs representing the provided upper `boundaries` and counts of elements in the corresponding bins (left-open and right-closed partitions) of the datatype. A boundary at the largest value of the datatype is automatically added when elements larger than all provided `boundaries` appear, see `is_histogram_other_bin`. Boundaries may be provided, e.g., via `equi_width_bins`.
`histogram_exact(arg, elements)`	Returns a `MAP` of key-value pairs representing the requested elements and their counts. A catch-all element specific to the data-type is automatically added to count other elements when they appear, see `is_histogram_other_bin`.
`histogram_values(source, col_name, technique, bin_count)`	A table macro (not an aggregate function) returning the upper boundary of each bin and its count for a column; see the detailed entry for the available `technique` values.
`last(arg)`	Returns the last value of a column. This function is affected by ordering.
`list(arg)`	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
`lttb(x, y, n)`	Downsamples the series of `(x, y)` points to at most n representative points using the Largest-Triangle-Three-Buckets (LTTB) algorithm. This function is affected by ordering.
`max(arg)`	Returns the maximum value present in `arg`. This function is unaffected by distinctness.
`max(arg, n)`	Returns a `LIST` containing the `arg` values for the ”top” n rows ordered by `arg` descending.
`min(arg)`	Returns the minimum value present in `arg`. This function is unaffected by distinctness.
`min(arg, n)`	Returns a `LIST` containing the `arg` values for the ”bottom” n rows ordered by `arg` ascending.
`product(arg)`	Calculates the product of all non-null values in `arg`. This function is affected by ordering.
`string_agg(arg)`	Concatenates the column string values with a comma separator (`,`). This function is affected by ordering.
`string_agg(arg, sep)`	Concatenates the column string values with a separator. This function is affected by ordering.
`sum(arg)`	Calculates the sum of all non-null values in `arg` / counts `true` values when `arg` is boolean. The floating-point versions of this function are affected by ordering.

Function	Description
weighted_avg(arg, weight)	Calculates the weighted average of all non-null values in arg, where each value is scaled by its corresponding weight. If weight is NULL, the corresponding arg value will be skipped. This function is affected by ordering.

any_value(arg)

Description	Returns the first non-NULL value from arg. This function is affected by ordering.
Example	any_value(A)

arg_max(arg, val)

Description	Finds the row with the maximum val and calculates the arg expression at that row. Rows where the value of the arg or val expression is NULL are ignored. This function is affected by ordering.
Example	arg_max(A, B)
Alias(es)	argmax(arg, val),max_by(arg, val)

arg_max(arg, val, n)

Description	The generalized case of arg_max for n values: returns a LIST containing the arg expressions for the top n rows ordered by val descending. This function is affected by ordering.
Example	arg_max(A, B, 2)
Alias(es)	argmax(arg, val, n),max_by(arg, val, n)

arg_max_null(arg, val)

Description	Finds the row with the maximum val and calculates the arg expression at that row. Rows where the val expression evaluates to NULL are ignored. This function is affected by ordering.
Example	arg_max_null(A, B)

arg_max_nulls_last(arg, val[, n])

Description	Finds the row with the maximum val and calculates the arg expression at that row. Unlike arg_max, NULL values of arg are not ignored and rows where val is NULL are sorted last, i.e., they are only selected if there are no other rows. If n is specified, returns a LIST containing the arg expressions for the top n rows ordered by val descending, with NULL values of val last. This function is affected by ordering.
Example	arg_max_nulls_last(A, B)

arg_min(arg, val)

Description	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the value of the `arg` or `val` expression is `NULL` are ignored. This function is affected by ordering.
Example	`arg_min(A, B)`
Alias(es)	`argmin(arg, val)`,`min_by(arg, val)`

`arg_min(arg, val, n)`

Description	The generalized case of `arg_min` for n values: returns a `LIST` containing the `arg` expressions for the bottom n rows ordered by `val` ascending. This function is affected by ordering.
Example	`arg_min(A, B, 2)`
Alias(es)	`argmin(arg, val, n)`,`min_by(arg, val, n)`

`arg_min_null(arg, val)`

Description	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the `val` expression evaluates to `NULL` are ignored. This function is affected by ordering.
Example	`arg_min_null(A, B)`

`arg_min_nulls_last(arg, val[, n])`

Description	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Unlike `arg_min`, `NULL` values of `arg` are not ignored and rows where `val` is `NULL` are sorted last, i.e., they are only selected if there are no other rows. If n is specified, returns a `LIST` containing the `arg` expressions for the top n rows ordered by `val` ascending, with `NULL` values of `val` last. This function is affected by ordering.
Example	`arg_min_nulls_last(A, B)`

`avg(arg)`

Description	Calculates the average of all non-null values in `arg`. This function is affected by ordering.
Example	`avg(A)`
Alias(es)	`mean`

`bit_and(arg)`

Description	Returns the bitwise AND of all bits in a given expression.
Example	`bit_and(A)`

`bit_or(arg)`

Description	Returns the bitwise OR of all bits in a given expression.
Example	`bit_or(A)`

`bit_xor(arg)`

Description	Returns the bitwise XOR of all bits in a given expression.
Example	`bit_xor(A)`

`bitstring_agg(arg)`

Description	Returns a bitstring whose length corresponds to the range of the non-null (integer) values, with bits set at the location of each (distinct) value.
Example	`bitstring_agg(A)`

`bool_and(arg)`

Description	Returns `true` if every input value is `true`, otherwise `false`.
Example	`bool_and(A)`

`bool_or(arg)`

Description	Returns `true` if any input value is `true`, otherwise `false`.
Example	`bool_or(A)`

`count()`

Description	Returns the number of rows.
Example	`count()`
Alias(es)	`count(*)`

`count(arg)`

Description	Returns the number of rows where `arg` is not `NULL`.
Example	`count(A)`

`countif(arg)`

Description	Returns the number of rows where `arg` is `true`.
Example	`countif(A)`

`decimal_average(arg)`

Description	Computes the average of a `DECIMAL` column using exact integer arithmetic with banker's rounding, returning a `DECIMAL` with the same width and scale as the input. `NULL` values are ignored.
Example	`decimal_average(A)`

> New The `decimal_average` function was added in DuckDB 2.0.

`favg(arg)`

Description	Calculates the average using a more accurate floating point summation (Kahan Sum). This function is affected by ordering.
Example	`favg(A)`

`first(arg)`

Description	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
Example	`first(A)`
Alias(es)	`arbitrary(A)`

`fsum(arg)`

Description	Calculates the sum using a more accurate floating point summation (Kahan Sum). This function is affected by ordering.
Example	`fsum(A)`
Alias(es)	`sumkahan`, `kahan_sum`

`geometric_mean(arg)`

Description	Calculates the geometric mean of all non-null values in `arg`. This function is affected by ordering.
Example	`geometric_mean(A)`
Alias(es)	`geomean(A)`

`histogram(arg)`

Description	Returns a `MAP` of key-value pairs representing buckets and counts.
Example	`histogram(A)`

`histogram(arg, boundaries)`

Description	Returns a MAP of key-value pairs representing the provided upper `boundaries` and counts of elements in the corresponding bins (left-open and right-closed partitions) of the datatype. A boundary at the largest value of the datatype is automatically added when elements larger than all provided `boundaries` appear, see `is_histogram_other_bin`. Boundaries may be provided, e.g., via `equi_width_bins`.
Example	`histogram(A, [0, 1, 10])`

`histogram_exact(arg, elements)`

Description	Returns a MAP of key-value pairs representing the requested elements and their counts. A catch-all element specific to the data-type is automatically added to count other elements when they appear, see `is_histogram_other_bin`.
Example	`histogram_exact(A, ['a', 'b', 'c'])`

`histogram_values(source, col_name, technique, bin_count)`

Description	A table macro (not an aggregate function) that computes a histogram over column `col_name` of table `source`, returning the upper boundary of each bin and its count. The optional `technique` argument selects the binning method and is one of `auto`, `sample`, `equi-height`, `equi-width`, or `equi-width-nice`. The optional `bin_count` argument sets the number of bins.
Example	`histogram_values(integers, i, bin_count := 2)`

`last(arg)`

Description	Returns the last value of a column. This function is affected by ordering.
Example	`last(A)`

`list(arg)`

Description	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
Example	`list(A)`
Alias(es)	`array_agg`

`lttb(x, y, n)`

> New This feature was introduced in DuckDB 2.0.

Description	Downsamples the series of `(x, y)` points to at most n representative points using the Largest-Triangle-Three-Buckets (LTTB) algorithm, which preserves the visual shape of the series. Returns a `LIST` of `STRUCT`s with fields `x` and `y`, always including the first and the last point. `x` and `y` each accept `FLOAT`, `DOUBLE`, or any `TIMESTAMP` type, while the triangle computations are performed using `DOUBLE` arithmetic. `n` has to be a constant with `n >= 2`. Points where `x` or `y` is `NULL` are ignored and empty groups return `NULL`. The points have to be sorted on `x`, so `ORDER BY x` is required in practice. This function is affected by ordering.
Example	`lttb(x, y, 5 ORDER BY x)`

`max(arg)`

Description	Returns the maximum value present in `arg`. This function is unaffected by distinctness.
Example	`max(A)`

`max(arg, n)`

Description	Returns a `LIST` containing the `arg` values for the "top" n rows ordered by `arg` descending.
Example	`max(A, 2)`

`min(arg)`

Description	Returns the minimum value present in `arg`. This function is unaffected by distinctness.
Example	`min(A)`

`min(arg, n)`

Description	Returns a `LIST` containing the `arg` values for the "bottom" n rows ordered by `arg` ascending.
Example	`min(A, 2)`

`product(arg)`

Description	Calculates the product of all non-null values in `arg`. This function is affected by ordering.
Example	`product(A)`

`string_agg(arg)`

Description	Concatenates the column string values with a comma separator (`,`). This function is affected by ordering.
Example	`string_agg(S, ',')`
Alias(es)	`group_concat(arg)`, `listagg(arg)`

`string_agg(arg, sep)`

Description	Concatenates the column string values with a separator. This function is affected by ordering.
Example	`string_agg(S, ',')`
Alias(es)	`group_concat(arg, sep)`,`listagg(arg, sep)`

`sum(arg)`

Description	Calculates the sum of all non-null values in `arg` / counts `true` values when `arg` is boolean. The floating-point versions of this function are affected by ordering.
Example	`sum(A)`

`weighted_avg(arg, weight)`

Description	Calculates the weighted average of all non-null values in `arg`, where each value is scaled by its corresponding `weight`. If `weight` is `NULL`, the value will be skipped. This function is affected by ordering.
Example	`weighted_avg(A, W)`
Alias(es)	`wavg(arg, weight)`

Approximate Aggregates

The table below shows the available approximate aggregate functions.

Function	Description	Example
`approx_count_distinct(x)`	Calculates the approximate count of distinct elements using HyperLogLog.	`approx_count_distinct(A)`
`approx_quantile(x, pos)`	Calculates the approximate quantile using T-Digest.	`approx_quantile(A, 0.5)`
`approx_top_k(arg, k)`	Calculates a `LIST` of the `k` approximately most frequent values of `arg` using Filtered Space-Saving.	
`reservoir_quantile(x, quantile, sample_size = 8192)`	Calculates the approximate quantile using reservoir sampling, the sample size is optional and uses 8192 as a default size.	`reservoir_quantile(A, 0.5, 1024)`

Statistical Aggregates

The table below shows the available statistical aggregate functions. They all ignore `NULL` values (in the case of a single input column `x`), or pairs where either input is `NULL` (in the case of two input columns `y` and `x`).

Function	Description
`corr(y, x)`	The correlation coefficient.
`covar_pop(y, x)`	The population covariance, which does not include bias correction.
`covar_samp(y, x)`	The sample covariance, which includes Bessel's bias correction.
`entropy(x)`	The log-2 entropy of count input-values.
`kurtosis_pop(x)`	The excess kurtosis (Fisher's definition) without bias correction.
`kurtosis(x)`	The excess kurtosis (Fisher's definition) with bias correction according to the sample size.
`mad(x)`	The median absolute deviation. Temporal types return a positive `INTERVAL`.
`median(x)`	The middle value of the set. For even value counts, quantitative values are averaged and ordinal values return the lower value.
`mode(x)`	The most frequent value. This function is affected by ordering.
`quantile_cont(x, pos)`	The interpolated `pos`-quantile of `x` for `-1 <= pos <= 1`. Returns the `pos * (n_nonnull_values - 1)`th (zero-indexed, in the specified order) value of `x` or an interpolation between the adjacent values if the index is not an integer. Values of `pos` between `-1` and `0` correspond to counting backwards from `1`. More precisely, `quantile_cont(x, -y) = quantile_cont(x, 1 - y)`. Intuitively, arranges the values of `x` as equispaced *points* on a line, starting at `0` and ending at `1`, and returns the (interpolated) value at `pos`. This is Type 7 in Hyndman & Fan (1996). If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding interpolated quantiles.
`quantile(x, pos)`	Alias for `quantile_disc`.
`quantile_disc(x, pos)`	The discrete `pos`-quantile of `x` for `0 <= pos <= 1`. Returns the `greatest(ceil(pos * n_nonnull_values) - 1, 0)`th (zero-indexed, in the specified order) value of `x`. Intuitively, assigns to each value of `x` an equisized *sub-interval* (left-open and right-closed except for the initial interval) of the interval `[0, 1]`, and picks the value of the sub-interval that contains `pos`. This is Type 1 in Hyndman & Fan (1996). If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding discrete quantiles.
`regr_avgx(y, x)`	The average of the independent variable for non-`NULL` pairs, where `x` is the independent variable and `y` is the dependent variable.
`regr_avgy(y, x)`	The average of the dependent variable for non-`NULL` pairs, where `x` is the independent variable and `y` is the dependent variable.
`regr_count(y, x)`	The number of non-`NULL` pairs.
`regr_intercept(y, x)`	The intercept of the univariate linear regression line, where `x` is the independent variable and `y` is the dependent variable.
`regr_r2(y, x)`	The squared Pearson correlation coefficient between `y` and `x`. Also: The coefficient of determination in a linear regression, where `x` is the independent variable and `y` is the dependent variable.
`regr_slope(y, x)`	The slope of the linear regression line, where `x` is the independent variable and `y` is the dependent variable.
`regr_sxx(y, x)`	The sample variance, which includes Bessel's bias correction, of the independent variable for non-`NULL` pairs, where `x` is the independent variable and `y` is the dependent variable.
`regr_sxy(y, x)`	The sample covariance, which includes Bessel's bias correction.
`regr_syy(y, x)`	The sample variance, which includes Bessel's bias correction, of the dependent variable for non-`NULL` pairs, where `x` is the independent variable and `y` is the dependent variable.
`skewness(x)`	The skewness.
`sem(x)`	The standard error of the mean.
`stddev_pop(x)`	The population standard deviation.

Function	Description
`stddev_samp(x)`	The sample standard deviation.
`var_pop(x)`	The population variance, which does not include bias correction.
`var_samp(x)`	The sample variance, which includes Bessel's bias correction.

`corr(y, x)`

Description	The correlation coefficient.
Formula	`covar_pop(y, x) / (stddev_pop(x) * stddev_pop(y))`

`covar_pop(y, x)`

Description	The population covariance, which does not include bias correction.
Formula	`(sum(x*y) - sum(x) * sum(y) / regr_count(y, x)) / regr_count(y, x)`, `covar_samp(y, x) * (1 - 1 / regr_count(y, x))`

`covar_samp(y, x)`

Description	The sample covariance, which includes Bessel's bias correction.
Formula	`(sum(x*y) - sum(x) * sum(y) / regr_count(y, x)) / (regr_count(y, x) - 1)`, `covar_pop(y, x) / (1 - 1 / regr_count(y, x))`
Alias(es)	`regr_sxy(y, x)`

`entropy(x)`

Description	The log-2 entropy of count input-values.
Formula	-

`kurtosis_pop(x)`

Description	The excess kurtosis (Fisher's definition) without bias correction.
Formula	-

`kurtosis(x)`

Description	The excess kurtosis (Fisher's definition) with bias correction according to the sample size.
Formula	-

`mad(x)`

Description	The median absolute deviation. Temporal types return a positive `INTERVAL`.
Formula	`median(abs(x - median(x)))`

`median(x)`

Description	The middle value of the set. For even value counts, quantitative values are averaged and ordinal values return the lower value.
Formula	`quantile_cont(x, 0.5)`

`mode(x)`

Description	The most frequent value. This function is affected by ordering.
Formula	-

`quantile_cont(x, pos)`

Description	The interpolated `pos`-quantile of `x` for `-1 <= pos <= 1`. Returns the `pos * (n_nonnull_values - 1)`th (zero-indexed, in the specified order) value of `x` or an interpolation between the adjacent values if the index is not an integer. Values of `pos` between `-1` and `0` correspond to counting backwards from `1`. More precisely, `quantile_cont(x, -y) = quantile_cont(x, 1 - y)`. Intuitively, arranges the values of `x` as equispaced *points* on a line, starting at 0 and ending at 1, and returns the (interpolated) value at `pos`. This is Type 7 in Hyndman & Fan (1996). If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding interpolated quantiles.
Formula	-

`quantile(x, pos)`

Description	Alias for `quantile_disc`.

`quantile_disc(x, pos)`

Description	The discrete `pos`-quantile of `x` for `0 <= pos <= 1`. Returns the `greatest(ceil(pos * n_nonnull_values) - 1, 0)`th (zero-indexed, in the specified order) value of `x`. Intuitively, assigns to each value of `x` an equisized *sub-interval* (left-open and right-closed except for the initial interval) of the interval `[0, 1]`, and picks the value of the sub-interval that contains `pos`. This is Type 1 in Hyndman & Fan (1996). If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding discrete quantiles.
Formula	-
Alias(es)	`quantile`

`regr_avgx(y, x)`

Description	The average of the independent variable for non-NULL pairs, where x is the independent variable and y is the dependent variable.
Formula	-

`regr_avgy(y, x)`

Description	The average of the dependent variable for non-NULL pairs, where x is the independent variable and y is the dependent variable.
Formula	-

`regr_count(y, x)`

Description	The number of non-NULL pairs.
Formula	-

`regr_intercept(y, x)`

Description	The intercept of the univariate linear regression line, where x is the independent variable and y is the dependent variable.
Formula	`regr_avgy(y, x) - regr_slope(y, x) * regr_avgx(y, x)`

`regr_r2(y, x)`

Description	The squared Pearson correlation coefficient between y and x. Also: The coefficient of determination in a linear regression, where x is the independent variable and y is the dependent variable.
Formula	-

`regr_slope(y, x)`

Description	Returns the slope of the linear regression line, where x is the independent variable and y is the dependent variable.
Formula	`regr_sxy(y, x) / regr_sxx(y, x)`
Alias(es)	-

`regr_sxx(y, x)`

Description	The sample variance, which includes Bessel's bias correction, of the independent variable for non-NULL pairs, where x is the independent variable and y is the dependent variable.
Formula	-

regr_sxy(y, x)

Description	The sample covariance, which includes Bessel's bias correction.
Formula	`(sum(x*y) - sum(x) * sum(y) / regr_count(y, x)) / (regr_count(y, x) - 1),covar_pop(y, x) / (1 - 1 / regr_count(y, x))`
Alias(es)	`covar_samp(y, x)`

regr_syy(y, x)

Description	The sample variance, which includes Bessel's bias correction, of the dependent variable for non-NULL pairs, where x is the independent variable and y is the dependent variable.
Formula	-

sem(x)

Description	The standard error of the mean.
Formula	-

skewness(x)

Description	The skewness.
Formula	-

stddev_pop(x)

Description	The population standard deviation.
Formula	`sqrt(var_pop(x))`

stddev_samp(x)

Description	The sample standard deviation.
Formula	`sqrt(var_samp(x))`
Alias(es)	`stddev(x)`

var_pop(x)

Description	The population variance, which does not include bias correction.
Formula	`(sum(x^2) - sum(x)^2 / count(x)) / count(x),var_samp(y, x) * (1 - 1 / count(x))`

`var_samp(x)`

Description	The sample variance, which includes Bessel's bias correction.
Formula	`(sum(x^2) - sum(x)^2 / count(x)) / (count(x) - 1)`,`var_pop(y, x) / (1 - 1 / count(x))`
Alias(es)	`variance(arg, val)`

Ordered Set Aggregate Functions

The table below shows the available "ordered set" aggregate functions. These functions are specified using the `WITHIN GROUP (ORDER BY sort_expression)` syntax, and they are converted to an equivalent aggregate function that takes the ordering expression as the first argument.

Function	Equivalent
mode() WITHIN GROUP (ORDER BY column [(ASC|DESC)])	mode(column ORDER BY column [(ASC|DESC)])
percentile_cont(fraction) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_cont(column, fraction ORDER BY column [(ASC|DESC)])
percentile_cont(fractions) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_cont(column, fractions ORDER BY column [(ASC|DESC)])
percentile_disc(fraction) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_disc(column, fraction ORDER BY column [(ASC|DESC)])
percentile_disc(fractions) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_disc(column, fractions ORDER BY column [(ASC|DESC)])

Miscellaneous Aggregate Functions

Function	Description	Alias
`grouping()`	For queries with GROUP BY and either ROLLUP or GROUPING SETS: Returns an integer identifying which of the argument expressions were used to group on to create the current super-aggregate row.	`grouping_id()`

Array Functions

All LIST functions work with the ARRAY data type. Additionally, several ARRAY-native functions are also supported.

Array-Native Functions

Function	Description
`array_cosine_distance(array1, array2)`	Computes the cosine distance between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.

Function	Description
array_cosine_similarity(array1, array2)	Computes the cosine similarity between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.
array_cross_product(array, array)	Computes the cross product of two arrays of size 3. The array elements cannot be NULL.
array_distance(array1, array2)	Computes the distance between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.
array_dot_product(array1, array2)	Alias for array_inner_product.
array_inner_product(array1, array2)	Computes the inner product between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.
array_negative_dot_product(array1, array2)	Alias for array_negative_inner_product.
array_negative_inner_product(array1, array2)	Computes the negative inner product between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.
array_value(arg, ...)	Creates an ARRAY containing the argument values.

array_cosine_distance(array1, array2)

Description	Computes the cosine distance between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.
Example	array_cosine_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))
Result	0.007416606

array_cosine_similarity(array1, array2)

Description	Computes the cosine similarity between two arrays of the same size. The array elements cannot be NULL. The arrays can have any size as long as the size is the same for both arguments.
Example	array_cosine_similarity(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))
Result	0.9925834

array_cross_product(array, array)

Description	Computes the cross product of two arrays of size 3. The array elements cannot be NULL.
Example	array_cross_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))
Result	[-1.0, 2.0, -1.0]

`array_distance(array1, array2)`

Description	Computes the distance between two arrays of the same size. The array elements cannot be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`1.7320508`

`array_inner_product(array1, array2)`

Description	Computes the inner product between two arrays of the same size. The array elements cannot be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`20.0`
Alias	`array_dot_product`

`array_negative_inner_product(array1, array2)`

Description	Computes the negative inner product between two arrays of the same size. The array elements cannot be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_negative_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`-20.0`
Alias	`array_negative_dot_product`

`array_value(arg, ...)`

Description	Creates an ARRAY containing the argument values.
Example	`array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT)`
Result	`[1.0, 2.0, 3.0]`

Bitstring Functions

This section describes functions and operators for examining and manipulating `BITSTRING` values. Bitstrings must be of equal length when performing the bitwise operands AND, OR and XOR. When bit shifting, the original length of the string is preserved.

Bitstring Operators

The table below shows the available mathematical operators for `BIT` type.

Operator	Description	Example	Result
&	Bitwise AND	`'10101'::BITSTRING & '10001'::BITSTRING`	10001
|	Bitwise OR	`'1011'::BITSTRING | '0001'::BITSTRING`	1011
xor	Bitwise XOR	`xor('101'::BITSTRING, '001'::BITSTRING)`	100
~	Bitwise NOT	`~('101'::BITSTRING)`	010
<<	Bitwise shift left	`'1001011'::BITSTRING << 3`	1011000
>>	Bitwise shift right	`'1001011'::BITSTRING >> 3`	0001001

Bitstring Functions

The table below shows the available scalar functions for `BIT` type.

Name	Description
`bit_count(bitstring)`	Returns the number of set bits in the bitstring.
`bit_length(bitstring)`	Returns the number of bits in the bitstring.
`bit_position(substring, bitstring)`	Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1.
`bitstring(bitstring, length)`	Returns a bitstring of determined length.
`get_bit(bitstring, index)`	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.
`length(bitstring)`	Alias for `bit_length`.
`octet_length(bitstring)`	Returns the number of bytes in the bitstring.
`set_bit(bitstring, index, new_value)`	Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.

`bit_count(bitstring)`

Description	Returns the number of set bits in the bitstring.
Example	`bit_count('1101011'::BITSTRING)`
Result	5

`bit_length(bitstring)`

Description	Returns the number of bits in the bitstring.
Example	`bit_length('1101011'::BITSTRING)`
Result	7

`bit_position(substring, bitstring)`

Description	Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1
Example	`bit_position('010'::BITSTRING, '1110101'::BITSTRING)`
Result	`4`

`bitstring(bitstring, length)`

Description	Returns a bitstring of determined length.
Example	`bitstring('1010'::BITSTRING, 7)`
Result	`0001010`

`get_bit(bitstring, index)`

Description	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.
Example	`get_bit('0110010'::BITSTRING, 2)`
Result	`1`

`length(bitstring)`

Description	Alias for `bit_length`.
Example	`length('1101011'::BITSTRING)`
Result	`7`

`octet_length(bitstring)`

Description	Returns the number of bytes in the bitstring.
Example	`octet_length('1101011'::BITSTRING)`
Result	`1`

`set_bit(bitstring, index, new_value)`

Description	Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.
Example	`set_bit('0110010'::BITSTRING, 2, 0)`
Result	`0100010`

Bitstring Aggregate Functions

These aggregate functions are available for `BIT` type.

Name	Description
`bit_and(arg)`	Returns the bitwise AND operation performed on all bitstrings in a given expression.
`bit_or(arg)`	Returns the bitwise OR operation performed on all bitstrings in a given expression.
`bit_xor(arg)`	Returns the bitwise XOR operation performed on all bitstrings in a given expression.
`bitstring_agg(arg)`	Returns a bitstring with bits set for each distinct position defined in `arg`.
`bitstring_agg(arg, min, max)`	Returns a bitstring with bits set for each distinct position defined in `arg`. All positions must be within the range [`min`, `max`] or an `Out of Range Error` will be thrown.

`bit_and(arg)`

Description	Returns the bitwise AND operation performed on all bitstrings in a given expression.
Example	`bit_and(A)`

`bit_or(arg)`

Description	Returns the bitwise OR operation performed on all bitstrings in a given expression.
Example	`bit_or(A)`

`bit_xor(arg)`

Description	Returns the bitwise XOR operation performed on all bitstrings in a given expression.
Example	`bit_xor(A)`

`bitstring_agg(arg)`

Description	The `bitstring_agg` function takes any integer type as input and returns a bitstring with bits set for each distinct value. The left-most bit represents the smallest value in the column and the right-most bit the maximum value. If possible, the min and max are retrieved from the column statistics. Otherwise, it is also possible to provide the min and max values.
Example	`bitstring_agg(A)`

> **Tip.** The combination of `bit_count` and `bitstring_agg` can be used as an alternative to `count(DISTINCT ...)`, with possible performance improvements in cases of low cardinality and dense values.

`bitstring_agg(arg, min, max)`

Description	Returns a bitstring with bits set for each distinct position defined in `arg`. All positions must be within the range [`min`, `max`] or an `Out of Range Error` will be thrown.
Example	`bitstring_agg(A, 1, 42)`

Blob Functions

This section describes functions and operators for examining and manipulating BLOB values.

Function	Description
`arg1 || arg2`	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.
`base64(blob)`	Alias for `to_base64`.
`decode(blob[, on_error])`	Converts `blob` to VARCHAR. The optional `on_error` parameter controls handling of invalid UTF-8: `'strict'` (default, throws error), `'replace'` (replaces invalid characters with `?`), or `'ignore'` (removes invalid characters).
`encode(string)`	Converts the `string` to BLOB. Converts UTF-8 characters into literal encoding.
`from_base64(string)`	Converts a base64 encoded `string` to a character string (BLOB).
`from_binary(value)`	Alias for `unbin`.
`from_hex(value)`	Alias for `unhex`.
`hex(blob)`	Converts `blob` to VARCHAR using hexadecimal encoding.
`md5(blob)`	Returns the MD5 hash of the `blob` as a VARCHAR.
`md5_number(blob)`	Returns the MD5 hash of the `blob` as a HUGEINT.
`octet_length(blob)`	Number of bytes in `blob`.
`read_blob(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
`repeat(blob, count)`	Repeats the `blob` `count` number of times.
`sha1(blob)`	Returns a VARCHAR with the SHA-1 hash of the `blob`.
`sha256(blob)`	Returns a VARCHAR with the SHA-256 hash of the `blob`.
`to_base64(blob)`	Converts a `blob` to a base64 encoded string.
`to_hex(blob)`	Alias for `hex`.
`unbin(value)`	Converts a `value` from binary representation to a blob.
`unhex(value)`	Converts a `value` from hexadecimal representation to a blob.

`arg1 || arg2`

Description	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.
Example 1	`'Duck' || 'DB'`
Result	DuckDB
Example 2	`[1, 2, 3] || [4, 5, 6]`
Result	`[1, 2, 3, 4, 5, 6]`

Example 3	`'\xAA'::BLOB || '\xBB'::BLOB`
Result	`\xAA\xBB`

`decode(blob[, on_error])`

Description	Converts `blob` to VARCHAR. The optional `on_error` parameter controls handling of invalid UTF-8: `'strict'` (default, throws error), `'replace'` (replaces invalid characters with ?), or `'ignore'` (removes invalid characters).
Example	`decode('\xC3\xBC'::BLOB)`
Result	ü
Example	`decode('\xAA'::BLOB, 'replace')`
Result	?

`encode(string)`

Description	Converts the `string` to BLOB. Converts UTF-8 characters into literal encoding.
Example	`encode('my_string_with_ü')`
Result	`my_string_with_\xC3\xBC`

`from_base64(string)`

Description	Converts a base64 encoded `string` to a character string (BLOB).
Example	`from_base64('QQ==')`
Result	A

`hex(blob)`

Description	Converts `blob` to VARCHAR using hexadecimal encoding.
Example	`hex('\xAA\xBB'::BLOB)`
Result	AABB
Alias	`to_hex`

`md5(blob)`

Description	Returns the MD5 hash of the `blob` as a VARCHAR.
Example	`md5('\xAA\xBB'::BLOB)`
Result	`58cea1f6b2b06520613e09af90dc1c47`

`md5_number(blob)`

Description	Returns the MD5 hash of the `blob` as a `HUGEINT`.
Example	`md5_number('\xAA\xBB'::BLOB)`
Result	`94525045605907259200829535064523132504`

`octet_length(blob)`

Description	Number of bytes in `blob`.
Example	`octet_length('\xAA\xBB'::BLOB)`
Result	`2`

`read_blob(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a `BLOB`. See the `read_blob` guide for more details.
Example	`read_blob('hello.bin')`
Result	`hello\x0A`

`repeat(blob, count)`

Description	Repeats the `blob` `count` number of times.
Example	`repeat('\xAA\xBB'::BLOB, 5)`
Result	`\xAA\xBB\xAA\xBB\xAA\xBB\xAA\xBB\xAA\xBB`

`sha1(blob)`

Description	Returns a `VARCHAR` with the SHA-1 hash of the `blob`.
Example	`sha1('\xAA\xBB'::BLOB)`
Result	`65b1e351a6cbfeb41c927222bc9ef53aad3396b0`

`sha256(blob)`

Description	Returns a `VARCHAR` with the SHA-256 hash of the `blob`.
Example	`sha256('\xAA\xBB'::BLOB)`
Result	`d798d1fac6bd4bb1c11f50312760351013379a0ab6f0a8c0af8a506b96b2525a`

`to_base64(blob)`

Description	Converts a `blob` to a base64 encoded string.
Example	`to_base64('A'::BLOB)`
Result	`QQ==`
Alias	`base64`

`unbin(value)`

Description	Converts a `value` from binary representation to a blob.
Example	`unbin('0110')`
Result	`\x06`
Alias	`from_binary`

`unhex(value)`

Description	Converts a `value` from hexadecimal representation to a blob.
Example	`unhex('2A')`
Result	`*`
Alias	`from_hex`

Date Format Functions

The `strftime` and `strptime` functions can be used to convert between `DATE` / `TIMESTAMP` values and strings. This is often required when parsing CSV files, displaying output to the user or transferring information between programs. Because there are many possible date representations, these functions accept a format string that describes how the date or timestamp should be structured.

`strftime` Examples

The `strftime(timestamp, format)` converts timestamps or dates to strings according to the specified pattern.

```
SELECT strftime(DATE '1992-03-02', '%d/%m/%Y');
```

```
02/03/1992
```

```
SELECT strftime(TIMESTAMP '1992-03-02 20:32:45', '%A, %-d %B %Y - %I:%M:%S %p');
```

```
Monday, 2 March 1992 - 08:32:45 PM
```

`strptime` Examples

The `strptime(text, format)` function converts strings to timestamps according to the specified pattern.

```
SELECT strptime('02/03/1992', '%d/%m/%Y');
```

```
1992-03-02 00:00:00
```

```
SELECT strptime('Monday, 2 March 1992 - 08:32:45 PM', '%A, %-d %B %Y - %I:%M:%S %p');
```

```
1992-03-02 20:32:45
```

The `strptime` function throws an error on failure:

```
SELECT strptime('02/50/1992', '%d/%m/%Y') AS x;
```

```
Invalid Input Error:
Could not parse string "02/50/1992" according to format specifier "%d/%m/%Y"
02/50/1992
   ^
Error: Month out of range, expected a value between 1 and 12
```

To return `NULL` on failure, use the `try_strptime` function:

```
SELECT try_strptime('02/50/1992', '%d/%m/%Y') AS x;
```

```
NULL
```

CSV Parsing

The date formats can also be specified during CSV parsing, either in the COPY statement or in the `read_csv` function. This can be done by either specifying a `DATEFORMAT` or a `TIMESTAMPFORMAT` (or both). `DATEFORMAT` will be used for converting dates, and `TIMESTAMPFORMAT` will be used for converting timestamps. Below are some examples for how to use this.

In a COPY statement:

```
COPY dates FROM 'test.csv' (DATEFORMAT '%d/%m/%Y', TIMESTAMPFORMAT '%A, %-d %B %Y - %I:%M:%S %p');
```

In a `read_csv` function:

```
SELECT *
FROM read_csv('test.csv', dateformat = '%m/%d/%Y', timestampformat = '%A, %-d %B %Y - %I:%M:%S %p');
```

Format Specifiers

Below is a full list of all available format specifiers.

Specifier	Description	Example
%a	Abbreviated weekday name.	Sun, Mon, ...
%A	Full weekday name.	Sunday, Monday, ...
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...
%c	ISO date and time representation	1992-03-02 10:30:20
%d	Day of the month as a zero-padded decimal.	01, 02, ..., 31
%-d	Day of the month as a decimal number.	1, 2, ..., 30
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%g	Millisecond as a decimal number, zero-padded on the left.	000 - 999
%G	ISO 8601 year with century representing the year that contains the greater part of the ISO week (see %V).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23

Specifier	Description	Example
%-H	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%-I	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
%-j	Day of the year as a decimal number.	1, 2, ..., 366
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%-m	Month as a decimal number.	1, 2, ..., 12
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59
%n	Nanosecond as a decimal number, zero-padded on the left.	000000000 - 999999999
%p	Locale's AM or PM.	AM, PM
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%u	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, ..., 7
%U	Week number of the year. Week 01 starts on the first Sunday of the year, so there can be week 00. Note that this is not compliant with the week date standard in ISO 8601.	00, 01, ..., 53
%V	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4. Note that %V is incompatible with year directive %Y. Use the ISO year %G instead.	01, ..., 53
%w	Weekday as a decimal number.	0, 1, ..., 6
%W	Week number of the year. Week 01 starts on the first Monday of the year, so there can be week 00. Note that this is not compliant with the week date standard in ISO 8601.	00, 01, ..., 53
%x	ISO date representation	1992-03-02
%X	ISO time representation	10:30:20
%y	Year without century as a zero-padded decimal number. Numbers 00 to 68 are turned into 2000 to 2068. Numbers 69 to 99 are turned into 1969 to 1999.	00, 01, ..., 99
%-y	Year without century as a decimal number. Numbers 0 to 68 are turned into 2000 to 2068. Numbers 69 to 99 are turned into 1969 to 1999.	0, 1, ..., 99
%Y	Year with century as a decimal number.	2013, 2019 etc.
%z	Time offset from UTC in the form ±HH:MM, ±HHMM, or ±HH.	-0700
%Z	Time zone name.	Europe/Amsterdam
%%	A literal % character.	%

Date Functions

This section describes functions and operators for examining and manipulating DATE values.

Date Operators

The table below shows the available mathematical operators for DATE types.

Operator	Description	Example	Result
+	addition of days (integers)	`DATE '1992-03-22' + 5`{:.language-sql .highlight}	`1992-03-27`
+	addition of AN `INTERVAL`	`DATE '1992-03-22' + INTERVAL 5 DAY`{:.language-sql .highlight}	`1992-03-27 00:00:00`
+	addition of a variable `INTERVAL`	`SELECT DATE '1992-03-22' + INTERVAL (d.days) DAY FROM (VALUES (5), (11)) d(days)`{:.language-sql .highlight}	`1992-03-27 00:00:00` and `1992-04-02 00:00:00`
-	subtraction of DATEs	`DATE '1992-03-27' - DATE '1992-03-22'`{:.language-sql .highlight}	`5`
-	subtraction of an `INTERVAL`	`DATE '1992-03-27' - INTERVAL 5 DAY`{:.language-sql .highlight}	`1992-03-22 00:00:00`
-	subtraction of a variable `INTERVAL`	`SELECT DATE '1992-03-27' - INTERVAL (d.days) DAY FROM (VALUES (5), (11)) d(days)`{:.language-sql .highlight}	`1992-03-22 00:00:00` and `1992-03-16 00:00:00`

Adding to or subtracting from infinite values produces the same infinite value.

Date Functions

The table below shows the available functions for DATE types. Dates can also be manipulated with the timestamp functions through type promotion.

Name	Description
`date_add(date, interval)`	Add the interval to the date and return a `DATETIME` value.
`date_diff(part, startdate, enddate)`	The number of `part` boundaries between `startdate` and `enddate`, inclusive of the larger date and exclusive of the smaller date.
`date_part(part, date)`	Get subfield (equivalent to `extract`).
`date_sub(part, startdate, enddate)`	The signed length of the interval between `startdate` and `enddate`, truncated to whole multiples of `part`.
`date_trunc(part, date)`	Truncate to specified precision.
`dayname(date)`	The (English) name of the weekday.
`days_in_month(date)`	The number of days in the month of the given date.
`extract(part from date)`	Get subfield from a date.
`greatest(date, date)`	The later of two dates.
`isfinite(date)`	Returns true if the date is finite, false otherwise.
`isinf(date)`	Returns true if the date is infinite, false otherwise.
`julian(date)`	Extract the Julian Day number from a date.
`last_day(date)`	The last day of the corresponding month in the date.

Name	Description
least(date, date)	The earlier of two dates.
make_date(year, month, day)	The date for the given parts.
monthname(date)	The (English) name of the month.
strftime(date, format)	Converts a date to a string according to the format string.
time_bucket(bucket_width, date[, offset])	Truncate date to a grid of width bucket_width. The grid is anchored at 2000-01-01[+ offset] when bucket_width is a number of months or coarser units, else 2000-01-03[+ offset]. Note that 2000-01-03 is a Monday.
time_bucket(bucket_width, date[, origin])	Truncate timestamptz to a grid of width bucket_width. The grid is anchored at the origin timestamp, which defaults to 2000-01-01 when bucket_width is a number of months or coarser units, else 2000-01-03. Note that 2000-01-03 is a Monday.
today()	Current date (start of current transaction) in the local time zone.

date_add(date, interval)

Description	Add the interval to the date and return a DATETIME value.
Example	date_add(DATE '1992-09-15', INTERVAL 2 MONTH)
Result	1992-11-15 00:00:00

date_diff(part, startdate, enddate)

Description	The number of part boundaries between startdate and enddate, inclusive of the larger date and exclusive of the smaller date.
Example	date_diff('month', DATE '1992-09-15', DATE '1992-11-14')
Result	2
Alias	datediff

date_part(part, date)

Description	Get the subfield (equivalent to extract).
Example	date_part('year', DATE '1992-09-20')
Result	1992
Alias	datepart

date_sub(part, startdate, enddate)

Description	The signed length of the interval between startdate and enddate, truncated to whole multiples of part.
Example	date_sub('month', DATE '1992-09-15', DATE '1992-11-14')
Result	1

Alias	`datesub`

`date_trunc(part, date)`

Description	Truncate to specified precision. Always returns a `TIMESTAMP`, even when the input is a `DATE`.
Example	`date_trunc('month', DATE '1992-03-07')`
Result	`1992-03-01 00:00:00`
Alias	`datetrunc`

`dayname(date)`

Description	The (English) name of the weekday.
Example	`dayname(DATE '1992-09-20')`
Result	Sunday

`days_in_month(date)`

Description	The number of days in the month of the given date.
Example	`days_in_month(DATE '1992-02-15')`
Result	`29`

`extract(part from date)`

Description	Get subfield from a date.
Example	`extract('year' FROM DATE '1992-09-20')`
Result	`1992`

`greatest(date, date)`

Description	The later of two dates.
Example	`greatest(DATE '1992-09-20', DATE '1992-03-07')`
Result	`1992-09-20`

`isfinite(date)`

Description	Returns `true` if the date is finite, false otherwise.
Example	`isfinite(DATE '1992-03-07')`
Result	`true`

`isinf(date)`

Description	Returns `true` if the date is infinite, false otherwise.
Example	`isinf(DATE '-infinity')`
Result	`true`

`julian(date)`

Description	Extract the Julian Day number from a date.
Example	`julian(DATE '1992-09-20')`
Result	`2448886.0`

`last_day(date)`

Description	The last day of the corresponding month in the date.
Example	`last_day(DATE '1992-09-20')`
Result	`1992-09-30`

`least(date, date)`

Description	The earlier of two dates.
Example	`least(DATE '1992-09-20', DATE '1992-03-07')`
Result	`1992-03-07`

`make_date(year, month, day)`

Description	The date for the given parts.
Example	`make_date(1992, 9, 20)`
Result	`1992-09-20`

`monthname(date)`

Description	The (English) name of the month.
Example	`monthname(DATE '1992-09-20')`
Result	`September`

`strftime(date, format)`

Description	Converts a date to a string according to the format string.
Example	`strftime(DATE '1992-01-01', '%a, %-d %B %Y')`

Result	`Wed, 1 January 1992`

`time_bucket(bucket_width, date[, offset])`

Description	Truncate `date` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03[ + offset]`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 months', DATE '1992-04-20', INTERVAL '1 month')`
Result	`1992-04-01`

`time_bucket(bucket_width, date[, origin])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01` when `bucket_width` is a number of months or coarser units, else `2000-01-03`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 weeks', DATE '1992-04-20', DATE '1992-04-01')`
Result	`1992-04-15`

`today()`

Description	Current date (start of current transaction) in the local time zone.
Example	`today()`
Result	`2022-10-08`
Alias	`current_date` (no parentheses necessary)

Date Part Extraction Functions

There are also dedicated extraction functions to get the subfields. A few examples include extracting the day from a date, or the day of the week from a date.

Functions applied to infinite dates will either return the same infinite dates (e.g., `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite date, the result will be `NULL`.

Date Part Functions

The `date_part`, `date_trunc` and `date_diff` functions can be used to extract or manipulate parts of temporal types such as `TIMESTAMP`, `TIMESTAMPTZ`, `DATE` and `INTERVAL`.

The parts to be extracted or manipulated are specified by one of the strings in the tables below. The example column provides the corresponding parts of the timestamp `2021-08-03 11:59:44.123456`. Only the entries of the first table can be extracted from `INTERVAL`s or used to construct them.

> Except for `julian` and `epoch`, which return DOUBLEs, all parts are extracted as integers. Since there are no infinite integer values in DuckDB, `NULL`s are returned for infinite timestamps.

Part Specifiers Usable as Date Part Specifiers and in Intervals

Specifier	Description	Synonyms	Example
`century`	Gregorian century	`cent`, `centuries`, `c`	21
`day`	Gregorian day	`days`, `d`, `dayofmonth`	3
`decade`	Gregorian decade	`dec`, `decades`, `decs`	202
`hour`	Hours	`hr`, `hours`, `hrs`, `h`	11
`microseconds`	Sub-minute microseconds	`microsecond`, `us`, `usec`, `usecs`, `usecond`, `useconds`	44123456
`millennium`	Gregorian millennium	`mil`, `millenniums`, `millennia`, `mils`, `millenium`	3
`milliseconds`	Sub-minute milliseconds	`millisecond`, `ms`, `msec`, `msecs`, `msecond`, `mseconds`	44123
`minute`	Minutes	`min`, `minutes`, `mins`, `m`	59
`month`	Gregorian month	`mon`, `months`, `mons`	8
`quarter`	Quarter of the year (1-4)	`quarters`	3
`second`	Seconds	`sec`, `seconds`, `secs`, `s`	44
`year`	Gregorian year	`yr`, `y`, `years`, `yrs`	2021

Part Specifiers Only Usable as Date Part Specifiers

Specifier	Description	Synonyms	Example
`dayofweek`	Day of the week (Sunday = 0, Saturday = 6)	`weekday`, `dow`	2
`dayofyear`	Day of the year (1-365/366)	`doy`	215
`epoch`	Seconds since 1970-01-01		1627991984.123456
`era`	Gregorian era (CE/AD, BCE/BC)		1
`isodow`	ISO day of the week (Monday = 1, Sunday = 7)		2
`isoyear`	ISO Year number (Starts on Monday of week containing Jan 4th)		2021
`julian`	Julian Day number.		2459430.4998162435
`timezone_hour`	Time zone offset hour portion		0
`timezone_minute`	Time zone offset minute portion		0

Specifier	Description	Synonyms	Example
`timezone`	Time zone offset in seconds		`0`
`week`	Week number	`weeks`, `w`	`31`
`yearweek`	ISO year and week number in `YYYYWW` format		`202131`

Note that the time zone parts are all zero unless a time zone extension such as ICU has been installed to support `TIMESTAMP WITH TIME ZONE`.

Part Functions

There are dedicated extraction functions to get certain subfields:

Name	Description
`century(date)`	Century.
`day(date)`	Day.
`dayofmonth(date)`	Day (synonym).
`dayofweek(date)`	Numeric weekday (Sunday = 0, Saturday = 6).
`dayofyear(date)`	Day of the year (starts from 1, i.e., January 1 = 1).
`decade(date)`	Decade (year / 10).
`epoch(date)`	Seconds since 1970-01-01.
`era(date)`	Calendar era.
`hour(date)`	Hours.
`isodow(date)`	Numeric ISO weekday (Monday = 1, Sunday = 7).
`isoyear(date)`	ISO Year number (Starts on Monday of week containing Jan 4th).
`julian(date)`	DOUBLE Julian Day number.
`microsecond(date)`	Sub-minute microseconds.
`millennium(date)`	Millennium.
`millisecond(date)`	Sub-minute milliseconds.
`minute(date)`	Minutes.
`month(date)`	Month.
`quarter(date)`	Quarter.
`second(date)`	Seconds.
`timezone_hour(date)`	Time zone offset hour portion.
`timezone_minute(date)`	Time zone offset minutes portion.
`timezone(date)`	Time zone offset in seconds.
`week(date)`	ISO Week.
`weekday(date)`	Numeric weekday synonym (Sunday = 0, Saturday = 6).
`weekofyear(date)`	ISO Week (synonym).
`year(date)`	Year.

Name	Description
yearweek(date)	BIGINT of combined ISO Year number and 2-digit version of ISO Week number.

century(date)

Description	Century.
Example	century(DATE '1992-02-15')
Result	20

day(date)

Description	Day.
Example	day(DATE '1992-02-15')
Result	15

dayofmonth(date)

Description	Day (synonym).
Example	dayofmonth(DATE '1992-02-15')
Result	15

dayofweek(date)

Description	Numeric weekday (Sunday = 0, Saturday = 6).
Example	dayofweek(DATE '1992-02-15')
Result	6

dayofyear(date)

Description	Day of the year (starts from 1, i.e., January 1 = 1).
Example	dayofyear(DATE '1992-02-15')
Result	46

decade(date)

Description	Decade (year / 10).
Example	decade(DATE '1992-02-15')
Result	199

epoch(date)

Description	Seconds since 1970-01-01.
Example	`epoch(DATE '1992-02-15')`
Result	`698112000.0`

era(date)

Description	Calendar era.
Example	`era(DATE '0044-03-15 (BC)')`
Result	`0`

hour(date)

Description	Hours.
Example	`hour(timestamp '2021-08-03 11:59:44.123456')`
Result	`11`

isodow(date)

Description	Numeric ISO weekday (Monday = 1, Sunday = 7).
Example	`isodow(DATE '1992-02-15')`
Result	`6`

isoyear(date)

Description	ISO Year number (Starts on Monday of week containing Jan 4th).
Example	`isoyear(DATE '2022-01-01')`
Result	`2021`

julian(date)

Description	DOUBLE Julian Day number.
Example	`julian(DATE '1992-09-20')`
Result	`2448886.0`

microsecond(date)

Description	Sub-minute microseconds.
Example	`microsecond(timestamp '2021-08-03 11:59:44.123456')`

Result	44123456

millennium(date)

Description	Millennium.
Example	millennium(DATE '1992-02-15')
Result	2

millisecond(date)

Description	Sub-minute milliseconds.
Example	millisecond(timestamp '2021-08-03 11:59:44.123456')
Result	44123

minute(date)

Description	Minutes.
Example	minute(timestamp '2021-08-03 11:59:44.123456')
Result	59

month(date)

Description	Month.
Example	month(DATE '1992-02-15')
Result	2

quarter(date)

Description	Quarter.
Example	quarter(DATE '1992-02-15')
Result	1

second(date)

Description	Seconds.
Example	second(timestamp '2021-08-03 11:59:44.123456')
Result	44

`timezone_hour(date)`

Description	Time zone offset hour portion. Only meaningful for `TIMESTAMP WITH TIME ZONE` values, where the offset is computed for the current `TimeZone` setting. Not supported for `DATE` values.
Example	`timezone_hour(TIMESTAMPTZ '1992-02-15 01:00:00+00')`
Result	`-5` (with `TimeZone` set to `America/New_York`)

`timezone_minute(date)`

Description	Time zone offset minutes portion. Only meaningful for `TIMESTAMP WITH TIME ZONE` values, where the offset is computed for the current `TimeZone` setting. Not supported for `DATE` values.
Example	`timezone_minute(TIMESTAMPTZ '1992-02-15 01:00:00+00')`
Result	`30` (with `TimeZone` set to `Asia/Kolkata`)

`timezone(date)`

Description	Time zone offset in seconds. Only meaningful for `TIMESTAMP WITH TIME ZONE` values, where the offset is computed for the current `TimeZone` setting. Not supported for `DATE` values.
Example	`timezone(TIMESTAMPTZ '1992-02-15 01:00:00+00')`
Result	`-18000` (with `TimeZone` set to `America/New_York`)

`week(date)`

Description	ISO Week.
Example	`week(DATE '1992-02-15')`
Result	7

`weekday(date)`

Description	Numeric weekday synonym (Sunday = 0, Saturday = 6).
Example	`weekday(DATE '1992-02-15')`
Result	6

`weekofyear(date)`

Description	ISO Week (synonym).
Example	`weekofyear(DATE '1992-02-15')`
Result	7

`year(date)`

Description	Year.
Example	`year(DATE '1992-02-15')`
Result	`1992`

`yearweek(date)`

Description	`BIGINT` of combined ISO Year number and 2-digit version of ISO Week number.
Example	`yearweek(DATE '1992-02-15')`
Result	`199207`

Enum Functions

This section describes functions and operators for examining and manipulating ENUM values. The examples assume an enum type created as:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy', 'anxious');
```

These functions can take `NULL` or a specific value of the type as argument(s). Most enum functions return a value that depends only on the type of the argument, and not on its value. The functions `enum_code()` and `enum_range_boundary()` are exceptions; they return a value that depends on the argument value, as well as its type.

Name	Description
`enum_code(enum_value)`	Returns the (0-based) numeric value backing the given enum value.
`enum_first(enum)`	Returns the first value of the input enum type.
`enum_last(enum)`	Returns the last value of the input enum type.
`enum_range(enum)`	Returns all values of the input enum type as an array.
`enum_range_boundary(enum, enum)`	Returns the range between the two given enum values as an array.

`enum_code(enum_value)`

Description	Returns the (0-based) numeric value backing the given enum value.
Example	`enum_code('happy'::mood)`
Result	2

`enum_first(enum)`

Description	Returns the first value of the input enum type.
Example	`enum_first(NULL::mood)`
Result	`sad`

`enum_last(enum)`

Description	Returns the last value of the input enum type.
Example	`enum_last(NULL::mood)`
Result	`anxious`

`enum_range(enum)`

Description	Returns all values of the input enum type as an array.
Example	`enum_range(NULL::mood)`
Result	`[sad, ok, happy, anxious]`

`enum_range_boundary(enum, enum)`

Description	Returns the range between the two given enum values as an array. The values must be of the same enum type. When the first parameter is `NULL`, the result starts with the first value of the enum type. When the second parameter is `NULL`, the result ends with the last value of the enum type.
Example	`enum_range_boundary(NULL, 'happy'::mood)`
Result	`[sad, ok, happy]`

Geometry Functions

This section describes the functions for examining and manipulating `GEOMETRY` values.

> The `spatial` extension provides additional functions for working with `GEOMETRY` values, which are documented in the Spatial Functions section.

Geometry Operators

The table below lists the operators that can be used with `GEOMETRY` values.

Operator	Description	Example	Result
&&	Returns true if the geometries bounding boxes intersect. Equivalent to `ST_Intersects_Extent`.	`'POINT(5 5)'::GEOMETRY && 'LINESTRING(0 0, 10 20)'::GEOMETRY`	`true`

Built-in Geometry Functions

Name	Description
`ST_GeomFromWKB`	Creates a geometry from Well-Known Binary (WKB) representation
`ST_AsWKB`	Returns the Well-Known Binary (WKB) representation of the geometry

Name	Description
ST_AsWKT	Returns the Well-Known Text (WKT) representation of the geometry
ST_Intersects_Extent	Returns true if the geometries bounding boxes intersect
ST_CRS	Returns the Coordinate Reference System (CRS) identifier of the geometry
ST_SetCRS	Sets the Coordinate Reference System (CRS) identifier of the geometry

ST_GeomFromWKB function

Description	Creates a geometry from Well-Known Binary (WKB) representation
Example	`ST_GeomFromWKB('\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xF0?\x00\x00\x00\x00`
Result	`POINT(1 2)`

ST_AsWKB function

Description	Returns the Well-Known Binary (WKB) representation of the geometry
Example	`ST_AsWKB('POINT(1 2)'::GEOMETRY)`
Result	`\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xF0?\x00\x00\x00\x00\x00\x00\x00`
Alias	`ST_AsBinary`

ST_AsWKT function

Description	Returns the Well-Known Text (WKT) representation of the geometry
Example	`ST_AsText('POINT(1 2)'::GEOMETRY)`
Result	`POINT (1 2)`
Alias	`ST_AsText`

ST_Intersects_Extent function

Description	Returns true if the geometries bounding boxes intersect
Example	`'POINT(5 5)'::GEOMETRY && 'LINESTRING(0 0, 10 20)'::GEOMETRY`
Result	`true`
Alias	`&&`

ST_CRS function

Description	Returns the Coordinate Reference System (CRS) identifier of the geometry

Example	`ST_CRS('POINT(1 2)'::GEOMETRY('OGC:CRS84'))`
Result	`OGC:CRS84`

`ST_SetCRS` function

Description	Sets the Coordinate Reference System (CRS) identifier of the geometry
Example	`typeof(ST_SetCRS('POINT(1 2)'::GEOMETRY, 'OGC:CRS84'))`
Result	`GEOMETRY('OGC:CRS84')`

Interval Functions

This section describes functions and operators for examining and manipulating `INTERVAL` values.

Interval Operators

The table below shows the available mathematical operators for `INTERVAL` types.

Operator	Description	Example	Result
`+`	Addition of an `INTERVAL`	`INTERVAL 1 HOUR + INTERVAL 5 HOUR`	`INTERVAL 6 HOUR`
`+`	Addition to a `DATE`	`DATE '1992-03-22' + INTERVAL 5 DAY`	`1992-03-27 00:00:00`
`+`	Addition to a `TIMESTAMP`	`TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`+`	Addition to a `TIME`	`TIME '01:02:03' + INTERVAL 5 HOUR`	`06:02:03`
`-`	Subtraction of an `INTERVAL`	`INTERVAL 5 HOUR - INTERVAL 1 HOUR`	`INTERVAL 4 HOUR`
`-`	Subtraction from a `DATE`	`DATE '1992-03-27' - INTERVAL 5 DAY`	`1992-03-22 00:00:00`
`-`	Subtraction from a `TIMESTAMP`	`TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`
`-`	Subtraction from a `TIME`	`TIME '06:02:03' - INTERVAL 5 HOUR`	`01:02:03`

Interval Functions

The table below shows the available scalar functions for `INTERVAL` types.

Name	Description
`date_part(part, interval)`	Extract datepart component (equivalent to `extract`). See `INTERVAL` for the sometimes surprising rules governing this extraction.
`datepart(part, interval)`	Alias of `date_part`.
`extract(part FROM interval)`	Alias of `date_part`.
`epoch(interval)`	Get total number of seconds, as double precision floating point number, in interval.

Name	Description
`normalized_interval(interval)`	Normalizes an interval so that units that overflow their range are carried over into the next larger unit.
`to_centuries(integer)`	Construct a century interval.
`to_days(integer)`	Construct a day interval.
`to_decades(integer)`	Construct a decade interval.
`to_hours(integer)`	Construct an hour interval.
`to_microseconds(integer)`	Construct a microsecond interval.
`to_millennia(integer)`	Construct a millennium interval.
`to_milliseconds(integer)`	Construct a millisecond interval.
`to_minutes(integer)`	Construct a minute interval.
`to_months(integer)`	Construct a month interval.
`to_quarters(integer)`	Construct an interval of `integer` quarters.
`to_seconds(integer)`	Construct a second interval.
`to_weeks(integer)`	Construct a week interval.
`to_years(integer)`	Construct a year interval.

> Only the documented date part components are defined for intervals.

`date_part(part, interval)`

Description	Extract datepart component (equivalent to `extract`). See `INTERVAL` for the sometimes surprising rules governing this extraction.
Example	`date_part('year', INTERVAL '14 months')`
Result	1

`datepart(part, interval)`

Description	Alias of `date_part`.
Example	`datepart('year', INTERVAL '14 months')`
Result	1

`extract(part FROM interval)`

Description	Alias of `date_part`.
Example	`extract('month' FROM INTERVAL '14 months')`
Result	2

`epoch(interval)`

Description	Get total number of seconds, as double precision floating point number, in interval.
Example	`epoch(INTERVAL 5 HOUR)`
Result	`18000.0`

`normalized_interval(interval)`

Description	Normalizes an interval so that units that overflow their range are carried over into the next larger unit.
Example	`normalized_interval(INTERVAL '30 days')`
Result	`INTERVAL 1 MONTH`

`to_centuries(integer)`

Description	Construct a century interval.
Example	`to_centuries(5)`
Result	`INTERVAL 500 YEAR`

`to_days(integer)`

Description	Construct a day interval.
Example	`to_days(5)`
Result	`INTERVAL 5 DAY`

`to_decades(integer)`

Description	Construct a decade interval.
Example	`to_decades(5)`
Result	`INTERVAL 50 YEAR`

`to_hours(integer)`

Description	Construct an hour interval.
Example	`to_hours(5)`
Result	`INTERVAL 5 HOUR`

`to_microseconds(integer)`

Description	Construct a microsecond interval.
Example	`to_microseconds(5)`

Result	`INTERVAL 5 MICROSECOND`

`to_millennia(integer)`

Description	Construct a millennium interval.
Example	`to_millennia(5)`
Result	`INTERVAL 5000 YEAR`

`to_milliseconds(integer)`

Description	Construct a millisecond interval.
Example	`to_milliseconds(5)`
Result	`INTERVAL 5 MILLISECOND`

`to_minutes(integer)`

Description	Construct a minute interval.
Example	`to_minutes(5)`
Result	`INTERVAL 5 MINUTE`

`to_months(integer)`

Description	Construct a month interval.
Example	`to_months(5)`
Result	`INTERVAL 5 MONTH`

`to_quarters(integer)`

Description	Construct an interval of `integer` quarters.
Example	`to_quarters(5)`
Result	`INTERVAL 1 YEAR 3 MONTHS`

`to_seconds(integer)`

Description	Construct a second interval.
Example	`to_seconds(5)`
Result	`INTERVAL 5 SECOND`

`to_weeks(integer)`

Description	Construct a week interval.
Example	`to_weeks(5)`
Result	`INTERVAL 35 DAY`

`to_years(integer)`

Description	Construct a year interval.
Example	`to_years(5)`
Result	`INTERVAL 5 YEAR`

Lambda Functions

DuckDB lambda functions (a.k.a. 'lambdas' or 'lambda expressions') are expressions that define an inline anonymous function. Lambdas are passed as argument to certain list functions, which evaluate the passed lambda expression against its elements. The value resulting from the lambda's evaluation is then processed further by the list function to create its return value.

Lambdas may also be used in the `COLUMNS` clause. In that context, the lambda function acts as a filter: if it evaluates to `true`, the column is retained, and discarded otherwise. `COLUMNS`-clause lambdas have a mandatory parameter that receives the column name. Optionally, a second parameter may be declared that receives the (1-based) column index.

The syntax for a lambda expression is:

```
lambda <parameters> : <expression>
```

- `<parameters>`{:.language-sql .highlight} is a comma-separated list of parameter names. Parameters may be given any name, and their name may be referenced in the `<expression>`{:.language-sql .highlight}.
- The number of allowed and required parameters depends on the list function to which it is passed.
- There is always a mandatory parameter that is used by the list function to pass the current element of the list. This is typically the first parameter.
- There is always an optional parameter used by the list function to pass the (1-based) index of the current element. This is always the last parameter.
- The expression must be a scalar expression. Additionally, it cannot contain a subquery, table function, and the like.
- The expression may contain macro invocations. However, only scalar macros are allowed, and only if they do not rely on a subquery.

For example, the following are all valid lambda functions:

- `lambda param : param > 1`{:.language-sql .highlight}
- `lambda s : contains(concat(s, 'DB'), 'duck')`{:.language-sql .highlight}
- `lambda acc, x : acc + x`{:.language-sql .highlight}

Lambda functions enable the use of more complex and flexible expressions in queries.

Scalar Functions That Accept Lambda Functions

Function	Description
`apply(list, lambda(x))`	Alias for `list_transform`.

Function	Description
array_apply(list, lambda(x))	Alias for list_transform.
array_filter(list, lambda(x))	Alias for list_filter.
array_reduce(list, lambda(x,y)[, initial_value])	Alias for list_reduce.
array_transform(list, lambda(x))	Alias for list_transform.
filter(list, lambda(x))	Alias for list_filter.
list_apply(list, lambda(x))	Alias for list_transform.
list_filter(list, lambda(x))	Constructs a list from those elements of the input list for which the lambda function returns true. DuckDB must be able to cast the lambda function's return type to BOOL. The return type of list_filter is the same as the input list's. See list_filter examples.
list_reduce(list, lambda(x,y)[, initial_value])	Reduces all elements of the input list into a single scalar value by executing the lambda function on a running result and the next list element. The lambda function has an optional initial_value argument. See list_reduce examples.
list_transform(list, lambda(x))	Returns a list that is the result of applying the lambda function to each element of the input list. The return type is defined by the return type of the lambda function. See list_transform examples.
reduce(list, lambda(x,y)[, initial_value])	Alias for list_reduce.

list_filter(list, lambda(x))

Description	Constructs a list from those elements of the input list for which the lambda function returns true. DuckDB must be able to cast the lambda function's return type to BOOL. The return type of list_filter is the same as the input list's. See list_filter examples.
Example	list_filter([3, 4, 5], lambda x : x > 4)
Result	[5]
Aliases	array_filter, filter

list_reduce(list, lambda(x,y)[, initial_value])

Description	Reduces all elements of the input list into a single scalar value by executing the lambda function on a running result and the next list element. The lambda function has an optional initial_value argument. See list_reduce examples.
Example	list_reduce([1, 2, 3], lambda x, y : x + y)
Result	6
Aliases	array_reduce, reduce

`list_transform(list, lambda(x))`

Description	Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function. See `list_transform` examples.
Example	`list_transform([1, 2, 3], lambda x : x + 1)`
Result	`[2, 3, 4]`
Aliases	`apply`, `array_apply`, `array_transform`, `list_apply`

Nesting Lambda Functions

All scalar functions can be arbitrarily nested. For example, nested lambda functions to get all squares of even list elements:

```
SELECT list_transform(
        list_filter([0, 1, 2, 3, 4, 5], lambda x: x % 2 = 0),
        lambda y: y * y
    );
```

```
[0, 4, 16]
```

Nested lambda function to add each element of the first list to the sum of the second list:

```
SELECT list_transform(
        [1, 2, 3],
        lambda x :
            list_reduce([4, 5, 6], lambda a, b: a + b) + x
    );
```

```
[16, 17, 18]
```

Scoping

Lambda functions conform to scoping rules in the following order:

- inner lambda parameters
- outer lambda parameters
- column names
- macro parameters

```
CREATE TABLE tbl (x INTEGER);
INSERT INTO tbl VALUES (10);
SELECT list_apply(
            [1, 2],
            lambda x: list_apply([4], lambda x: x + tbl.x)[1] + x
    )
FROM tbl;
```

```
[15, 16]
```

Indexes as Parameters

All lambda functions accept an optional extra parameter that represents the index of the current element. This is always the last parameter of the lambda function (e.g., `i` in `(x, i)`), and is 1-based (i.e., the first element has index 1).

Get all elements that are larger than their index:

```
SELECT list_filter([1, 3, 1, 5], lambda x, i: x > i);
```

```
[3, 5]
```

Examples

`list_transform` Examples

Incrementing each list element by one:

```
SELECT list_transform([1, 2, NULL, 3], lambda x: x + 1);
```

```
[2, 3, NULL, 4]
```

Transforming strings:

```
SELECT list_transform(['Duck', 'Goose', 'Sparrow'], lambda s: concat(s, 'DB'));
```

```
[DuckDB, GooseDB, SparrowDB]
```

Combining lambda functions with other functions:

```
SELECT list_transform([5, NULL, 6], lambda x: coalesce(x, 0) + 1);
```

```
[6, 1, 7]
```

`list_filter` Examples

Filter out negative values:

```
SELECT list_filter([5, -6, NULL, 7], lambda x: x > 0);
```

```
[5, 7]
```

Divisible by 2 and 5:

```
SELECT list_filter(
        list_filter([2, 4, 3, 1, 20, 10, 3, 30], lambda x: x % 2 = 0),
        lambda y: y % 5 = 0
    );
```

```
[20, 10, 30]
```

In combination with `range(...)` to construct lists:

```
SELECT list_filter([1, 2, 3, 4], lambda x: x > #1) FROM range(4);
```

```
[1, 2, 3, 4]
[2, 3, 4]
[3, 4]
[4]
```

`list_reduce` Examples

Sum of all list elements:

```
SELECT list_reduce([1, 2, 3, 4], lambda acc, x: acc + x);
```

```
10
```

Only add up list elements if they are greater than 2:

```
SELECT list_reduce(
        list_filter([1, 2, 3, 4], lambda x: x > 2),
        lambda acc, x: acc + x
    );
```

```
7
```

Concat all list elements:

```
SELECT list_reduce(['DuckDB', 'is', 'awesome'], lambda acc, x: concat(acc, ' ', x));
```

```
DuckDB is awesome
```

Concatenate elements with the index without an initial value:

```
SELECT list_reduce(
        ['a', 'b', 'c', 'd'],
        lambda x, y, i: x || ' - ' || CAST(i AS VARCHAR) || ' - ' || y
    );
```

```
a - 2 - b - 3 - c - 4 - d
```

Concatenate elements with the index with an initial value:

```
SELECT list_reduce(
        ['a', 'b', 'c', 'd'],
        lambda x, y, i: x || ' - ' || CAST(i AS VARCHAR) || ' - ' || y, 'INITIAL'
    );
```

```
INITIAL - 1 - a - 2 - b - 3 - c - 4 - d
```

Limitations

- Subqueries in lambda expressions are currently not supported. For example:

  ```
  SELECT list_apply([1, 2, 3], lambda x: (SELECT 42) + x);
  ```

  ```
  Binder Error:
  subqueries in lambda expressions are not supported
  ```

  2) The expression can contain macro calls, but only if the macro is a scalar macro. In addition, the macro must not contain a subquery or rely on another macro that contains a subquery. 3) lambda functions must be defined inline as argument to the list function. You cannot return a lambda function from a macro (but you can invoke a macro in the lambda's expression).

Deprecated Syntax

> **Deprecated.** DuckDB 1.3 deprecated the old lambda single arrow syntax (`x -> x + 1`) in favor of the Python-style syntax (`lambda x : x + 1`).
>
> DuckDB 1.3 also introduced a new setting to configure the lambda syntax.
>
> ```
> SET lambda_syntax = 'default';
> SET lambda_syntax = 'enable_single_arrow';
> SET lambda_syntax = 'disable_single_arrow';
> ```
>
> From DuckDB 1.3 to 1.5, `default` enabled both syntax styles, i.e., the old single arrow syntax and the Python-style syntax.
>
> DuckDB 1.5 is the last release supporting the single arrow syntax without explicitly enabling it.
>
> DuckDB 2.0 disables the single arrow syntax by default. Under `default`, only the Python-style syntax is enabled; use `SET lambda_syntax = 'enable_single_arrow'`{:.language-sql .highlight} to revert to the deprecated behavior.
>
> DuckDB 2.1 will remove the `lambda_syntax` flag and the single arrow syntax entirely, so the old behavior will no longer be possible.

List Functions

Function	Description
`list[index]`	Extracts a single list element using a (1-based) `index`.

Function	Description
`list[begin[:end][:step]]`	Extracts a sublist using slice conventions. Negative values are accepted.
`list1 && list2`	Alias for `list_has_any`.
`list1 <-> list2`	Alias for `list_distance`.
`list1 <=> list2`	Alias for `list_cosine_distance`.
`list1 <@ list2`	Alias for `list_has_all`.
`list1 @> list2`	Alias for `list_has_all`.
`arg1 || arg2`	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.
`aggregate(list, function_name, ...)`	Alias for `list_aggregate`.
`apply(list, lambda(x))`	Alias for `list_transform`.
`array_aggr(list, function_name, ...)`	Alias for `list_aggregate`.
`array_aggregate(list, function_name, ...)`	Alias for `list_aggregate`.
`array_append(list, element)`	Alias for `list_append`.
`array_apply(list, lambda(x))`	Alias for `list_transform`.
`array_cat(list_1, ..., list_n)`	Alias for `list_concat`.
`array_concat(list_1, ..., list_n)`	Alias for `list_concat`.
`array_contains(list, element)`	Alias for `list_contains`.
`array_distinct(list)`	Alias for `list_distinct`.
`array_extract(list, index)`	Extracts the `index`th (1-based) value from the `list`.
`array_filter(list, lambda(x))`	Alias for `list_filter`.
`array_grade_up(list[, col1][, col2])`	Alias for `list_grade_up`.
`array_has(list, element)`	Alias for `list_contains`.
`array_has_all(list1, list2)`	Alias for `list_has_all`.
`array_has_any(list1, list2)`	Alias for `list_has_any`.
`array_indexof(list, element)`	Alias for `list_position`.
`array_intersect(list1, list2)`	Alias for `list_intersect`.
`array_length(list)`	Alias for `length`.

Function	Description
`array_pop_back(list)`	Returns the `list` without the last element.
`array_pop_front(list)`	Returns the `list` without the first element.
`array_position(list, element)`	Alias for `list_position`.
`array_prepend(element, list)`	Alias for `list_prepend`.
`array_push_back(list, element)`	Alias for `list_append`.
`array_push_front(list, element)`	Prepends `element` to `list`.
`array_reduce(list, lambda(x,y)[, initial_value])`	Alias for `list_reduce`.
`array_resize(list, size[[, value]])`	Alias for `list_resize`.
`array_reverse(list)`	Alias for `list_reverse`.
`array_reverse_sort(list[, col1])`	Alias for `list_reverse_sort`.
`array_select(value_list, index_list)`	Alias for `list_select`.
`array_slice(list, begin, end)`	Alias for `list_slice`.
`array_slice(list, begin, end, step)`	Alias for `list_slice`.
`array_sort(list[, col1][, col2])`	Alias for `list_sort`.
`array_to_string(list, delimiter)`	Concatenates list/array elements using an optional `delimiter`.
`array_to_string_comma_default(array)`	Concatenates list/array elements with a comma delimiter.
`array_transform(list, lambda(x))`	Alias for `list_transform`.
`array_unique(list)`	Alias for `list_unique`.
`array_where(value_list, mask_list)`	Alias for `list_where`.
`array_zip(list_1, ..., list_n[, truncate])`	Alias for `list_zip`.
`char_length(list)`	Alias for `length`.
`character_length(list)`	Alias for `length`.
`concat(value, ...)`	Concatenates multiple strings or lists. `NULL` inputs are skipped. See also operator `||`.
`contains(list, element)`	Returns `true` if the `list` contains the `element`.
`filter(list, lambda(x))`	Alias for `list_filter`.
`flatten(nested_list)`	Flattens a nested list by one level.

Function	Description
`generate_series(start[, stop][, step])`	Creates a list of values between `start` and `stop` - the stop parameter is inclusive.
`grade_up(list[, col1][, col2])`	Alias for `list_grade_up`.
`len(list)`	Alias for `length`.
`length(list)`	Returns the length of the `list`.
`list_aggr(list, function_name, ...)`	Alias for `list_aggregate`.
`list_aggregate(list, function_name, ...)`	Executes the aggregate function `function_name` on the elements of `list`. See the List Aggregates section for more details.
`list_any_value(list)`	Applies aggregate function `any_value` to the `list`.
`list_append(list, element)`	Appends `element` to `list`.
`list_apply(list, lambda(x))`	Alias for `list_transform`.
`list_approx_count_distinct(list)`	Applies aggregate function `approx_count_distinct` to the `list`.
`list_avg(list)`	Applies aggregate function `avg` to the `list`.
`list_bit_and(list)`	Applies aggregate function `bit_and` to the `list`.
`list_bit_or(list)`	Applies aggregate function `bit_or` to the `list`.
`list_bit_xor(list)`	Applies aggregate function `bit_xor` to the `list`.
`list_bool_and(list)`	Applies aggregate function `bool_and` to the `list`.
`list_bool_or(list)`	Applies aggregate function `bool_or` to the `list`.
`list_cat(list_1, ..., list_n)`	Alias for `list_concat`.
`list_concat(list_1, ..., list_n)`	Concatenates lists. `NULL` inputs are skipped. See also operator `||`.
`list_contains(list, element)`	Returns true if the list contains the element.
`list_cosine_distance(list1, list2)`	Computes the cosine distance between two same-sized lists.
`list_cosine_similarity(list1, list2)`	Computes the cosine similarity between two same-sized lists.
`list_count(list)`	Applies aggregate function `count` to the `list`.
`list_distance(list1, list2)`	Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.
`list_distinct(list)`	Removes all duplicates and `NULL` values from a list. Does not preserve the original order.
`list_dot_product(list1, list2)`	Alias for `list_inner_product`.
`list_element(list, index)`	Alias for `list_extract`.
`list_entropy(list)`	Applies aggregate function `entropy` to the `list`.
`list_extract(list, index)`	Extract the `index`th (1-based) value from the list.

Function	Description
`list_filter(list, lambda(x))`	Constructs a list from those elements of the input `list` for which the `lambda` function returns `true`. DuckDB must be able to cast the `lambda` function's return type to `BOOL`. The return type of `list_filter` is the same as the input list's. See `list_filter` examples.
`list_first(list)`	Applies aggregate function `first` to the `list`.
`list_grade_up(list[, col1][, col2])`	Works like `list_sort`, but the results are the indexes that correspond to the position in the original list instead of the actual values.
`list_has(list, element)`	Alias for `list_contains`.
`list_has_all(list1, list2)`	Returns true if all elements of list2 are in list1. NULLs are ignored.
`list_has_any(list1, list2)`	Returns true if the lists have any element in common. NULLs are ignored.
`list_histogram(list)`	Applies aggregate function `histogram` to the `list`.
`list_indexof(list, element)`	Alias for `list_position`.
`list_inner_product(list1, list2)`	Computes the inner product between two same-sized lists.
`list_intersect(list1, list2)`	Returns a list of all the elements that exist in both `list1` and `list2`, without duplicates.
`list_kurtosis(list)`	Applies aggregate function `kurtosis` to the `list`.
`list_kurtosis_pop(list)`	Applies aggregate function `kurtosis_pop` to the `list`.
`list_last(list)`	Applies aggregate function `last` to the `list`.
`list_mad(list)`	Applies aggregate function `mad` to the `list`.
`list_max(list)`	Applies aggregate function `max` to the `list`.
`list_median(list)`	Applies aggregate function `median` to the `list`.
`list_min(list)`	Applies aggregate function `min` to the `list`.
`list_mode(list)`	Applies aggregate function `mode` to the `list`.
`list_negative_dot_product(list1, list2)`	Alias for `list_negative_inner_product`.
`list_negative_inner_product(list1, list2)`	Computes the negative inner product between two same-sized lists.
`list_pack(arg, ...)`	Alias for `list_value`.
`list_position(list, element)`	Returns the index of the `element` if the `list` contains the `element`. If the `element` is not found, it returns `NULL`.
`list_prepend(element, list)`	Prepends `element` to `list`.
`list_product(list)`	Applies aggregate function `product` to the `list`.
`list_reduce(list, lambda(x,y)[, initial_value])`	Reduces all elements of the input `list` into a single scalar value by executing the `lambda` function on a running result and the next list element. The `lambda` function has an optional `initial_value` argument. See `list_reduce` examples.
`list_resize(list, size[[, value]])`	Resizes the `list` to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.
`list_reverse(list)`	Reverses the `list`.

Function	Description
`list_reverse_sort(list[, col1])`	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about sorting order and NULL values.
`list_select(value_list, index_list)`	Returns a list based on the elements selected by the `index_list`.
`list_sem(list)`	Applies aggregate function `sem` to the `list`.
`list_skewness(list)`	Applies aggregate function `skewness` to the `list`.
`list_slice(list, begin, end)`	Extracts a sublist or substring using slice conventions. Negative values are accepted.
`list_slice(list, begin, end, step)`	list_slice with added step feature.
`list_sort(list[, col1][, col2])`	Sorts the elements of the list. See the Sorting Lists section for more details about sorting order and NULL values.
`list_stddev_pop(list)`	Applies aggregate function `stddev_pop` to the `list`.
`list_stddev_samp(list)`	Applies aggregate function `stddev_samp` to the `list`.
`list_string_agg(list)`	Applies aggregate function `string_agg` to the `list`.
`list_sum(list)`	Applies aggregate function `sum` to the `list`.
`list_transform(list, lambda(x))`	Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function. See `list_transform` examples.
`list_unique(list)`	Counts the unique elements of a `list`.
`list_value(arg, ...)`	Creates a LIST containing the argument values.
`list_var_pop(list)`	Applies aggregate function `var_pop` to the `list`.
`list_var_samp(list)`	Applies aggregate function `var_samp` to the `list`.
`list_where(value_list, mask_list)`	Returns a list with the BOOLEANs in `mask_list` applied as a mask to the `value_list`.
`list_zip(list_1, ..., list_n[, truncate])`	Zips n `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of n elements from each list `list_1`, …, `list_n`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.
`range(start[, stop][, step])`	Creates a list of values between `start` and `stop` - the stop parameter is exclusive.
`reduce(list, lambda(x,y)[, initial_value])`	Alias for `list_reduce`.
`repeat(list, count)`	Repeats the `list` `count` number of times.
`unnest(list)`	Unnests a list by one level. Note that this is a special function that alters the cardinality of the result. See the unnest page for more details.
`unpivot_list(arg, ...)`	Identical to list_value, but generated as part of unpivot for better error messages.

`list[index]`

Description	Extracts a single list element using a (1-based) `index`.
Example	`[4, 5, 6][3]`

Result	6
Alias	`list_extract`

`list[begin[:end][:step]]`

Description	Extracts a sublist using slice conventions. Negative values are accepted.
Example	`[4, 5, 6][3]`
Result	6
Alias	`list_slice`

`arg1 || arg2`

Description	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.
Example 1	`'Duck' || 'DB'`
Result	DuckDB
Example 2	`[1, 2, 3] || [4, 5, 6]`
Result	`[1, 2, 3, 4, 5, 6]`
Example 3	`'\xAA'::BLOB || '\xBB'::BLOB`
Result	\xAA\xBB

`array_extract(list, index)`

Description	Extracts the `index`th (1-based) value from the `list`.
Example	`array_extract([4, 5, 6], 3)`
Result	6

`array_pop_back(list)`

Description	Returns the `list` without the last element.
Example	`array_pop_back([4, 5, 6])`
Result	`[4, 5]`

`array_pop_front(list)`

Description	Returns the `list` without the first element.
Example	`array_pop_front([4, 5, 6])`
Result	`[5, 6]`

`array_push_front(list, element)`

Description	Prepends `element` to `list`.
Example	`array_push_front([4, 5, 6], 3)`
Result	`[3, 4, 5, 6]`

`array_to_string(list, delimiter)`

Description	Concatenates list/array elements using an optional `delimiter`.
Example 1	`array_to_string([1, 2, 3], '-')`
Result	`1-2-3`
Example 2	`array_to_string(['aa', 'bb', 'cc'], '')`
Result	`aabbcc`

`array_to_string_comma_default(array)`

Description	Concatenates list/array elements with a comma delimiter.
Example	`array_to_string_comma_default(['Banana', 'Apple', 'Melon'])`
Result	`Banana,Apple,Melon`

`concat(value, ...)`

Description	Concatenates multiple strings or lists. `NULL` inputs are skipped. See also operator `||`.
Example 1	`concat('Hello', ' ', 'World')`
Result	`Hello World`
Example 2	`concat([1, 2, 3], NULL, [4, 5, 6])`
Result	`[1, 2, 3, 4, 5, 6]`

`contains(list, element)`

Description	Returns `true` if the `list` contains the `element`.
Example	`contains([1, 2, NULL], 1)`
Result	`true`

`flatten(nested_list)`

Description	Flattens a nested list by one level.
Example	`flatten([[1, 2, 3], [4, 5]])`
Result	`[1, 2, 3, 4, 5]`

`generate_series(start[, stop][, step])`

Description	Creates a list of values between `start` and `stop` - the stop parameter is inclusive.
Example	`generate_series(2, 5, 3)`
Result	`[2, 5]`

`length(list)`

Description	Returns the length of the `list`.
Example	`length([1,2,3])`
Result	3
Aliases	`char_length`, `character_length`, `len`

`list_aggregate(list, function_name, ...)`

Description	Executes the aggregate function `function_name` on the elements of `list`. See the List Aggregates section for more details.
Example	`list_aggregate([1, 2, NULL], 'min')`
Result	1
Aliases	`aggregate`, `array_aggr`, `array_aggregate`, `list_aggr`

`list_any_value(list)`

Description	Applies aggregate function `any_value` to the `list`.
Example	`list_any_value([3,3,9])`
Result	3

`list_append(list, element)`

Description	Appends `element` to `list`.
Example	`list_append([2, 3], 4)`
Result	`[2, 3, 4]`
Aliases	`array_append`, `array_push_back`

`list_approx_count_distinct(list)`

Description	Applies aggregate function `approx_count_distinct` to the `list`.
Example	`list_approx_count_distinct([3,3,9])`
Result	2

`list_avg(list)`

Description	Applies aggregate function `avg` to the `list`.
Example	`list_avg([3,3,9])`
Result	`5.0`

`list_bit_and(list)`

Description	Applies aggregate function `bit_and` to the `list`.
Example	`list_bit_and([3,3,9])`
Result	`1`

`list_bit_or(list)`

Description	Applies aggregate function `bit_or` to the `list`.
Example	`list_bit_or([3,3,9])`
Result	`11`

`list_bit_xor(list)`

Description	Applies aggregate function `bit_xor` to the `list`.
Example	`list_bit_xor([3,3,9])`
Result	`9`

`list_bool_and(list)`

Description	Applies aggregate function `bool_and` to the `list`.
Example	`list_bool_and([true, false])`
Result	`false`

`list_bool_or(list)`

Description	Applies aggregate function `bool_or` to the `list`.
Example	`list_bool_or([true, false])`
Result	`true`

`list_concat(list_1, ..., list_n)`

Description	Concatenates lists. `NULL` inputs are skipped. See also operator `||`.
Example	`list_concat([2, 3], [4, 5, 6], [7])`

Result	`[2, 3, 4, 5, 6, 7]`
Aliases	`list_cat`, `array_concat`, `array_cat`

`list_contains(list, element)`

Description	Returns true if the list contains the element.
Example	`list_contains([1, 2, NULL], 1)`
Result	`true`
Aliases	`array_contains`, `array_has`, `list_has`

`list_cosine_distance(list1, list2)`

Description	Computes the cosine distance between two same-sized lists.
Example	`list_cosine_distance([1, 2, 3], [1, 2, 3])`
Result	`0.0`
Alias	`<=>`

`list_cosine_similarity(list1, list2)`

Description	Computes the cosine similarity between two same-sized lists.
Example	`list_cosine_similarity([1, 2, 3], [1, 2, 3])`
Result	`1.0`

`list_count(list)`

Description	Applies aggregate function `count` to the `list`.
Example	`list_count([3,3,9])`
Result	`3`

`list_distance(list1, list2)`

Description	Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.
Example	`list_distance([1, 2, 3], [1, 2, 5])`
Result	`2.0`
Alias	`<->`

`list_distinct(list)`

Description	Removes all duplicates and NULL values from a list. Does not preserve the original order.
Example	`list_distinct([1, 1, NULL, -3, 1, 5])`
Result	`[5, -3, 1]`
Alias	`array_distinct`

`list_entropy(list)`

Description	Applies aggregate function `entropy` to the `list`.
Example	`list_entropy([3,3,9])`
Result	`0.9182958340544893`

`list_extract(list, index)`

Description	Extract the `index`th (1-based) value from the list.
Example	`list_extract([4, 5, 6], 3)`
Result	`6`
Alias	`list_element`

`list_filter(list, lambda(x))`

Description	Constructs a list from those elements of the input `list` for which the `lambda` function returns `true`. DuckDB must be able to cast the `lambda` function's return type to `BOOL`. The return type of `list_filter` is the same as the input list's. See `list_filter` examples.
Example	`list_filter([3, 4, 5], lambda x : x > 4)`
Result	`[5]`
Aliases	`array_filter`, `filter`

`list_first(list)`

Description	Applies aggregate function `first` to the `list`.
Example	`list_first([3,3,9])`
Result	`3`

`list_grade_up(list[, col1][, col2])`

Description	Works like `list_sort`, but the results are the indexes that correspond to the position in the original list instead of the actual values.
Example	`list_grade_up([3, 6, 1, 2])`
Result	`[3, 4, 1, 2]`
Aliases	`array_grade_up`, `grade_up`

`list_has_all(list1, list2)`

Description	Returns true if all elements of list2 are in list1. NULLs are ignored.
Example	`list_has_all([1, 2, 3], [2, 3])`
Result	`true`
Aliases	`<@`, `@>`, `array_has_all`

`list_has_any(list1, list2)`

Description	Returns true if the lists have any element in common. NULLs are ignored.
Example	`list_has_any([1, 2, 3], [2, 3, 4])`
Result	`true`
Aliases	`&&`, `array_has_any`

`list_histogram(list)`

Description	Applies aggregate function `histogram` to the `list`.
Example	`list_histogram([3,3,9])`
Result	`{3=2, 9=1}`

`list_inner_product(list1, list2)`

Description	Computes the inner product between two same-sized lists.
Example	`list_inner_product([1, 2, 3], [1, 2, 3])`
Result	`14.0`
Alias	`list_dot_product`

`list_intersect(list1, list2)`

Description	Returns a list of all the elements that exist in both `list1` and `list2`, without duplicates.
Example	`list_intersect([1, 2, 3], [2, 3, 4])`
Result	`[3, 2]`
Alias	`array_intersect`

`list_kurtosis(list)`

Description	Applies aggregate function `kurtosis` to the `list`.
Example	`list_kurtosis([3,3,9])`

Result	`NULL`

`list_kurtosis_pop(list)`

Description	Applies aggregate function `kurtosis_pop` to the `list`.
Example	`list_kurtosis_pop([3,3,9])`
Result	`-1.4999999999999978`

`list_last(list)`

Description	Applies aggregate function `last` to the `list`.
Example	`list_last([3,3,9])`
Result	`9`

`list_mad(list)`

Description	Applies aggregate function `mad` to the `list`.
Example	`list_mad([3,3,9])`
Result	`0.0`

`list_max(list)`

Description	Applies aggregate function `max` to the `list`.
Example	`list_max([3,3,9])`
Result	`9`

`list_median(list)`

Description	Applies aggregate function `median` to the `list`.
Example	`list_median([3,3,9])`
Result	`3.0`

`list_min(list)`

Description	Applies aggregate function `min` to the `list`.
Example	`list_min([3,3,9])`
Result	`3`

`list_mode(list)`

Description	Applies aggregate function `mode` to the `list`.
Example	`list_mode([3,3,9])`
Result	`3`

`list_negative_inner_product(list1, list2)`

Description	Computes the negative inner product between two same-sized lists.
Example	`list_negative_inner_product([1, 2, 3], [1, 2, 3])`
Result	`-14.0`
Alias	`list_negative_dot_product`

`list_position(list, element)`

Description	Returns the index of the `element` if the `list` contains the `element`. If the `element` is not found, it returns `NULL`.
Example	`list_position([1, 2, NULL], 2)`
Result	`2`
Aliases	`array_indexof`, `array_position`, `list_indexof`

`list_prepend(element, list)`

Description	Prepends `element` to `list`.
Example	`list_prepend(3, [4, 5, 6])`
Result	`[3, 4, 5, 6]`
Alias	`array_prepend`

`list_product(list)`

Description	Applies aggregate function `product` to the `list`.
Example	`list_product([3,3,9])`
Result	`81.0`

`list_reduce(list, lambda(x,y)[, initial_value])`

Description	Reduces all elements of the input `list` into a single scalar value by executing the `lambda` function on a running result and the next list element. The `lambda` function has an optional `initial_value` argument. See `list_reduce` examples.
Example	`list_reduce([1, 2, 3], lambda x, y : x + y)`
Result	`6`

Aliases	`array_reduce`, `reduce`

`list_resize(list, size[[, value]])`

Description	Resizes the `list` to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.
Example	`list_resize([1, 2, 3], 5, 0)`
Result	`[1, 2, 3, 0, 0]`
Alias	`array_resize`

`list_reverse(list)`

Description	Reverses the `list`.
Example	`list_reverse([3, 6, 1, 2])`
Result	`[2, 1, 6, 3]`
Alias	`array_reverse`

`list_reverse_sort(list[, col1])`

Description	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about sorting order and `NULL` values.
Example	`list_reverse_sort([3, 6, 1, 2])`
Result	`[6, 3, 2, 1]`
Alias	`array_reverse_sort`

`list_select(value_list, index_list)`

Description	Returns a list based on the elements selected by the `index_list`.
Example	`list_select([10, 20, 30, 40], [1, 4])`
Result	`[10, 40]`
Alias	`array_select`

`list_sem(list)`

Description	Applies aggregate function `sem` to the `list`.
Example	`list_sem([3,3,9])`
Result	`1.6329931618554523`

`list_skewness(list)`

Description	Applies aggregate function `skewness` to the `list`.
Example	`list_skewness([3,3,9])`
Result	`1.7320508075688796`

`list_slice(list, begin, end)`

Description	Extracts a sublist or substring using slice conventions. Negative values are accepted.
Example	`list_slice([4, 5, 6], 2, 3)`
Result	`[5, 6]`
Alias	`array_slice`

`list_slice(list, begin, end, step)`

Description	list_slice with added step feature.
Example	`list_slice([4, 5, 6], 1, 3, 2)`
Result	`[4, 6]`
Alias	`array_slice`

`list_sort(list[, col1][, col2])`

Description	Sorts the elements of the list. See the Sorting Lists section for more details about sorting order and `NULL` values.
Example	`list_sort([3, 6, 1, 2])`
Result	`[1, 2, 3, 6]`
Alias	`array_sort`

`list_stddev_pop(list)`

Description	Applies aggregate function `stddev_pop` to the `list`.
Example	`list_stddev_pop([3,3,9])`
Result	`2.8284271247461903`

`list_stddev_samp(list)`

Description	Applies aggregate function `stddev_samp` to the `list`.
Example	`list_stddev_samp([3,3,9])`
Result	`3.4641016151377544`

`list_string_agg(list)`

Description	Applies aggregate function `string_agg` to the `list`.
Example	`list_string_agg([3,3,9])`
Result	`3,3,9`

`list_sum(list)`

Description	Applies aggregate function `sum` to the `list`.
Example	`list_sum([3,3,9])`
Result	`15`

`list_transform(list, lambda(x))`

Description	Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function. See `list_transform` examples.
Example	`list_transform([1, 2, 3], lambda x : x + 1)`
Result	`[2, 3, 4]`
Aliases	`apply`, `array_apply`, `array_transform`, `list_apply`

`list_unique(list)`

Description	Counts the unique elements of a `list`.
Example	`list_unique([1, 1, NULL, -3, 1, 5])`
Result	`3`
Alias	`array_unique`

`list_value(arg, ...)`

Description	Creates a LIST containing the argument values.
Example	`list_value(4, 5, 6)`
Result	`[4, 5, 6]`
Alias	`list_pack`

`list_var_pop(list)`

Description	Applies aggregate function `var_pop` to the `list`.
Example	`list_var_pop([3,3,9])`
Result	`8.0`

`list_var_samp(list)`

Description	Applies aggregate function `var_samp` to the `list`.
Example	`list_var_samp([3,3,9])`
Result	`12.0`

`list_where(value_list, mask_list)`

Description	Returns a list with the BOOLEANs in `mask_list` applied as a mask to the `value_list`.
Example	`list_where([10, 20, 30, 40], [true, false, false, true])`
Result	`[10, 40]`
Alias	`array_where`

`list_zip(list_1, ..., list_n[, truncate])`

Description	Zips n `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of n elements from each list `list_1`, …, `list_n`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.
Example 1	`list_zip([1, 2], [3, 4], [5, 6])`
Result	`[(1, 3, 5), (2, 4, 6)]`
Example 2	`list_zip([1, 2], [3, 4], [5, 6, 7])`
Result	`[(1, 3, 5), (2, 4, 6), (NULL, NULL, 7)]`
Example 3	`list_zip([1, 2], [3, 4], [5, 6, 7], true)`
Result	`[(1, 3, 5), (2, 4, 6)]`
Alias	`array_zip`

`range(start[, stop][, step])`

Description	Creates a list of values between `start` and `stop` - the stop parameter is exclusive.
Example	`range(2, 5, 3)`
Result	`[2]`

`repeat(list, count)`

Description	Repeats the `list` `count` number of times.
Example	`repeat([1, 2, 3], 5)`
Result	`[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]`

`unnest(list)`

Description	Unnests a list by one level. Note that this is a special function that alters the cardinality of the result. See the unnest page for more details.
Example	`unnest([1, 2, 3])`
Result	Multiple rows: `'1','2','3'`

`unpivot_list(arg, ...)`

Description	Identical to list_value, but generated as part of unpivot for better error messages.
Example	`unpivot_list(4, 5, 6)`
Result	`[4, 5, 6]`

List Operators

The following operators are supported for lists:

Operator	Description	Example	Result
`&&`	Alias for `list_has_any`.	`[1, 2, 3, 4, 5] && [2, 5, 5, 6]`	`true`
`@>`	Alias for `list_has_all`, where the list on the **right** of the operator is the sublist.	`[1, 2, 3, 4] @> [3, 4, 3]`	`true`
`<@`	Alias for `list_has_all`, where the list on the **left** of the operator is the sublist.	`[1, 4] <@ [1, 2, 3, 4]`	`true`
`||`	Similar to `list_concat`, except any `NULL` input results in `NULL`.	`[1, 2, 3] || [4, 5, 6]`	`[1, 2, 3, 4, 5, 6]`
`<=>`	Alias for `list_cosine_distance`.	`[1, 2, 3] <=> [1, 2, 5]`	`0.024099927051466796`
`<->`	Alias for `list_distance`.	`[1, 2, 3] <-> [1, 2, 5]`	`2.0`

List Comprehension

Python-style list comprehension can be used to compute expressions over elements in a list. For example:

```
SELECT [lower(x) FOR x IN strings] AS strings
FROM (VALUES (['Hello', '', 'World'])) t(strings);
```

strings
[hello, , world]

```
SELECT [upper(x) FOR x IN strings IF len(x) > 0] AS strings
FROM (VALUES (['Hello', '', 'World'])) t(strings);
```

strings
[HELLO, WORLD]

List comprehensions can also use the position of the list elements by adding a second variable. In the following example, we use `x, i`, where `x` is the value and `i` is the position:

```
SELECT [4, 5, 6] AS l, [x FOR x, i IN l IF i != 2] AS filtered;
```

l	filtered
[4, 5, 6]	[4, 6]

Under the hood, `[f(x) FOR x IN l IF g(x)]` is translated to:

```
l.list_apply(lambda x, i: {'filter': g(x, i), 'result': f(x, i)})
    .list_filter(lambda x: x.filter)
    .list_apply(lambda x: x.result)
```

Range Functions

DuckDB offers two range functions, `range(start, stop, step)` and `generate_series(start, stop, step)`, and their variants with default arguments for `stop` and `step`. The two functions' behavior is different regarding their `stop` argument. This is documented below.

`range`

The `range` function creates a list of values in the range between `start` and `stop`. The `start` parameter is inclusive, while the `stop` parameter is exclusive. The default value of `start` is 0 and the default value of `step` is 1.

Based on the number of arguments, the following variants of `range` exist.

`range(stop)`

```
SELECT range(5);
```

```
[0, 1, 2, 3, 4]
```

`range(start, stop)`

```
SELECT range(2, 5);
```

```
[2, 3, 4]
```

`range(start, stop, step)`

```
SELECT range(2, 5, 3);
```

```
[2]
```

`generate_series`

The `generate_series` function creates a list of values in the range between `start` and `stop`. Both the `start` and the `stop` parameters are inclusive. The default value of `start` is 0 and the default value of `step` is 1. Based on the number of arguments, the following variants of `generate_series` exist.

generate_series(stop)

```
SELECT generate_series(5);
```

```
[0, 1, 2, 3, 4, 5]
```

generate_series(start, stop)

```
SELECT generate_series(2, 5);
```

```
[2, 3, 4, 5]
```

generate_series(start, stop, step)

```
SELECT generate_series(2, 5, 3);
```

```
[2, 5]
```

generate_subscripts(arr, dim) The `generate_subscripts(arr, dim)` function generates indexes along the `dim`th dimension of array `arr`.

```
SELECT generate_subscripts([4, 5, 6], 1) AS i;
```

i
1
2
3

Date Ranges

Date ranges are also supported for `TIMESTAMP` and `TIMESTAMP WITH TIME ZONE` values. Note that for these types, the `stop` and `step` arguments have to be specified explicitly (a default value is not provided).

range for Date Ranges

```
SELECT *
FROM range(DATE '1992-01-01', DATE '1992-03-01', INTERVAL '1' MONTH);
```

range
1992-01-01 00:00:00
1992-02-01 00:00:00

generate_series for Date Ranges

```
SELECT *
FROM generate_series(DATE '1992-01-01', DATE '1992-03-01', INTERVAL '1' MONTH);
```

generate_series
1992-01-01 00:00:00
1992-02-01 00:00:00

generate_series
1992-03-01 00:00:00

Slicing

The function `list_slice` can be used to extract a sublist from a list. The following variants exist:

- `list_slice(list, begin, end)`
- `list_slice(list, begin, end, step)`
- `array_slice(list, begin, end)`
- `array_slice(list, begin, end, step)`
- `list[begin:end]`
- `list[begin:end:step]`

The arguments are as follows:

- `list`
  - Is the list to be sliced
- `begin`
  - Is the index of the first element to be included in the slice
  - When `begin < 0` the index is counted from the end of the list
  - When `begin < 0` and `-begin > length`, `begin` is clamped to the beginning of the list
  - When `begin > length`, the result is an empty list
  - **Bracket Notation:** When `begin` is omitted, it defaults to the beginning of the list
- `end`
  - Is the index of the last element to be included in the slice
  - When `end < 0` the index is counted from the end of the list
  - When `end > length`, `end` is clamped to `length`
  - When `end < begin`, the result is an empty list
  - **Bracket Notation:** When `end` is omitted, it defaults to the end of the list. When `end` is omitted and a `step` is provided, `end` must be replaced with a `-`
- `step` *(optional)*
  - Is the step size between elements in the slice
  - When `step < 0` the slice is reversed, and `begin` and `end` are swapped
  - Must be non-zero

Examples:

```
SELECT list_slice([1, 2, 3, 4, 5], 2, 4);
```

```
[2, 3, 4]
```

```
SELECT ([1, 2, 3, 4, 5])[2:4:2];
```

```
[2, 4]
```

```
SELECT([1, 2, 3, 4, 5])[4:2:-2];
```

```
[4, 2]
```

```
SELECT ([1, 2, 3, 4, 5])[:];
```

```
[1, 2, 3, 4, 5]
```

```
SELECT ([1, 2, 3, 4, 5])[:-:2];
```

```
[1, 3, 5]
```

```
SELECT ([1, 2, 3, 4, 5])[:-:-2];
```

```
[5, 3, 1]
```

List Aggregates

The function `list_aggregate` allows the execution of arbitrary existing aggregate functions on the elements of a list. Its first argument is the list (column), its second argument is the aggregate function name, e.g., `min`, `histogram` or `sum`.

`list_aggregate` accepts additional arguments after the aggregate function name. These extra arguments are passed directly to the aggregate function, which serves as the second argument of `list_aggregate`.

Order-sensitive aggregate functions are applied in the order of the list. The `ORDER BY`, `DISTINCT` and `FILTER` clauses are not supported by `list_aggregate`. They may instead be emulated using `list_sort`, `list_grade_up`, `list_select`, `list_distinct` and `list_filter`.

```
SELECT list_aggregate([1, 2, -4, NULL], 'min');
```

```
-4
```

```
SELECT list_aggregate([2, 4, 8, 42], 'sum');
```

```
56
```

```
SELECT list_aggregate([[1, 2], [NULL], [2, 10, 3]], 'last');
```

```
[2, 10, 3]
```

```
SELECT list_aggregate([2, 4, 8, 42], 'string_agg', '|');
```

```
2|4|8|42
```

`list_*` Rewrite Functions

The following is a list of existing rewrites. Rewrites simplify the use of the list aggregate function by only taking the list (column) as their argument. `list_avg`, `list_var_samp`, `list_var_pop`, `list_stddev_pop`, `list_stddev_samp`, `list_sem`, `list_approx_count_distinct`, `list_bit_xor`, `list_bit_or`, `list_bit_and`, `list_bool_and`, `list_bool_or`, `list_count`, `list_entropy`, `list_last`, `list_first`, `list_kurtosis`, `list_kurtosis_pop`, `list_min`, `list_max`, `list_product`, `list_skewness`, `list_sum`, `list_string_agg`, `list_mode`, `list_median`, `list_mad` and `list_histogram`.

```
SELECT list_min([1, 2, -4, NULL]);
```

```
-4
```

```
SELECT list_sum([2, 4, 8, 42]);
```

```
56
```

```
SELECT list_last([[1, 2], [NULL], [2, 10, 3]]);
```

```
[2, 10, 3]
```

`array_to_string` Concatenates list/array elements using an optional delimiter.

```
SELECT array_to_string([1, 2, 3], '-') AS str;
```

```
1-2-3
```

This is equivalent to the following SQL:

```
SELECT list_aggr([1, 2, 3], 'string_agg', '-') AS str;
```

```
1-2-3
```

Sorting Lists

The function `list_sort` sorts the elements of a list either in ascending or descending order. In addition, it allows specifying whether NULL values should be moved to the beginning or to the end of the list. It has the same sorting behavior as DuckDB's `ORDER BY` clause. Therefore, (nested) values compare the same in `list_sort` as in `ORDER BY`.

By default, if no modifiers are provided, DuckDB sorts `ASC NULLS LAST`. I.e., the values are sorted in ascending order and NULL values are placed last. For ascending order, this matches PostgreSQL's default. The default sort order can be changed using PRAGMA statements..

`list_sort` leaves it open to the user whether they want to use the default sort order or a custom order. `list_sort` takes up to two additional optional parameters. The second parameter provides the sort order and can be either ASC or DESC. The third parameter provides the NULL order and can be either `NULLS FIRST` or `NULLS LAST`.

This query uses the default sort order and the default NULL order.

```
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
```

```
[-5, 1, 3, 5, NULL, NULL]
```

This query provides the sort order. The NULL order uses the configurable default value.

```
SELECT list_sort([1, 3, NULL, 2], 'ASC');
```

```
[1, 2, 3, NULL]
```

This query provides both the sort order and the NULL order.

```
SELECT list_sort([1, 3, NULL, 2], 'DESC', 'NULLS FIRST');
```

```
[NULL, 3, 2, 1]
```

`list_reverse_sort` has an optional second parameter providing the NULL sort order. It can be either `NULLS FIRST` or `NULLS LAST`.

This query uses the default NULL sort order.

```
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
```

```
[-5, 1, 3, 5, NULL, NULL]
```

This query provides the NULL sort order.

```
SELECT list_reverse_sort([1, 3, NULL, 2], 'NULLS LAST');
```

```
[3, 2, 1, NULL]
```

Flattening

The flatten function is a scalar function that converts a list of lists into a single list by concatenating each sub-list together. Note that this only flattens one level at a time, not all levels of sub-lists.

Convert a list of lists into a single list:

```
SELECT
    flatten([
        [1, 2],
        [3, 4]
    ]);
```

```
[1, 2, 3, 4]
```

If the list has multiple levels of lists, only the first level of sub-lists is concatenated into a single list:

```
SELECT
    flatten([
        [
            [1, 2],
            [3, 4],
        ],
        [
            [5, 6],
            [7, 8],
        ]
    ]);
```

```
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

In general, the input to the flatten function should be a list of lists (not a single level list). However, the flatten function has specific behavior when handling empty lists and `NULL` values.

If the input list is empty, return an empty list:

```
SELECT flatten([]);
```

```
[]
```

If the entire input to flatten is `NULL`, return `NULL`:

```
SELECT flatten(NULL);
```

```
NULL
```

If a list whose only entry is `NULL` is flattened, return an empty list:

```
SELECT flatten([NULL]);
```

```
[]
```

If the sub-list in a list of lists only contains `NULL`, do not modify the sub-list:

```
-- (Note the extra set of parentheses vs. the prior example)
SELECT flatten([[NULL]]);
```

```
[NULL]
```

Even if the only contents of each sub-list is `NULL`, still concatenate them together. Note that no de-duplication occurs when flattening. See `list_distinct` function for de-duplication:

```
SELECT flatten([[NULL], [NULL]]);
```

```
[NULL, NULL]
```

Lambda Functions

DuckDB supports lambda functions in the form `lambda parameter1, parameter2, ...: expression`. For details, see the lambda functions page.

Related Functions

- The aggregate functions `list` and `histogram` produce lists and lists of structs.
- The `unnest` function is used to unnest a list by one level.

Map Functions

Name	Description
`cardinality(map)`	Return the size of the map (or the number of entries in the map).
`element_at(map, key)`	Return the value for a given `key` as a list, or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
`map_concat(maps...)`	Returns a map created from merging the input `maps`. On key collision the value is taken from the last map with that key.
`map_contains(map, key)`	Checks if a map contains a given key.
`map_contains_entry(map, key, value)`	Check if a map contains a given key-value pair.
`map_contains_value(map, value)`	Checks if a map contains a given value.
`map_entries(map)`	Return a list of struct(k, v) for each key-value pair in the map.
`map_extract(map, key)`	Return the value for a given `key` as a list, or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
`map_extract_value(map, key)`	Returns the value for a given `key` or `NULL` if the `key` is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
`map_from_entries(STRUCT(k, v)[])`	Returns a map created from the entries of the array.
`map_keys(map)`	Return a list of all keys in the map.
`map_values(map)`	Return a list of all values in the map.
`map()`	Returns an empty map.
`map[entry]`	Returns the value for a given `key` or `NULL` if the `key` is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
`switch(key, map)`	Returns the value for a given `key` in the `map`, or `NULL` if the `key` is not present. An optional third argument, `switch(key, map, default)`, returns `default` instead of `NULL` when the `key` is not present.

`cardinality(map)`

Description	Return the size of the map (or the number of entries in the map).
Example	`cardinality(map([4, 2], ['a', 'b']))`
Result	2

`element_at(map, key)`

Description	Return the value for a given `key` as a list, or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
Example	`element_at(map([100, 5], [42, 43]), 100)`

Result	`[42]`
Aliases	`map_extract(map, key)`

`map_concat(maps...)`

Description	Returns a map created from merging the input `maps`. On key collision the value is taken from the last map with that key.
Example	`map_concat(MAP {'key1': 10, 'key2': 20}, MAP {'key3': 30}, MAP {'key2': 5})`
Result	`{key1=10, key2=5, key3=30}`

`map_contains(map, key)`

Description	Checks if a map contains a given key.
Example	`map_contains(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 'key2')`
Result	`true`

`map_contains_entry(map, key, value)`

Description	Check if a map contains a given key-value pair.
Example	`map_contains_entry(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 'key2', 20)`
Result	`true`

`map_contains_value(map, value)`

Description	Checks if a map contains a given value.
Example	`map_contains_value(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 20)`
Result	`true`

`map_entries(map)`

Description	Return a list of struct(k, v) for each key-value pair in the map.
Example	`map_entries(map([100, 5], [42, 43]))`
Result	`[{'key': 100, 'value': 42}, {'key': 5, 'value': 43}]`

`map_extract(map, key)`

Description	Return the value for a given `key` as a list, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
Example	`map_extract(map([100, 5], [42, 43]), 100)`
Result	`[42]`
Aliases	`element_at(map, key)`

`map_extract_value(map, key)`

Description	Returns the value for a given `key` or `NULL` if the `key` is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
Example	`map_extract_value(map([100, 5], [42, 43]), 100);`
Result	`42`
Aliases	`map[key]`

`map_from_entries(STRUCT(k, v)[])`

Description	Returns a map created from the entries of the array.
Example	`map_from_entries([{k: 5, v: 'val1'}, {k: 3, v: 'val2'}])`
Result	`{5=val1, 3=val2}`

`map_keys(map)`

Description	Return a list of all keys in the map.
Example	`map_keys(map([100, 5], [42,43]))`
Result	`[100, 5]`

`map_values(map)`

Description	Return a list of all values in the map.
Example	`map_values(map([100, 5], [42, 43]))`
Result	`[42, 43]`

`map()`

Description	Returns an empty map.
Example	`map()`
Result	`{}`

`map[entry]`

Description	Returns the value for a given `key` or `NULL` if the `key` is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys; else, an error is thrown.
Example	`map([100, 5], ['a', 'b'])[100]`
Result	`a`
Aliases	`map_extract_value(map, key)`

`switch(key, map)`

Description	Returns the value for a given `key` in the `map`, or `NULL` if the `key` is not present. This is equivalent to `map_extract_value(map, key)`. An optional third argument, `switch(key, map, default)`, returns `default` instead of `NULL` when the `key` is not present.
Example	`switch('key1', MAP {'key1': 10, 'key2': 20})`
Result	`10`

Nested Functions

The following nested data types have dedicated function pages (`TUPLE` values are handled by the `STRUCT` functions):

Name	Type page	Functions page
`ARRAY`	`ARRAY` type	`ARRAY` functions
`LIST`	`LIST` type	`LIST` functions
`MAP`	`MAP` type	`MAP` functions
`STRUCT`	`STRUCT` type	`STRUCT` functions
`UNION`	`UNION` type	`UNION` functions
`VARIANT`	`VARIANT` type	`VARIANT` functions

Numeric Functions

Numeric Operators

The table below shows the available mathematical operators for numeric types.

Operator	Description	Example	Result
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`2 - 3`	`-1`
`*`	Multiplication	`2 * 3`	`6`
`/`	Float division	`5 / 2`	`2.5`
`//`	Division	`5 // 2`	`2`
`%`	Modulo (remainder)	`5 % 4`	`1`

Operator	Description	Example	Result
`**`	Exponent	`3 ** 4`	81
`^`	Exponent (alias for `**`)	`3 ^ 4`	81
&	Bitwise AND	`91 & 15`	11
|	Bitwise OR	`32 | 3`	35
<<	Bitwise shift left	`1 << 4`	16
>>	Bitwise shift right	`8 >> 2`	2
~	Bitwise negation	`~15`	-16
!	Factorial of x	`4!`	24

Division and Modulo Operators

There are two division operators: `/` and `//`. They are equivalent when at least one of the operands is a `FLOAT` or a `DOUBLE`. When both operands are integers, `/` performs floating points division (`5 / 2 = 2.5`) while `//` performs integer division (`5 // 2 = 2`).

Integer division (`//`), modulo (`%`), and `INTERVAL / 0` throw an `Invalid Input Error` on division by zero instead of returning `NULL`. Wrap the expression in `TRY(...)` to return `NULL` for that expression, or `SET error_on_division_by_zero = false`{:.language-sql .highlight} to restore the old `NULL` result for all divisions by zero.

Supported Types

The bitwise, bitwise negation, and factorial operators work only on integral data types, whereas the others are available for all numeric data types.

Numeric Functions

The table below shows the available mathematical functions.

Name	Description
`@(x)`	Absolute value. Parentheses are optional if x is a column name.
`abs(x)`	Absolute value.
`acos(x)`	Computes the inverse cosine of x.
`acosh(x)`	Computes the inverse hyperbolic cosine of x.
`add(x, y)`	Alias for `x + y`.
`asin(x)`	Computes the inverse sine of x.
`asinh(x)`	Computes the inverse hyperbolic sine of x.
`atan(x)`	Computes the inverse tangent of x.
`atanh(x)`	Computes the inverse hyperbolic tangent of x.
`atan2(y, x)`	Computes the inverse tangent of `(y, x)`.
`binom(n, k)`	Computes the binomial coefficient of n and k, i.e., the number of ways to choose k elements from a set of n elements. Returns a `HUGEINT`.
`bit_count(x)`	Returns the number of bits that are set.
`cbrt(x)`	Returns the cube root of the number.

Name	Description
`ceil(x)`	Rounds the number up.
`ceiling(x)`	Rounds the number up. Alias of `ceil`.
`cos(x)`	Computes the cosine of `x`.
`cosh(x)`	Computes the hyperbolic cosine of `x`.
`cot(x)`	Computes the cotangent of `x`.
`decimal_division(x, y)`	Divides `x` by `y` using exact decimal arithmetic, returning a `DECIMAL`.
`degrees(x)`	Converts radians to degrees.
`divide(x, y)`	Alias for `x // y`.
`even(x)`	Round to next even number by rounding away from zero.
`exp(x)`	Computes `e ** x`.
`factorial(x)`	See the `!` operator. Computes the product of the current integer and all integers below it.
`fdiv(x, y)`	Performs integer division (`x // y`) using the floored definition and returns a `DOUBLE` value.
`floor(x)`	Rounds the number down.
`fmod(x, y)`	Calculates the modulo value using the floored definition. Always returns a `DOUBLE` value.
`gamma(x)`	Interpolation of the factorial of `x - 1`. Fractional inputs are allowed.
`gcd(x, y)`	Computes the greatest common divisor of `x` and `y`.
`greatest_common_divisor(x, y)`	Computes the greatest common divisor of `x` and `y`.
`greatest(x1, x2, ...)`	Selects the largest value.
`isfinite(x)`	Returns true if the floating point value is finite, false otherwise.
`isinf(x)`	Returns true if the floating point value is infinite, false otherwise.
`isnan(x)`	Returns true if the floating point value is not a number, false otherwise.
`lcm(x, y)`	Computes the least common multiple of `x` and `y`.
`least_common_multiple(x, y)`	Computes the least common multiple of `x` and `y`.
`least(x1, x2, ...)`	Selects the smallest value.
`lgamma(x)`	Computes the log of the `gamma` function.
`ln(x)`	Computes the natural logarithm of `x`.
`log(x)`	Computes the base-10 logarithm of `x`.
`log10(x)`	Alias of `log`. Computes the base-10 logarithm of `x`.
`log2(x)`	Computes the base-2 log of `x`.
`multiply(x, y)`	Alias for `x * y`.
`nextafter(x, y)`	Return the next floating point value after `x` in the direction of `y`.
`pi()`	Returns the value of pi.
`pow(x, y)`	Computes `x` to the power of `y`.
`power(x, y)`	Alias of pow. Computes `x` to the power of `y`.
`radians(x)`	Converts degrees to radians.
`random()`	Returns a random number `x` in the range `0.0 <= x < 1.0`.

Name	Description
`round_even(v NUMERIC, s INTEGER)`	Alias of `roundbankers(v, s)`. Round to `s` decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
`roundbankers(v NUMERIC, s INTEGER)`	Alias of `round_even(v, s)`. Round to `s` decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
`round(v NUMERIC, s INTEGER)`	Round to `s` decimal places. Values `s < 0` are allowed.
`setseed(x)`	Sets the seed to be used for the random function.
`sign(x)`	Returns the sign of `x` as -1, 0 or 1.
`signbit(x)`	Returns whether the signbit is set or not.
`sin(x)`	Computes the sin of `x`.
`sinh(x)`	Computes the hyperbolic sine of `x`.
`sqrt(x)`	Returns the square root of the number.
`subtract(x, y)`	Alias for `x - y`.
`tan(x)`	Computes the tangent of `x`.
`tanh(x)`	Computes the hyperbolic tangent of `x`.
`trunc(x)`	Truncates the number.
`xor(x, y)`	Bitwise XOR.

`@(x)`

Description	Absolute value. Parentheses are optional if `x` is a column name.
Example	`@(-17.4)`
Result	`17.4`
Alias	`abs`

`abs(x)`

Description	Absolute value.
Example	`abs(-17.4)`
Result	`17.4`
Alias	`@`

`acos(x)`

Description	Computes the inverse cosine of `x`.
Example	`acos(0.5)`
Result	`1.0471975511965976`

acosh(x)

Description	Computes the inverse hyperbolic cosine of x.
Example	`acosh(1.5)`
Result	`0.9624236501192069`

add(x, y)

Description	Alias for `x + y`.
Example	`add(2, 3)`
Result	`5`

asin(x)

Description	Computes the inverse sine of x.
Example	`asin(0.5)`
Result	`0.5235987755982989`

asinh(x)

Description	Computes the inverse hyperbolic sine of x.
Example	`asinh(0.5)`
Result	`0.48121182505960347`

atan(x)

Description	Computes the inverse tangent of x.
Example	`atan(0.5)`
Result	`0.4636476090008061`

atanh(x)

Description	Computes the inverse hyperbolic tangent of x.
Example	`atanh(0.5)`
Result	`0.5493061443340549`

atan2(y, x)

Description	Computes the inverse tangent (y, x).
Example	`atan2(0.5, 0.5)`

Result	`0.7853981633974483`

`binom(n, k)`

Description	Computes the binomial coefficient of n and k, i.e., the number of ways to choose k elements from a set of n elements. Returns a `HUGEINT`.
Example	`binom(5, 2)`
Result	`10`

`bit_count(x)`

Description	Returns the number of bits that are set.
Example	`bit_count(31)`
Result	`5`

`cbrt(x)`

Description	Returns the cube root of the number.
Example	`cbrt(8)`
Result	`2`

`ceil(x)`

Description	Rounds the number up.
Example	`ceil(17.4)`
Result	`18`

`ceiling(x)`

Description	Rounds the number up. Alias of `ceil`.
Example	`ceiling(17.4)`
Result	`18`

`cos(x)`

Description	Computes the cosine of `x`.
Example	`cos(pi() / 3)`
Result	`0.5000000000000001`

cosh(x)

Description	Computes the hyperbolic cosine of x.
Example	`cosh(1)`
Result	`1.5430806348152437`

cot(x)

Description	Computes the cotangent of x.
Example	`cot(0.5)`
Result	`1.830487721712452`

decimal_division(x, y)

Description	Divides the `DECIMAL` value x by y using exact decimal arithmetic with round-half-to-even, returning a `DECIMAL` rather than a `DOUBLE`. An optional third argument sets the scale of the result.
Example	`decimal_division(10.00, 4.00)`
Result	`2.500000`

> New The `decimal_division` function was added in DuckDB 2.0.

degrees(x)

Description	Converts radians to degrees.
Example	`degrees(pi())`
Result	`180`

divide(x, y)

Description	Alias for `x // y`.
Example	`divide(5, 2)`
Result	`2`

even(x)

Description	Round to next even number by rounding away from zero.
Example	`even(2.9)`
Result	`4`

`exp(x)`

Description	Computes `e ** x`.
Example	`exp(0.693)`
Result	`2`

`factorial(x)`

Description	See the `!` operator. Computes the product of the current integer and all integers below it.
Example	`factorial(4)`
Result	`24`

> **Note.** In DuckDB 2.0, the `!` operator binds more tightly than in earlier versions. For example, `SELECT 2 * 3!;` now returns `12` (`2 * (3!)`), while in v1.5 it returned `720` (`(2 * 3)!`). Use parentheses to get the old behavior, e.g., `SELECT (2 * 3)!;`.

`fdiv(x, y)`

Description	Performs integer division (`x // y`) using the floored definition and returns a `DOUBLE` value.
Example	`fdiv(5, 2)`
Result	`2.0`

`floor(x)`

Description	Rounds the number down.
Example	`floor(17.4)`
Result	`17`

`fmod(x, y)`

Description	Calculates the modulo value using the floored definition. Always returns a `DOUBLE` value.
Example	`fmod(5, 2)`
Result	`1.0`

`gamma(x)`

Description	Interpolation of the factorial of `x - 1`. Fractional inputs are allowed.
Example	`gamma(5.5)`
Result	`52.34277778455352`

`gcd(x, y)`

Description	Computes the greatest common divisor of x and y.
Example	`gcd(42, 57)`
Result	3

`greatest_common_divisor(x, y)`

Description	Computes the greatest common divisor of x and y.
Example	`greatest_common_divisor(42, 57)`
Result	3

`greatest(x1, x2, ...)`

Description	Selects the largest value.
Example	`greatest(3, 2, 4, 4)`
Result	4

`isfinite(x)`

Description	Returns true if the floating point value is finite, false otherwise.
Example	`isfinite(5.5)`
Result	`true`

`isinf(x)`

Description	Returns true if the floating point value is infinite, false otherwise.
Example	`isinf('Infinity'::float)`
Result	`true`

`isnan(x)`

Description	Returns true if the floating point value is not a number, false otherwise.
Example	`isnan('NaN'::float)`
Result	`true`

`lcm(x, y)`

Description	Computes the least common multiple of x and y.
Example	`lcm(42, 57)`

Result	798

least_common_multiple(x, y)

Description	Computes the least common multiple of x and y.
Example	`least_common_multiple(42, 57)`
Result	798

least(x1, x2, ...)

Description	Selects the smallest value.
Example	`least(3, 2, 4, 4)`
Result	2

lgamma(x)

Description	Computes the log of the gamma function.
Example	`lgamma(2)`
Result	0

ln(x)

Description	Computes the natural logarithm of x.
Example	`ln(2)`
Result	0.693

log(x)

Description	Computes the base-10 log of x.
Example	`log(100)`
Result	2

log10(x)

Description	Alias of `log`. Computes the base-10 log of x.
Example	`log10(1000)`
Result	3

log2(x)

Description	Computes the base-2 log of x.
Example	`log2(8)`
Result	3

multiply(x, y)

Description	Alias for `x * y`.
Example	`multiply(2, 3)`
Result	6

nextafter(x, y)

Description	Return the next floating point value after x in the direction of y.
Example	`nextafter(1::float, 2::float)`
Result	`1.0000001`

pi()

Description	Returns the value of pi.
Example	`pi()`
Result	`3.141592653589793`

pow(x, y)

Description	Computes x to the power of y.
Example	`pow(2, 3)`
Result	8

power(x, y)

Description	Alias of pow. Computes x to the power of y.
Example	`power(2, 3)`
Result	8

radians(x)

Description	Converts degrees to radians.
Example	`radians(90)`

Result	`1.5707963267948966`

`random()`

Description	Returns a random number x in the range `0.0 <= x < 1.0`.
Example	`random()`
Result	various

`round_even(v NUMERIC, s INTEGER)`

Description	Alias of `roundbankers(v, s)`. Round to s decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
Example	`round_even(24.5, 0)`
Result	`24.0`

`roundbankers(v NUMERIC, s INTEGER)`

Description	Alias of `round_even(v, s)`. Round to s decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
Example	`roundbankers(24.5, 0)`
Result	`24.0`

`round(v NUMERIC, s INTEGER)`

Description	Round to s decimal places. Values `s < 0` are allowed.
Example	`round(42.4332, 2)`
Result	`42.43`

`setseed(x)`

Description	Sets the seed to be used for the random function.
Example	`setseed(0.42)`

`sign(x)`

Description	Returns the sign of x as -1, 0 or 1.
Example	`sign(-349)`
Result	`-1`

signbit(x)

Description	Returns whether the signbit is set or not.
Example	`signbit(-1.0)`
Result	`true`

sin(x)

Description	Computes the sin of x.
Example	`sin(pi() / 6)`
Result	`0.49999999999999994`

sinh(x)

Description	Computes the hyperbolic sine of x.
Example	`sinh(1)`
Result	`1.1752011936438014`

sqrt(x)

Description	Returns the square root of the number.
Example	`sqrt(9)`
Result	`3`

subtract(x, y)

Description	Alias for `x - y`.
Example	`subtract(2, 3)`
Result	`-1`

tan(x)

Description	Computes the tangent of x.
Example	`tan(pi() / 4)`
Result	`0.9999999999999999`

tanh(x)

Description	Computes the hyperbolic tangent of x.
Example	`tanh(1)`

Result	`0.7615941559557649`

`trunc(x)`

Description	Truncates the number.
Example	`trunc(17.4)`
Result	`17`

`xor(x, y)`

Description	Bitwise XOR.
Example	`xor(17, 5)`
Result	`20`

Pattern Matching

There are four separate approaches to pattern matching provided by DuckDB: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), a `GLOB` operator, and POSIX-style regular expressions.

`LIKE`

The `LIKE` expression returns `true` if the string matches the supplied pattern. (As expected, the `NOT LIKE` expression returns `false` if `LIKE` returns `true`, and vice versa. An equivalent expression is `NOT (string LIKE pattern)`.)

If pattern does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in pattern stands for (matches) any single character; a percent sign (%) matches any sequence of zero or more characters.

`LIKE` pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

Some examples:

```
SELECT 'abc' LIKE 'abc';    -- true
SELECT 'abc' LIKE 'a%' ;    -- true
SELECT 'abc' LIKE '_b_';    -- true
SELECT 'abc' LIKE 'c';      -- false
SELECT 'abc' LIKE 'c%' ;    -- false
SELECT 'abc' LIKE '%c';     -- true
SELECT 'abc' NOT LIKE '%c'; -- false
```

The keyword `ILIKE` can be used instead of `LIKE` to make the match case-insensitive according to the active locale:

```
SELECT 'abc' ILIKE '%C'; -- true
```

```
SELECT 'abc' NOT ILIKE '%C'; -- false
```

To search within a string for a character that is a wildcard (% or _), the pattern must use an `ESCAPE` clause and an escape character to indicate the wildcard should be treated as a literal character instead of a wildcard. See an example below.

Additionally, the function `like_escape` has the same functionality as a `LIKE` expression with an `ESCAPE` clause, but using function syntax. See the Text Functions page for details.

Search for strings with a then a literal percent sign then c:

```sql
SELECT 'a%c' LIKE 'a$%c' ESCAPE '$'; -- true
SELECT 'azc' LIKE 'a$%c' ESCAPE '$'; -- false
```

Case-insensitive `ILIKE` with ESCAPE:

```sql
SELECT 'A%c' ILIKE 'a$%c' ESCAPE '$'; -- true
```

There are also alternative characters that can be used as keywords in place of `LIKE` expressions. These enhance PostgreSQL compatibility.

PostgreSQL-style	`LIKE`-style
`~~`	`LIKE`
`!~~`	`NOT LIKE`
`~~*`	`ILIKE`
`!~~*`	`NOT ILIKE`

`SIMILAR TO`

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using a regular expression. Like `LIKE`, the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a regular set). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with `LIKE`, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than `LIKE` does.

Some examples:

```sql
SELECT 'abc' SIMILAR TO 'abc';        -- true
SELECT 'abc' SIMILAR TO 'a';          -- false
SELECT 'abc' SIMILAR TO '.*(b|d).*'; -- true
SELECT 'abc' SIMILAR TO '(b|c).*';    -- false
SELECT 'abc' NOT SIMILAR TO 'abc';    -- false
```

> In PostgreSQL, the `~` and `!~` operators perform POSIX regular expression matching, not `SIMILAR TO` matching. DuckDB follows the same partial-matching semantics by default, see the PostgreSQL compatibility page.

Globbing

DuckDB supports file name expansion, also known as globbing, for discovering files. DuckDB's glob syntax uses the question mark (`?`) wildcard to match any single character and the asterisk (`*`) to match zero or more characters. In addition, you can use the bracket syntax (`[...]`) to match any single character contained within the brackets, or within the character range specified by the brackets. An exclamation mark (`!`) may be used inside the first bracket to search for a character that is not contained within the brackets. To learn more, visit the "glob (programming)" Wikipedia page.

`GLOB`

The `GLOB` operator returns `true` or `false` if the string matches the `GLOB` pattern. The `GLOB` operator is most commonly used when searching for filenames that follow a specific pattern (for example a specific file extension).

Some examples:

```
SELECT 'best.txt' GLOB '*.txt';            -- true
SELECT 'best.txt' GLOB '????.txt';         -- true
SELECT 'best.txt' GLOB '?.txt';            -- false
SELECT 'best.txt' GLOB '[abc]est.txt';     -- true
SELECT 'best.txt' GLOB '[a-z]est.txt';     -- true
```

The bracket syntax is case-sensitive:

```
SELECT 'Best.txt' GLOB '[a-z]est.txt';     -- false
SELECT 'Best.txt' GLOB '[a-zA-Z]est.txt';  -- true
```

The `!` applies to all characters within the brackets:

```
SELECT 'Best.txt' GLOB '[!a-zA-Z]est.txt'; -- false
```

To negate a GLOB operator, negate the entire expression:

```
SELECT NOT 'best.txt' GLOB '*.txt';        -- false
```

Three tildes (~~~) may also be used in place of the `GLOB` keyword.

GLOB-style	Symbolic-style
GLOB	~~~

Glob Function to Find Filenames

The glob pattern matching syntax can also be used to search for filenames using the `glob` table function. It accepts one parameter: the path to search (which may include glob patterns).

Search the current directory for all files:

```
SELECT * FROM glob('*');
```

file
duckdb.exe
test.csv
test.json
test.parquet
test2.csv
test2.parquet
todos.json

Globbing Semantics

The `**` wildcard matches any number of subdirectories, including none, so `**/<filename>{:.language-sql .highlight}` also returns a file named `<filename>{:.language-sql .highlight}` in the top-level directory. For example, with a `README.md` file present both in the current directory and in its `sub` subdirectory, the following query returns both files:

```
SELECT * FROM glob('**/README.md');
```

file
README.md
sub/README.md

Regular Expressions

DuckDB's regular expression support is documented on the Regular Expressions page. DuckDB supports some PostgreSQL-style operators for regular expression matching:

PostgreSQL-style	Equivalent expression
`~`	`regexp_matches`
`!~`	NOT `regexp_matches`
`~*`	`regexp_matches` with case-insensitive matching
`!~*`	NOT `regexp_matches` with case-insensitive matching

> New As of DuckDB 2.0, the `~` and `!~` operators perform partial matching by default, and the `~*` and `!~*` operators are supported. In earlier versions, `~` and `!~` performed full matching.

By default, these operators perform partial matching, following PostgreSQL's semantics. This is controlled by the deprecated `regex_match_operator_semantics` setting. Setting it to `full` restores the legacy behavior in which `~` and `!~` map to `regexp_full_match`, but this setting is deprecated and will be removed in a future release.

Regular Expressions

DuckDB offers pattern matching operators (`LIKE`, `SIMILAR TO`, `GLOB`), as well as support for regular expressions via functions.

Regular Expression Syntax

DuckDB uses the RE2 library as its regular expression engine. For the regular expression syntax, see the RE2 docs.

Functions

All functions accept an optional set of options.

Name	Description
`regexp_extract(string, pattern[, group = 0][, options])`	If `string` contains the regexp `pattern`, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to `0`. A set of optional `options` can be set.
`regexp_extract(string, pattern, name_list[, options])`	If `string` contains the regexp `pattern`, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values.

Name	Description
`regexp_extract_all(string, regex[, group = 0][, options])`	Finds non-overlapping occurrences of `regex` in `string` and returns the corresponding values of `group`.
`regexp_extract_all(string, regex, name_list[, options])`	Finds non-overlapping occurrences of `regex` in `string` and returns the capturing groups as a list of structs with corresponding names from `name_list`.
`regexp_full_match(string, regex[, options])`	Returns `true` if the entire `string` matches the `regex`.
`regexp_matches(string, pattern[, options])`	Returns `true` if `string` contains the regexp `pattern`, `false` otherwise.
`regexp_replace(string, pattern, replacement[, options])`	If `string` contains the regexp `pattern`, replaces the matching part with `replacement`. By default, only the first occurrence is replaced. A set of optional `options`, including the global flag `g`, can be set.
`regexp_split_to_array(string, regex[, options])`	Alias of `string_split_regex`. Splits the `string` along the `regex`.
`regexp_split_to_table(string, regex[, options])`	Splits the `string` along the `regex` and returns a row for each part.

`regexp_extract(string, pattern[, group = 0][, options])`

Description	If `string` contains the regexp `pattern`, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional `options` can be set.
Example	`regexp_extract('abc', '([a-z])(b)', 1)`
Result	a

`regexp_extract(string, pattern, name_list[, options])`

Description	If `string` contains the regexp `pattern`, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional `options` can be set.
Example	`regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])`
Result	`{'y':'2023', 'm':'04', 'd':'15'}`

`regexp_extract_all(string, regex[, group = 0][, options])`

Description	Finds non-overlapping occurrences of `regex` in `string` and returns the corresponding values of `group`. A set of optional `options` can be set.
Example	`regexp_extract_all('Peter: 33, Paul:14', '(\w+):\s*(\d+)', 2)`
Result	`[33, 14]`

`regexp_extract_all(string, regex, name_list[, options])`

Description	Finds non-overlapping occurrences of `regex` in `string` and returns the capturing groups as a list of structs with corresponding names from `name_list`. A set of optional `options` can be set.
Example	`regexp_extract_all('Peter: 33, Paul: 14', '(\w+):\s*(\d+)', ['name', 'age'])`
Result	`[{'name': Peter, 'age': 33}, {'name': Paul, 'age': 14}]`

`regexp_full_match(string, regex[, options])`

Description	Returns `true` if the entire `string` matches the `regex`. A set of optional `options` can be set.
Example	`regexp_full_match('anabanana', '(an)*')`
Result	`false`

`regexp_matches(string, pattern[, options])`

Description	Returns `true` if `string` contains the regexp `pattern`, `false` otherwise. A set of optional `options` can be set.
Example	`regexp_matches('anabanana', '(an)*')`
Result	`true`

`regexp_replace(string, pattern, replacement[, options])`

Description	If `string` contains the regexp `pattern`, replaces the matching part with `replacement`. By default, only the first occurrence is replaced. A set of optional `options`, including the global flag g, can be set.
Example	`regexp_replace('hello', '[lo]', '-')`
Result	`he-lo`

`regexp_split_to_array(string, regex[, options])`

Description	Alias of `string_split_regex`. Splits the `string` along the `regex`. A set of optional `options` can be set.
Example	`regexp_split_to_array('hello world; 42', ';? ')`
Result	`['hello', 'world', '42']`

`regexp_split_to_table(string, regex[, options])`

Description	Splits the `string` along the `regex` and returns a row for each part. A set of optional `options` can be set.
Example	`regexp_split_to_table('hello world; 42', ';? ')`

Result	Three rows: `'hello'`, `'world'`, `'42'`

The `regexp_matches` function is similar to the `SIMILAR TO` operator, however, it does not require the entire string to match. Instead, `regexp_matches` returns `true` if the string merely contains the pattern (unless the special tokens `^` and `$` are used to anchor the regular expression to the start and end of the string). Below are some examples:

```
SELECT regexp_matches('abc', 'abc');       -- true
SELECT regexp_matches('abc', '^abc$');     -- true
SELECT regexp_matches('abc', 'a');         -- true
SELECT regexp_matches('abc', '^a$');       -- false
SELECT regexp_matches('abc', '.*(b|d).*'); -- true
SELECT regexp_matches('abc', '(b|c).*');   -- true
SELECT regexp_matches('abc', '^(b|c).*');  -- false
SELECT regexp_matches('abc', '(?i)A');     -- true
SELECT regexp_matches('abc', 'A', 'i');    -- true
```

Options for Regular Expression Functions

The regex functions support the following `options`.

Option	Description
`'c'`	Case-sensitive matching
`'i'`	Case-insensitive matching
`'l'`	Match literals instead of regular expression tokens
`'m'`, `'n'`, `'p'`	Newline sensitive matching
`'g'`	Global replace, only available for `regexp_replace`
`'s'`	Non-newline sensitive matching

For example:

```
SELECT regexp_matches('abcd', 'ABC', 'c'); -- false
SELECT regexp_matches('abcd', 'ABC', 'i'); -- true
SELECT regexp_matches('ab^/$cd', '^/$', 'l'); -- true
SELECT regexp_matches(E'hello\nworld', 'hello.world', 'p'); -- false
SELECT regexp_matches(E'hello\nworld', 'hello.world', 's'); -- true
```

Using `regexp_matches`

The `regexp_matches` operator will be optimized to the `LIKE` operator when possible. To achieve best performance, the `'c'` option (case-sensitive matching) should be passed if applicable. Note that by default the RE2 library doesn't match the `.` character to newline.

Original	Optimized equivalent
`regexp_matches('hello world', '^hello', 'c')`	`prefix('hello world', 'hello')`
`regexp_matches('hello world', 'world$', 'c')`	`suffix('hello world', 'world')`
`regexp_matches('hello world', 'hello.world', 'c')`	`LIKE 'hello_world'`
`regexp_matches('hello world', 'he.*rld', 'c')`	`LIKE '%he%rld'`

Using `regexp_replace`

The `regexp_replace` function can be used to replace the part of a string that matches the regexp pattern with a replacement string. The notation `\d` (where `d` is a number indicating the group) can be used to refer to groups captured in the regular expression in the replacement string. Note that by default, `regexp_replace` only replaces the first occurrence of the regular expression. To replace all occurrences, use the global replace (`g`) flag.

Some examples for using `regexp_replace`:

```
SELECT regexp_replace('abc', '(b|c)', 'X');         -- aXc
SELECT regexp_replace('abc', '(b|c)', 'X', 'g');    -- aXX
SELECT regexp_replace('abc', '(b|c)', '\1\1\1\1'); -- abbbbc
SELECT regexp_replace('abc', '(.*)c', '\1e');       -- abe
SELECT regexp_replace('abc', '(a)(b)', '\2\1');     -- bac
```

Using `regexp_extract`

The `regexp_extract` function is used to extract a part of a string that matches the regexp pattern. A specific capturing group within the pattern can be extracted using the `group` parameter. If `group` is not specified, it defaults to 0, extracting the first match with the whole pattern.

```
SELECT regexp_extract('abc', '.b.');           -- abc
SELECT regexp_extract('abc', '.b.', 0);        -- abc
SELECT regexp_extract('abc', '.b.', 1);        -- (empty)
SELECT regexp_extract('abc', '([a-z])(b)', 1); -- a
SELECT regexp_extract('abc', '([a-z])(b)', 2); -- b
```

> New The following feature was introduced in DuckDB 2.0.

There is also an overload that takes regex options as the third argument, with no group required:

```
SELECT regexp_extract('ABC', 'abc', 'i'); -- ABC
```

The `regexp_extract` function also supports a `name_list` argument, which is a `LIST` of strings. Using `name_list`, the `regexp_extract` will return the corresponding capture groups as fields of a `STRUCT`:

```
SELECT regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd']);
```

```
{'y': 2023, 'm': 04, 'd': 15}
```

```
SELECT regexp_extract('2023-04-15 07:59:56', '^(\d+)-(\d+)-(\d+) (\d+):(\d+):(\d+)', ['y', 'm', 'd']);
```

```
{'y': 2023, 'm': 04, 'd': 15}
```

```
SELECT regexp_extract('duckdb_0_7_1', '^(\w+)_(\d+)_(\d+)', ['tool', 'major', 'minor', 'fix']);
```

```
Binder Error:
Not enough capturing groups (3) for provided names (4)
```

If the number of column names is less than the number of capture groups, then only the first groups are returned. If the number of column names is greater, then an error is generated.

Limitations

Regular expressions only support 9 capture groups: `\1`, `\2`, `\3`, ..., `\9`. Capture groups with two or more digits are not supported.

Struct Functions

The functions on this page work on STRUCT values. Functions that access entries by position, such as `struct[idx]`, `struct_extract_at`, `struct_contains`, `struct_position` and `struct_values`, also accept TUPLE values, i.e., structs with unnamed entries. `struct_concat` accepts either only STRUCTs or only TUPLEs. Functions that work with entry names, i.e., `struct_insert`, `struct_keys`, `struct_update` and `remap_struct`, require a STRUCT.

Name	Description
`struct.entry`	Dot notation that serves as an alias for `struct_extract` from named STRUCTs.
`struct[entry]`	Bracket notation that serves as an alias for `struct_extract` from named STRUCTs.
`struct[idx]`	Bracket notation that serves as an alias for `struct_extract` from TUPLEs, using an index (1-based).
`remap_struct(input, target_type, mapping, defaults)`	Remap a STRUCT, LIST, or MAP value into the shape given by the type of `target_type`, renaming and reordering fields, descending into nested types, and filling missing fields from `defaults`.
`row(any, ...)`	Create a TUPLE containing the argument values.
`struct_concat(structs...)`	Merge the multiple `structs` into a single STRUCT.
`struct_contains(struct, entry)`	Check if the STRUCT contains the specified entry.
`struct_extract(struct, 'entry')`	Extract the named entry from the STRUCT.
`struct_extract(struct, idx)`	Extract the entry from a TUPLE using an index (1-based).
`struct_extract_at(struct, idx)`	Extract the entry from a STRUCT or TUPLE using an index (1-based).
`struct_insert(struct, name := any, ...)`	Add field(s) to an existing STRUCT.
`struct_keys(struct)`	Return the names of the entries of a STRUCT as a list.
`struct_pack(name := any, ...)`	Create a STRUCT containing the argument values. The entry name will be the bound variable name.
`struct_position(struct, entry)`	Return the index of the entry within the STRUCT (1-based), or NULL if not found.
`struct_update(struct, name := any, ...)`	Add or update field(s) of an existing STRUCT.
`struct_values(struct)`	Return the values of a STRUCT as a TUPLE.

`struct.entry`

Description	Dot notation that serves as an alias for `struct_extract` from named STRUCTs.
Example	`({'i': 3, 's': 'string'}).i`
Result	3

`struct[entry]`

Description	Bracket notation that serves as an alias for `struct_extract` from named STRUCTs.
Example	`({'i': 3, 's': 'string'})['i']`
Result	3

`struct[idx]`

Description	Bracket notation that serves as an alias for `struct_extract` from TUPLEs, using an index (1-based).
Example	`(row(42, 84))[1]`
Result	42

`remap_struct(input, target_type, mapping, defaults)`

Description	Remap an `input` value of type STRUCT, `LIST`, or `MAP` into the shape declared by the type of `target_type`, renaming and reordering fields, descending into nested types, and filling target fields that have no source counterpart from `defaults`. Only the type of `target_type` is used, not its value, so any value of the desired type works; a typed `NULL` literal (e.g. `NULL::STRUCT(v1 INT, v2 INT)`) is a convenient way to declare the target shape when no suitable value is at hand. `mapping` is a STRUCT keyed by target field name whose values name the source field to read from; `defaults` is a STRUCT of constant values for the remaining target fields. Every target field must be covered by either `mapping` or `defaults`.
Example	`remap_struct({'i': 1, 'j': 2}, NULL::STRUCT(v1 INT, v2 INT, v3 INT), {'v1': 'j', 'v3': 'i'}, {'v2': NULL::INTEGER})`
Result	`{'v1': 2, 'v2': NULL, 'v3': 1}`

To remap a target field that is itself a STRUCT, give its mapping value as `ROW(source_field, {inner_mapping})`, and nest its defaults the same way. A `LIST`'s single child is addressed by the mapping key `'list'`, and a `MAP`'s children by the keys `'key'` and `'value'`.

`row(any, ...)`

Description	Create a TUPLE containing the argument values.
Example	`row(i, i % 4, i / 4)`
Result	`(10, 2, 2.5)`

`struct_concat(structs...)`

Description	Merge the multiple `structs` into a single STRUCT.
Example	`struct_concat(struct_pack(i := 4), struct_pack(s := 'string'))`
Result	`{'i': 4, 's': string}`

`struct_contains(struct, entry)`

Description	Check if the STRUCT contains the specified entry.
Example	`struct_contains(row(1, 2, 3), 2)`
Result	`true`
Alias	`struct_has`

`struct_extract(struct, 'entry')`

Description	Extract the named entry from the STRUCT.
Example	`struct_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')`
Result	`3`

`struct_extract(struct, idx)`

Description	Extract the entry from a TUPLE using an index (1-based).
Example	`struct_extract(row(42, 84), 1)`
Result	`42`

`struct_extract_at(struct, idx)`

Description	Extract the entry from a STRUCT or TUPLE using an index (1-based).
Example	`struct_extract_at({'v1': 10, 'v2': 20, 'v3': 3}, 2)`
Result	`20`

`struct_insert(struct, name := any, ...)`

Description	Add field(s) to an existing STRUCT.
Example	`struct_insert({'a': 1}, b := 2)`
Result	`{'a': 1, 'b': 2}`

`struct_keys(struct)`

Description	Return the names of the entries of a STRUCT as a list.
Example	`struct_keys({'a': 1, 'b': 2})`
Result	`[a, b]`

`struct_pack(name := any, ...)`

Description	Create a STRUCT containing the argument values. The entry name will be the bound variable name.
Example	`struct_pack(i := 4, s := 'string')`
Result	`{'i': 4, 's': string}`

`struct_position(struct, entry)`

Description	Return the index of the entry within the STRUCT (1-based), or NULL if not found.
Example	`struct_position(row(1, 2, 3), 2)`
Result	2
Alias	`struct_indexof`

`struct_update(struct, name := any, ...)`

Description	Add or update field(s) of an existing STRUCT.
Example	`struct_update({'a': 1, 'b': 2}, b := 3, c := 4)`
Result	`{'a': 1, 'b': 3, 'c': 4}`

`struct_values(struct)`

Description	Return the values of a STRUCT as a TUPLE.
Example	`struct_values({'a': 1, 'b': 2, 'c': 3})`
Result	`(1, 2, 3)`

Text Functions

Text Functions and Operators

This section describes functions and operators for examining and manipulating STRING values.

Function	Description
`string[index]`	Extracts a single character using a (1-based) `index`.
`string[begin:end]`	Extracts a string using slice conventions similar to Python. Missing `begin` or `end` arguments are interpreted as the beginning or end of the list respectively. Negative values are accepted.
`string LIKE target`	Returns `true` if the `string` matches the like specifier (see Pattern Matching).
`string SIMILAR TO regex`	Returns `true` if the `string` matches the `regex` (see Pattern Matching).
`string ^@ search_string`	Alias for `starts_with`.
`arg1 || arg2`	Concatenates two strings, lists, or blobs. Any NULL input results in NULL. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.
`array_extract(string, index)`	Extracts a single character from a `string` using a (1-based) `index`.

Function	Description
`array_slice(list, begin, end)`	Extracts a sublist or substring using slice conventions. Negative values are accepted.
`ascii(string)`	Returns an integer that represents the Unicode code point of the first character of the `string`.
`bar(x, min, max[, width])`	Draws a band whose width is proportional to (`x - min`) and equal to `width` characters when `x` = `max`. `width` defaults to 80.
`base64(blob)`	Alias for `to_base64`.
`bin(string)`	Converts the `string` to binary representation.
`bit_length(string)`	Number of bits in a `string`.
`char_length(string)`	Alias for `length`.
`character_length(string)`	Alias for `length`.
`chr(code_point)`	Returns a character which is corresponding the ASCII code value or Unicode code point.
`concat(value, ...)`	Concatenates multiple strings or lists. `NULL` inputs are skipped. See also operator `||`.
`concat_ws(separator, string, ...)`	Concatenates many strings, separated by `separator`. `NULL` inputs are skipped.
`contains(string, search_string)`	Returns `true` if `search_string` is found within `string`. Note that collations are not supported.
`ends_with(string, search_string)`	Alias for `suffix`.
`format(format, ...)`	Formats a string using the fmt syntax.
`formatReadableDecimalSize(integer)`	Converts `integer` to a human-readable representation using units based on powers of 10 (KB, MB, GB, etc.).
`format_bytes(integer)`	Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
`from_base64(string)`	Converts a base64 encoded `string` to a character string (`BLOB`).
`from_binary(value)`	Alias for `unbin`.
`from_hex(value)`	Alias for `unhex`.
`greatest(arg1, ...)`	Returns the largest value in lexicographical order. Note that lowercase characters are considered larger than uppercase characters and collations are not supported.
`hash(value, ...)`	Returns a `UBIGINT` with the hash of the `value`. Note that this is not a cryptographic hash.
`hex(string)`	Converts the `string` to hexadecimal representation.
`ilike_escape(string, like_specifier, escape_character)`	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`instr(string, search_string)`	Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`lcase(string)`	Alias for `lower`.
`least(arg1, ...)`	Returns the smallest value in lexicographical order. Note that uppercase characters are considered smaller than lowercase characters and collations are not supported.
`left(string, count)`	Extracts the left-most count characters.
`left_grapheme(string, count)`	Extracts the left-most count grapheme clusters.
`len(string)`	Alias for `length`.

Function	Description
`length(string)`	Number of characters in `string`.
`length_grapheme(string)`	Number of grapheme clusters in `string`.
`like_escape(string, like_specifier, escape_character)`	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`lower(string)`	Converts `string` to lower case.
`lpad(string, count, character)`	Pads the `string` with the `character` on the left until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
`ltrim(string[, characters])`	Removes any occurrences of any of the `characters` from the left side of the `string`. `characters` defaults to `space`.
`md5(string)`	Returns the MD5 hash of the `string` as a `VARCHAR`.
`md5_number(string)`	Returns the MD5 hash of the `string` as a `HUGEINT`.
`md5_number_lower(string)`	Returns the lower 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`md5_number_upper(string)`	Returns the upper 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`nfc_normalize(string)`	Converts `string` to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.
`not_ilike_escape(string, like_specifier, escape_character)`	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`not_like_escape(string, like_specifier, escape_character)`	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`ord(string)`	Alias for `unicode`.
`parse_dirname(path[, separator])`	Returns the top-level directory name from the given `path`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`parse_dirpath(path[, separator])`	Returns the head of the `path` (the pathname until the last slash) similarly to Python's `os.path.dirname`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`parse_filename(string[, trim_extension][, separator])`	Returns the last component of the `path` similarly to Python's `os.path.basename` function. If `trim_extension` is `true`, the file extension will be removed (defaults to `false`). `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`parse_path(path[, separator])`	Returns a list of the components (directories and filename) in the `path` similarly to Python's `pathlib.parts` function. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`position(search_string IN string)`	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`position(string, search_string)`	Alias for `instr`.
`prefix(string, search_string)`	Returns `true` if `string` starts with `search_string`.
`printf(format, ...)`	Formats a `string` using printf syntax.

Function	Description
`read_text(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
`regexp_escape(string)`	Escapes special patterns to turn `string` into a regular expression similarly to Python's `re.escape` function.
`regexp_extract(string, regex[, group][, options])`	If `string` contains the `regex` pattern, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional regex `options` can be set.
`regexp_extract(string, regex, name_list[, options])`	If `string` contains the `regex` pattern, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional regex `options` can be set.
`regexp_extract_all(string, regex[, group][, options])`	Finds non-overlapping occurrences of the `regex` in the `string` and returns the corresponding values of the capturing `group`. A set of optional regex `options` can be set.
`regexp_extract_all(string, regex, name_list[, options])`	Finds non-overlapping occurrences of `regex` in `string` and returns the capturing groups as a list of structs with corresponding names from `name_list`. A set of optional regex `options` can be set.
`regexp_full_match(string, regex[, col2])`	Returns `true` if the entire `string` matches the `regex`. A set of optional regex `options` can be set.
`regexp_matches(string, regex[, options])`	Returns `true` if `string` contains the `regex`, `false` otherwise. A set of optional regex `options` can be set.
`regexp_replace(string, regex, replacement[, options])`	If `string` contains the `regex`, replaces the matching part with `replacement`. A set of optional regex `options` can be set.
`regexp_split_to_array(string, regex[, options])`	Alias for `string_split_regex`.
`regexp_split_to_table(string, regex)`	Splits the `string` along the `regex` and returns a row for each part.
`repeat(string, count)`	Repeats the `string` `count` number of times.
`replace(string, source, target)`	Replaces any occurrences of the `source` with `target` in `string`.
`reverse(string)`	Reverses the `string`.
`right(string, count)`	Extract the right-most `count` characters.
`right_grapheme(string, count)`	Extracts the right-most `count` grapheme clusters.
`rpad(string, count, character)`	Pads the `string` with the `character` on the right until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
`rtrim(string[, characters])`	Removes any occurrences of any of the `characters` from the right side of the `string`. `characters` defaults to `space`.
`sha1(value)`	Returns a VARCHAR with the SHA-1 hash of the `value`.
`sha256(value)`	Returns a VARCHAR with the SHA-256 hash of the `value`.
`split(string, separator)`	Alias for `string_split`.

Function	Description
split_part(string, separator, index)	Splits the string along the separator and returns the data at the (1-based) index of the list. If the index is outside the bounds of the list, return an empty string (to match PostgreSQL's behavior).
starts_with(string, search_string)	Returns true if string begins with search_string.
str_split(string, separator)	Alias for string_split.
str_split_regex(string, regex[, options])	Alias for string_split_regex.
string_split(string, separator)	Splits the string along the separator.
string_split_regex(string, regex[, options])	Splits the string along the regex. A set of optional regex options can be set.
string_to_array(string, separator)	Alias for string_split.
strip_accents(string)	Strips accents from string.
strlen(string)	Number of bytes in string.
strpos(string, search_string)	Alias for instr.
substr(string, start[, length])	Alias for substring.
substring(string, start[, length])	Extracts substring starting from character start up to the end of the string. If optional argument length is set, extracts a substring of length characters instead. Note that a start value of 1 refers to the first character of the string.
substring_grapheme(string, start[, length])	Extracts substring starting from grapheme clusters start up to the end of the string. If optional argument length is set, extracts a substring of length grapheme clusters instead. Note that a start value of 1 refers to the first character of the string.
suffix(string, search_string)	Returns true if string ends with search_string. Note that collations are not supported.
to_base(number, radix[, min_length])	Converts number to a string in the given base radix, optionally padding with leading zeros to min_length.
to_base64(blob)	Converts a blob to a base64 encoded string.
to_binary(string)	Alias for bin.
to_hex(string)	Alias for hex.
translate(string, from, to)	Replaces each character in string that matches a character in the from set with the corresponding character in the to set. If from is longer than to, occurrences of the extra characters in from are deleted.
trim(string[, characters])	Removes any occurrences of any of the characters from either side of the string. characters defaults to space.
ucase(string)	Alias for upper.
unbin(value)	Converts a value from binary representation to a blob.
unhex(value)	Converts a value from hexadecimal representation to a blob.

Function	Description
`unicode(string)`	Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.
`upper(string)`	Converts `string` to upper case.
`url_decode(string)`	Decodes a URL from a representation using Percent-Encoding.
`url_encode(string)`	Encodes a URL to a representation using Percent-Encoding.

`string[index]`

Description	Extracts a single character using a (1-based) `index`.
Example	`'DuckDB'[4]`
Result	`k`
Alias	`array_extract`

`string[begin:end]`

Description	Extracts a string using slice conventions similar to Python. Missing `begin` or `end` arguments are interpreted as the beginning or end of the list respectively. Negative values are accepted.
Example	`'DuckDB'[:4]`
Result	`Duck`
Alias	`array_slice`

`string LIKE target`

Description	Returns `true` if the `string` matches the like specifier (see Pattern Matching).
Example	`'hello' LIKE '%lo'`
Result	`true`

`string SIMILAR TO regex`

Description	Returns `true` if the `string` matches the `regex` (see Pattern Matching).
Example	`'hello' SIMILAR TO 'l+'`
Result	`false`
Alias	`regexp_full_match`

`arg1 || arg2`

Description	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.
Example 1	`'Duck' || 'DB'`

Result	DuckDB
Example 2	`[1, 2, 3] || [4, 5, 6]`
Result	`[1, 2, 3, 4, 5, 6]`
Example 3	`'\xAA'::BLOB || '\xBB'::BLOB`
Result	\xAA\xBB

`array_extract(string, index)`

Description	Extracts a single character from a `string` using a (1-based) `index`.
Example	`array_extract('DuckDB', 2)`
Result	u

`array_slice(list, begin, end)`

Description	Extracts a sublist or substring using slice conventions. Negative values are accepted.
Example 1	`array_slice('DuckDB', 3, 4)`
Result	ck
Example 2	`array_slice('DuckDB', 3, NULL)`
Result	NULL
Example 3	`array_slice('DuckDB', 0, -3)`
Result	Duck
Alias	`list_slice`

`ascii(string)`

Description	Returns an integer that represents the Unicode code point of the first character of the `string`.
Example	`ascii('Ω')`
Result	937

`bar(x, min, max[, width])`

Description	Draws a band whose width is proportional to (`x - min`) and equal to `width` characters when x = `max`. `width` defaults to 80.
Example	`bar(5, 0, 20, 10)`
Result	██

`bin(string)`

Description	Converts the `string` to binary representation.

Example	`bin('Aa')`
Result	`0100000101100001`
Alias	`to_binary`

`bit_length(string)`

Description	Number of bits in a `string`.
Example	`bit_length('abc')`
Result	`24`

`chr(code_point)`

Description	Returns a character which is corresponding the ASCII code value or Unicode code point.
Example	`chr(65)`
Result	A

`concat(value, ...)`

Description	Concatenates multiple strings or lists. `NULL` inputs are skipped. See also operator `||`.
Example 1	`concat('Hello', ' ', 'World')`
Result	`Hello World`
Example 2	`concat([1, 2, 3], NULL, [4, 5, 6])`
Result	`[1, 2, 3, 4, 5, 6]`

`concat_ws(separator, string, ...)`

Description	Concatenates many strings, separated by `separator`. `NULL` inputs are skipped.
Example	`concat_ws(', ', 'Banana', 'Apple', 'Melon')`
Result	`Banana, Apple, Melon`

`contains(string, search_string)`

Description	Returns `true` if `search_string` is found within `string`.
Example	`contains('abc', 'a')`
Result	`true`

`format(format, ...)`

Description	Formats a string using the fmt syntax.
Example	`format('Benchmark "{}" took {} seconds', 'CSV', 42)`
Result	`Benchmark "CSV" took 42 seconds`

`formatReadableDecimalSize(integer)`

Description	Converts `integer` to a human-readable representation using units based on powers of 10 (KB, MB, GB, etc.).
Example	`formatReadableDecimalSize(16000)`
Result	`16.0 kB`

`format_bytes(integer)`

Description	Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
Example	`format_bytes(16_000)`
Result	`15.6 KiB`
Alias	`formatReadableSize`, `pg_size_pretty`

`from_base64(string)`

Description	Converts a base64 encoded `string` to a character string (`BLOB`).
Example	`from_base64('QQ==')`
Result	A

`greatest(arg1, ...)`

Description	Returns the largest value in lexicographical order. Note that lowercase characters are considered larger than uppercase characters and collations are not supported.
Example 1	`greatest(42, 84)`
Result	`84`
Example 2	`greatest('abc', 'bcd', 'cde', 'EFG')`
Result	`cde`

`hash(value, ...)`

Description	Returns a `UBIGINT` with the hash of the `value`. Note that this is not a cryptographic hash.
Example	`hash('🦆')`
Result	`4164431626903154684`

`hex(string)`

Description	Converts the `string` to hexadecimal representation.
Example	`hex('Hello')`
Result	`48656C6C6F`
Alias	`to_hex`

`ilike_escape(string, like_specifier, escape_character)`

Description	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`ilike_escape('A%c', 'a$%C', '$')`
Result	`true`

`instr(string, search_string)`

Description	Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`instr('test test', 'es')`
Result	2
Aliases	`position`, `strpos`

`least(arg1, ...)`

Description	Returns the smallest value in lexicographical order. Note that uppercase characters are considered smaller than lowercase characters and collations are not supported.
Example 1	`least(42, 84)`
Result	42
Example 2	`least('abc', 'bcd', 'cde', 'EFG')`
Result	EFG

`left(string, count)`

Description	Extracts the left-most count characters.
Example	`left('Hello🦆', 2)`
Result	He

`left_grapheme(string, count)`

Description	Extracts the left-most count grapheme clusters.

Example	`left_grapheme('👩🏻‍🤝‍👨🏽‍♀', 1)`
Result	👩🏻‍🤝‍👨

`length(string)`

Description	Number of characters in `string`.
Example	`length('Hello🦆')`
Result	6
Aliases	`char_length`, `character_length`, `len`

`length_grapheme(string)`

Description	Number of grapheme clusters in `string`.
Example	`length_grapheme('👩🏻‍🤝‍👨🏽‍♀')`
Result	2

`like_escape(string, like_specifier, escape_character)`

Description	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`like_escape('a%c', 'a$%c', '$')`
Result	`true`

`lower(string)`

Description	Converts `string` to lower case.
Example	`lower('Hello')`
Result	`hello`
Alias	`lcase`

`lpad(string, count, character)`

Description	Pads the `string` with the `character` on the left until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
Example	`lpad('hello', 8, '>')`
Result	`>>>hello`

`ltrim(string[, characters])`

Description	Removes any occurrences of any of the `characters` from the left side of the `string`. `characters` defaults to `space`.
Example 1	ltrim(' test ')
Result	test
Example 2	`ltrim('>>>>test<<', '><')`
Result	`test<<`

`md5(string)`

Description	Returns the MD5 hash of the `string` as a VARCHAR.
Example	`md5('abc')`
Result	`900150983cd24fb0d6963f7d28e17f72`

`md5_number(string)`

Description	Returns the MD5 hash of the `string` as a HUGEINT.
Example	`md5_number('abc')`
Result	`152195979970564155685860391459828531600`

`md5_number_lower(string)`

Description	Returns the lower 64-bit segment of the MD5 hash of the `string` as a UBIGINT.
Example	`md5_number_lower('abc')`
Result	`8250560606382298838`

`md5_number_upper(string)`

Description	Returns the upper 64-bit segment of the MD5 hash of the `string` as a UBIGINT.
Example	`md5_number_upper('abc')`
Result	`12704604231530709392`

`nfc_normalize(string)`

Description	Converts `string` to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.
Example	`nfc_normalize('ardèch')`
Result	`ardèch`

not_ilike_escape(string, like_specifier, escape_character)

Description	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`not_ilike_escape('A%c', 'a$%C', '$')`
Result	`false`

not_like_escape(string, like_specifier, escape_character)

Description	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`not_like_escape('a%c', 'a$%c', '$')`
Result	`false`

parse_dirname(path[, separator])

Description	Returns the top-level directory name from the given `path`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_dirname('path/to/file.csv', 'system')`
Result	`path`

parse_dirpath(path[, separator])

Description	Returns the head of the `path` (the pathname until the last slash) similarly to Python's `os.path.dirname`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_dirpath('path/to/file.csv', 'forward_slash')`
Result	`path/to`

parse_filename(string[, trim_extension][, separator])

Description	Returns the last component of the `path` similarly to Python's `os.path.basename` function. If `trim_extension` is `true`, the file extension will be removed (defaults to `false`). `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_filename('path/to/file.csv', true, 'forward_slash')`
Result	`file`

parse_path(path[, separator])

Description	Returns a list of the components (directories and filename) in the `path` similarly to Python's `pathlib.parts` function. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_path('path/to/file.csv', 'system')`
Result	`[path, to, file.csv]`

`position(search_string IN string)`

Description	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`position('b' IN 'abc')`
Result	2
Aliases	`instr`, `strpos`

`prefix(string, search_string)`

Description	Returns `true` if `string` starts with `search_string`.
Example	`prefix('abc', 'ab')`
Result	`true`

`printf(format, ...)`

Description	Formats a `string` using printf syntax.
Example	`printf('Benchmark "%s" took %d seconds', 'CSV', 42)`
Result	`Benchmark "CSV" took 42 seconds`

`read_text(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
Example	`read_text('hello.txt')`
Result	`hello\n`

`regexp_escape(string)`

Description	Escapes special patterns to turn `string` into a regular expression similarly to Python's `re.escape` function.
Example	`regexp_escape('https://duckdb.org')`
Result	`https\:\/\/duckdb\.org`

`regexp_extract(string, regex[, group][, options])`

Description	If `string` contains the `regex` pattern, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional regex `options` can be set.
Example	`regexp_extract('ABC', '([a-z])(b)', 1, 'i')`
Result	A

`regexp_extract(string, regex, name_list[, options])`

Description	If `string` contains the `regex` pattern, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional regex `options` can be set.
Example	`regexp_extract('John Doe', '([a-z]+) ([a-z]+)', ['first_name', 'last_name'], 'i')`
Result	`{'first_name': John, 'last_name': Doe}`

`regexp_extract_all(string, regex[, group][, options])`

Description	Finds non-overlapping occurrences of the `regex` in the `string` and returns the corresponding values of the capturing `group`. A set of optional regex `options` can be set.
Example	`regexp_extract_all('Peter: 33, Paul:14', '(\w+):\s*(\d+)', 2)`
Result	`[33, 14]`

`regexp_extract_all(string, regex, name_list[, options])`

Description	Finds non-overlapping occurrences of `regex` in `string` and returns the capturing groups as a list of structs with corresponding names from `name_list`. A set of optional regex `options` can be set.
Example	`regexp_extract_all('Peter: 33, Paul: 14', '(\w+):\s*(\d+)', ['name', 'age'])`
Result	`[{'name': Peter, 'age': 33}, {'name': Paul, 'age': 14}]`

`regexp_full_match(string, regex[, col2])`

Description	Returns `true` if the entire `string` matches the `regex`. A set of optional regex `options` can be set.
Example	`regexp_full_match('anabanana', '(an)*')`
Result	`false`

`regexp_matches(string, regex[, options])`

Description	Returns `true` if `string` contains the `regex`, `false` otherwise. A set of optional regex `options` can be set.
Example	`regexp_matches('anabanana', '(an)*')`
Result	`true`

`regexp_replace(string, regex, replacement[, options])`

Description	If `string` contains the `regex`, replaces the matching part with `replacement`. A set of optional regex `options` can be set.
Example	`regexp_replace('hello', '[lo]', '-')`
Result	`he-lo`

`regexp_split_to_table(string, regex)`

Description	Splits the `string` along the `regex` and returns a row for each part.
Example	`regexp_split_to_table('hello world; 42', ';? ')`
Result	Multiple rows: `'hello'`, `'world'`, `'42'`

`repeat(string, count)`

Description	Repeats the `string` `count` number of times.
Example	`repeat('A', 5)`
Result	AAAAA

`replace(string, source, target)`

Description	Replaces any occurrences of the `source` with `target` in `string`.
Example	`replace('hello', 'l', '-')`
Result	`he--o`

`reverse(string)`

Description	Reverses the `string`.
Example	`reverse('hello')`
Result	`olleh`

`right(string, count)`

Description	Extract the right-most `count` characters.

Example	`right('Hello🦆', 3)`
Result	`lo🦆`

`right_grapheme(string, count)`

Description	Extracts the right-most `count` grapheme clusters.
Example	`right_grapheme('👩🏻‍🤝‍👨🏽👩🏽', 1)`
Result	👩🏽

`rpad(string, count, character)`

Description	Pads the `string` with the `character` on the right until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
Example	`rpad('hello', 10, '<')`
Result	`hello<<<<<`

`rtrim(string[, characters])`

Description	Removes any occurrences of any of the `characters` from the right side of the `string`. `characters` defaults to space.
Example 1	rtrim(' test ')
Result	test
Example 2	`rtrim('>>>>test<<', '><')`
Result	`>>>>test`

`sha1(value)`

Description	Returns a VARCHAR with the SHA-1 hash of the `value`.
Example	`sha1('🦆')`
Result	`949bf843dc338be348fb9525d1eb535d31241d76`

`sha256(value)`

Description	Returns a VARCHAR with the SHA-256 hash of the `value`
Example	`sha256('🦆')`
Result	`d7a5c5e0d1d94c32218539e7e47d4ba9c3c7b77d61332fb60d633dde89e473fb`

`split_part(string, separator, index)`

Description	Splits the `string` along the `separator` and returns the data at the (1-based) `index` of the list. If the `index` is outside the bounds of the list, return an empty string (to match PostgreSQL's behavior).
Example	`split_part('a;b;c', ';', 2)`
Result	b

`starts_with(string, search_string)`

Description	Returns `true` if `string` begins with `search_string`.
Example	`starts_with('abc', 'a')`
Result	`true`
Alias	`^@`

`string_split(string, separator)`

Description	Splits the `string` along the `separator`.
Example	`string_split('hello-world', '-')`
Result	`[hello, world]`
Aliases	`split`, `str_split`, `string_to_array`

`string_split_regex(string, regex[, options])`

Description	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
Example	`string_split_regex('hello world; 42', ';? ')`
Result	`[hello, world, 42]`
Aliases	`regexp_split_to_array`, `str_split_regex`

`strip_accents(string)`

Description	Strips accents from `string`. Only combining diacritical marks are removed. Characters that are distinct letters in their own right rather than accented variants, such as ø, ł, and ß, are left unchanged (e.g., `strip_accents('großmøller')` returns `großmøller`).
Example	`strip_accents('mühleisen')`
Result	`muhleisen`

`strlen(string)`

Description	Number of bytes in `string`.
Example	`strlen('🦆')`
Result	4

`substring(string, start[, length])`

Description	Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.
Example 1	`substring('Hello', 2)`
Result	`ello`
Example 2	`substring('Hello', 2, 2)`
Result	`el`
Alias	`substr`

`substring_grapheme(string, start[, length])`

Description	Extracts substring starting from grapheme clusters `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` grapheme clusters instead. Note that a `start` value of `1` refers to the `first` character of the `string`.
Example 1	`substring_grapheme('🦆👨🏽‍🦰👨🏾‍🦰👩🏽‍🦰🪿', 3)`
Result	👨🏾‍🦰👩🏽‍🦰🪿
Example 2	`substring_grapheme('🦆👨🏽‍🦰👨🏾‍🦰👩🏽‍🦰🪿', 3, 2)`
Result	👨🏾‍🦰👩🏽‍🦰🪿

`suffix(string, search_string)`

Description	Returns `true` if `string` ends with `search_string`. Note that collations are not supported.
Example	`suffix('abc', 'bc')`
Result	`true`
Alias	`ends_with`

`to_base(number, radix[, min_length])`

Description	Converts `number` to a string in the given base `radix`, optionally padding with leading zeros to `min_length`.
Example	`to_base(42, 16, 5)`
Result	`0002A`

`to_base64(blob)`

Description	Converts a `blob` to a base64 encoded string.
Example	`to_base64('A'::BLOB)`
Result	`QQ==`
Alias	`base64`

translate(string, from, to)

Description	Replaces each character in `string` that matches a character in the `from` set with the corresponding character in the `to` set. If `from` is longer than `to`, occurrences of the extra characters in `from` are deleted.
Example	`translate('12345', '143', 'ax')`
Result	`a2x5`

trim(string[, characters])

Description	Removes any occurrences of any of the `characters` from either side of the `string`. `characters` defaults to `space`.
Example 1	trim(' test ')
Result	`test`
Example 2	`trim('>>>>test<<', '><')`
Result	`test`

unbin(value)

Description	Converts a `value` from binary representation to a blob.
Example	`unbin('0110')`
Result	`\x06`
Alias	`from_binary`

unhex(value)

Description	Converts a `value` from hexadecimal representation to a blob.
Example	`unhex('2A')`
Result	`*`
Alias	`from_hex`

unicode(string)

Description	Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.
Example	`[unicode('âbcd'), unicode('â'), unicode(''), unicode(NULL)]`
Result	`[226, 226, -1, NULL]`
Alias	`ord`

upper(string)

Description	Converts `string` to upper case.
Example	`upper('Hello')`
Result	`HELLO`
Alias	`ucase`

`url_decode(string)`

Description	Decodes a URL from a representation using Percent-Encoding.
Example	`url_decode('https%3A%2F%2Fduckdb.org%2Fwhy_duckdb%23portable')`
Result	`https://duckdb.org/why_duckdb#portable`

`url_encode(string)`

Description	Encodes a URL to a representation using Percent-Encoding.
Example	`url_encode('this string has/ special+ characters>')`
Result	`this%20string%20has%2F%20special%2B%20characters%3E`

`overlay`

> New The `overlay` function was introduced in DuckDB 2.0.

The `overlay` function replaces a substring of a string with a replacement string. It supports the SQL-standard syntax `overlay(<string> PLACING <replacement> FROM <start> [FOR <count>])`, where `start` is the (1-based) position of the first character to replace and `count` is the number of characters to replace. If `count` is omitted, it defaults to the length of `replacement`:

```
SELECT overlay('Txxxxas' PLACING 'hom' FROM 2 FOR 4) AS result;
```

result varchar
Thomas

The same can be written using the regular function call syntax `overlay(<string>, <replacement>, <start>[, <count>])`:

```
SELECT
    overlay('Txxxxas', 'hom', 2, 4) AS a,
    overlay('hello' PLACING 'xyz' FROM 2) AS b,
    overlay('abc' PLACING 'XY' FROM 2 FOR 0) AS c;
```

a varchar	b varchar	c varchar
Thomas	hxyzo	aXYbc

Text Similarity Functions

These functions are used to measure the similarity of two strings using various similarity measures.

Function	Description
`damerau_levenshtein(s1, s2)`	Extension of Levenshtein distance to also include transposition of adjacent characters as an allowed edit operation. In other words, the minimum number of edit operations (insertions, deletions, substitutions or transpositions) required to change one string to another. Characters of different cases (e.g., a and A) are considered different.
`editdist3(s1, s2)`	Alias for `levenshtein`.
`hamming(s1, s2)`	The Hamming distance between two strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
`jaccard(s1, s2)`	The Jaccard similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
`jaro_similarity(s1, s2[, score_cutoff])`	The Jaro similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
`jaro_winkler_similarity(s1, s2[, score_cutoff])`	The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
`levenshtein(s1, s2)`	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
`mismatches(s1, s2)`	Alias for `hamming`.

`damerau_levenshtein(s1, s2)`

Description	Extension of Levenshtein distance to also include transposition of adjacent characters as an allowed edit operation. In other words, the minimum number of edit operations (insertions, deletions, substitutions or transpositions) required to change one string to another. Characters of different cases (e.g., a and A) are considered different.
Example	`damerau_levenshtein('duckdb', 'udckbd')`
Result	2

`hamming(s1, s2)`

Description	The Hamming distance between two strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
Example	`hamming('duck', 'luck')`
Result	1
Alias	`mismatches`

`jaccard(s1, s2)`

Description	The Jaccard similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
Example	`jaccard('duck', 'luck')`
Result	`0.6`

`jaro_similarity(s1, s2[, score_cutoff])`

Description	The Jaro similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
Example	`jaro_similarity('duck', 'duckdb')`
Result	`0.8888888888888888`

`jaro_winkler_similarity(s1, s2[, score_cutoff])`

Description	The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
Example	`jaro_winkler_similarity('duck', 'duckdb')`
Result	`0.9333333333333333`

`levenshtein(s1, s2)`

Description	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
Example	`levenshtein('duck', 'db')`
Result	3
Alias	`editdist3`

Formatters

`fmt` Syntax

The `format(format, parameters...)` function formats strings, loosely following the syntax of the {fmt} open-source formatting library.

Format without additional parameters:

```
SELECT format('Hello world'); -- Hello world
```

Format a string using {}:

```
SELECT format('The answer is {}', 42); -- The answer is 42
```

Format a string using positional arguments:

```
SELECT format('I''d rather be {1} than {0}.', 'right', 'happy'); -- I'd rather be happy than right.
```

Format Specifiers

Specifier	Description	Example
{:d}	integer	654321
{:E}	scientific notation	3.141593E+00
{:f}	float	4.560000
{:o}	octal	2375761
{:s}	string	asd
{:x}	hexadecimal	9fbf1
{:tX}	integer, X is the thousand separator	654 321

Formatting Types Integers:

```
SELECT format('{} + {} = {}', 3, 5, 3 + 5); -- 3 + 5 = 8
```

Booleans:

```
SELECT format('{} != {}', true, false); -- true != false
```

Format datetime values:

```
SELECT format('{}', DATE '1992-01-01'); -- 1992-01-01
SELECT format('{}', TIME '12:01:00'); -- 12:01:00
SELECT format('{}', TIMESTAMP '1992-01-01 12:01:00'); -- 1992-01-01 12:01:00
```

Format BLOB:

```
SELECT format('{}', BLOB '\x00hello'); -- \x00hello
```

Pad integers with 0s:

```
SELECT format('{:04d}', 33); -- 0033
```

> Padding cannot currently be combined with the specification of a thousands separator.

Create timestamps from integers:

```
SELECT format('{:02d}:{:02d}:{:02d} {}', 12, 3, 16, 'AM'); -- 12:03:16 AM
```

Convert to hexadecimal:

```
SELECT format('{:x}', 123_456_789); -- 75bcd15
```

Convert to binary:

```
SELECT format('{:b}', 123_456_789); -- 111010110111100110100010101
```

Print Numbers with Thousand Separators Integers:

```
SELECT format('{:,}',  123_456_789); -- 123,456,789
SELECT format('{:t.}', 123_456_789); -- 123.456.789
SELECT format('{:''}', 123_456_789); -- 123'456'789
SELECT format('{:_}',  123_456_789); -- 123_456_789
SELECT format('{:t }', 123_456_789); -- 123 456 789
SELECT format('{:tX}', 123_456_789); -- 123X456X789
```

Float, double and decimal:

```
SELECT format('{:,f}',    123456.789); -- 123,456.789000
SELECT format('{:,.2f}',  123456.789); -- 123,456.79
SELECT format('{:t..2f}', 123456.789); -- 123.456,79
```

`printf` Syntax

The `printf(format, parameters...)` function formats strings using the `printf` syntax.

Format without additional parameters:

```
SELECT printf('Hello world');
```

```
Hello world
```

Format a string using arguments in a given order:

```
SELECT printf('The answer to %s is %d', 'life', 42);
```

```
The answer to life is 42
```

Format a string using positional arguments `%position$formatter`, e.g., the second parameter as a string is encoded as `%2$s`:

```
SELECT printf('I''d rather be %2$s than %1$s.', 'right', 'happy');
```

```
I'd rather be happy than right.
```

Format Specifiers

Specifier	Description	Example
%c	character code to character	a
%d	integer	654321
%Xd	integer with thousand separator X from `,`, `.`, `''`, `_`	654_321
%E	scientific notation	3.141593E+00
%f	float	4.560000
%hd	integer	654321
%hhd	integer	654321
%lld	integer	654321
%o	octal	2375761
%s	string	asd
%x	hexadecimal	9fbf1

Formatting Types Integers:

```
SELECT printf('%d + %d = %d', 3, 5, 3 + 5); -- 3 + 5 = 8
```

Booleans:

```
SELECT printf('%s != %s', true, false); -- true != false
```

Format datetime values:

```
SELECT printf('%s', DATE '1992-01-01'); -- 1992-01-01
SELECT printf('%s', TIME '12:01:00'); -- 12:01:00
SELECT printf('%s', TIMESTAMP '1992-01-01 12:01:00'); -- 1992-01-01 12:01:00
```

Format BLOB:

```
SELECT printf('%s', BLOB '\x00hello'); -- \x00hello
```

Pad integers with 0s:

```
SELECT printf('%04d', 33); -- 0033
```

Create timestamps from integers:

```
SELECT printf('%02d:%02d:%02d %s', 12, 3, 16, 'AM'); -- 12:03:16 AM
```

Convert to hexadecimal:

```
SELECT printf('%x', 123_456_789); -- 75bcd15
```

Convert to binary:

```
SELECT printf('%b', 123_456_789); -- 111010110111100110100010101
```

Thousand Separators Integers:

```
SELECT printf('%,d',  123_456_789); -- 123,456,789
SELECT printf('%.d',  123_456_789); -- 123.456.789
SELECT printf('%''d', 123_456_789); -- 123'456'789
SELECT printf('%_d',  123_456_789); -- 123_456_789
```

Float, double and decimal:

```
SELECT printf('%,f',   123456.789); -- 123,456.789000
SELECT printf('%,.2f', 123456.789); -- 123,456.79
```

Time Functions

This section describes functions and operators for examining and manipulating `TIME` values.

Time Operators

The table below shows the available mathematical operators for `TIME` types.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIME '01:02:03' + INTERVAL 5 HOUR`	`06:02:03`
`-`	subtraction of an `INTERVAL`	`TIME '06:02:03' - INTERVAL 5 HOUR`	`01:02:03`

Time Functions

The table below shows the available scalar functions for `TIME` types.

Name	Description
`current_localtime()`	Current time (start of current transaction) in the local time zone, as a `TIME` value without a time zone.
`date_diff(part, starttime, endtime)`	The number of `part` boundaries between `starttime` and `endtime`, inclusive of the larger time and exclusive of the smaller time.
`date_part(part, time)`	Get subfield (equivalent to `extract`).

Name	Description
`date_sub(part, starttime, endtime)`	The signed length of the interval between `starttime` and `endtime`, truncated to whole multiples of `part`.
`extract(part FROM time)`	Get subfield from a time.
`get_current_time()`	Current time (start of current transaction).
`make_time(bigint, bigint, double)`	The time for the given parts.

The only date parts that are defined for times are `epoch`, `hours`, `minutes`, `seconds`, `milliseconds` and `microseconds`.

`current_localtime()`

Description	Current time (start of current transaction) in the local time zone, as a `TIME` value without a time zone.
Example	`current_localtime()`
Result	`06:09:59.988`
Alias	`localtime` (no parentheses necessary)

`date_diff(part, starttime, endtime)`

Description	The number of `part` boundaries between `starttime` and `endtime`, inclusive of the larger time and exclusive of the smaller time.
Example	`date_diff('hour', TIME '01:02:03', TIME '06:01:03')`
Result	`5`
Alias	`datediff`

`date_part(part, time)`

Description	Get subfield (equivalent to `extract`).
Example	`date_part('minute', TIME '14:21:13')`
Result	`21`
Alias	`datepart`

`date_sub(part, starttime, endtime)`

Description	The signed length of the interval between `starttime` and `endtime`, truncated to whole multiples of `part`.
Example	`date_sub('hour', TIME '01:02:03', TIME '06:01:03')`
Result	`4`
Alias	`datesub`

`extract(part FROM time)`

Description	Get subfield from a time.
Example	`extract('hour' FROM TIME '14:21:13')`
Result	`14`

`get_current_time()`

Description	Current time (start of current transaction) in the local time zone as `TIMETZ`.
Example	`get_current_time()`
Result	`06:09:59.988+2`
Alias	`current_time` (no parentheses necessary)

`make_time(bigint, bigint, double)`

Description	The time for the given parts.
Example	`make_time(13, 34, 27.123456)`
Result	`13:34:27.123456`

Timestamp Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP` values. See also the related `TIMESTAMPTZ` functions.

Timestamp Operators

The table below shows the available mathematical operators for `TIMESTAMP` types.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`-`	subtraction of `TIMESTAMP`s	`TIMESTAMP '1992-03-27' - TIMESTAMP '1992-03-22'`	`5 days`
`-`	subtraction of an `INTERVAL`	`TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`

Adding to or subtracting from infinite values produces the same infinite value.

Scalar Timestamp Functions

The table below shows the available scalar functions for `TIMESTAMP` values.

Name	Description
age(timestamp, timestamp)	Subtract arguments, resulting in the time difference between the two timestamps.
age(timestamp)	Subtract from current_date.
ago(interval)	Subtracts an interval from the current timestamp.
century(timestamp)	Extracts the century of a timestamp.
current_localtimestamp()	Returns the current timestamp (at the start of the transaction).
date_diff(part, starttimestamp, endtimestamp)	The number of part boundaries between starttimestamp and endtimestamp, inclusive of the larger timestamp and exclusive of the smaller timestamp.
date_part([part, ...], timestamp)	Get the listed subfields as a struct. The list must be constant.
date_part(part, timestamp)	Get subfield (equivalent to extract).
date_sub(part, starttimestamp, endtimestamp)	The signed length of the interval between starttimestamp and endtimestamp, truncated to whole multiples of part.
date_trunc(part, timestamp)	Truncate to specified precision.
dayname(timestamp)	The (English) name of the weekday.
epoch_ms(timestamp)	Returns the total number of milliseconds since the epoch.
epoch_ns(timestamp)	Returns the total number of nanoseconds since the epoch.
epoch_us(timestamp)	Returns the total number of microseconds since the epoch.
epoch(timestamp)	Returns the total number of seconds since the epoch.
extract(field FROM timestamp)	Get subfield from a timestamp.
greatest(timestamp, timestamp)	The later of two timestamps.
isfinite(timestamp)	Returns true if the timestamp is finite, false otherwise.
isinf(timestamp)	Returns true if the timestamp is infinite, false otherwise.
julian(timestamp)	Extract the Julian Day number from a timestamp.
last_day(timestamp)	The last day of the month.
least(timestamp, timestamp)	The earlier of two timestamps.
make_timestamp(bigint, bigint, bigint, bigint, bigint, double)	The timestamp for the given parts.
make_timestamp(microseconds)	Converts microseconds since the epoch to a timestamp.
make_timestamp_ms(milliseconds)	Converts milliseconds since the epoch to a timestamp.
make_timestamp_ns(nanoseconds)	Converts nanoseconds since the epoch to a timestamp.
monthname(timestamp)	The (English) name of the month.

Name	Description
`strftime(timestamp, format)`	Converts timestamp to string according to the format string.
`strptime(text, format-list)`	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
`strptime(text, format)`	Converts the string `text` to timestamp according to the format string. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
`time_bucket(bucket_width, timestamp[, offset])`	Truncate `timestamp` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01 00:00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00[ + offset]`. Note that `2000-01-03` is a Monday.
`time_bucket(bucket_width, timestamp[, origin])`	Truncate `timestamp` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00`. Note that `2000-01-03` is a Monday.
`try_strptime(text, format-list)`	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Returns `NULL` on failure.
`try_strptime(text, format)`	Converts the string `text` to timestamp according to the format string. Returns `NULL` on failure.

There are also dedicated extraction functions to get the subfields.

Functions applied to infinite dates will either return the same infinite dates (e.g., `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite date, the result will be `NULL`.

`age(timestamp, timestamp)`

Description	Subtract arguments, resulting in the time difference between the two timestamps.
Example	`age(TIMESTAMP '2001-04-10', TIMESTAMP '1992-09-20')`
Result	`8 years 6 months 20 days`

`age(timestamp)`

Description	Subtract from current_date.
Example	`age(TIMESTAMP '1992-09-20')`
Result	`29 years 1 month 27 days 12:39:00.844`

`ago(interval)`

Description	Subtracts an interval from the current timestamp, returning a timestamp in the past. Equivalent to `current_timestamp - interval`.
Example	`ago(INTERVAL 1 HOUR)`
Result	`2024-11-30 12:28:48.895` (if current time is `2024-11-30 13:28:48.895`)

`century(timestamp)`

Description	Extracts the century of a timestamp.
Example	`century(TIMESTAMP '1992-03-22')`
Result	`20`

`current_localtimestamp()`

Description	Returns the current timestamp with time zone (at the start of the transaction).
Example	`current_localtimestamp()`
Result	`2024-11-30 13:28:48.895`

`date_diff(part, starttimestamp, endtimestamp)`

Description	The signed number of `part` boundaries between `starttimestamp` and `endtimestamp`, inclusive of the larger timestamp and exclusive of the smaller timestamp.
Example	`date_diff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`2`

`date_part([part, ...], timestamp)`

Description	Get the listed subfields as a `struct`. The list must be constant.
Example	`date_part(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`
Result	`{year: 1992, month: 9, day: 20}`

`date_part(part, timestamp)`

Description	Get subfield (equivalent to `extract`).
Example	`date_part('minute', TIMESTAMP '1992-09-20 20:38:40')`
Result	`38`

`date_sub(part, starttimestamp, endtimestamp)`

Description	The signed length of the interval between `starttimestamp` and `endtimestamp`, truncated to whole multiples of `part`.
Example	`date_sub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`1`

`date_trunc(part, timestamp)`

Description	Truncate to specified precision.
Example	`date_trunc('hour', TIMESTAMP '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`dayname(timestamp)`

Description	The (English) name of the weekday.
Example	`dayname(TIMESTAMP '1992-03-22')`
Result	Sunday

`epoch_ms(timestamp)`

Description	Returns the total number of milliseconds since the epoch.
Example	`epoch_ms(TIMESTAMP '2021-08-03 11:59:44.123456')`
Result	`1627991984123`

`epoch_ns(timestamp)`

Description	Returns the total number of nanoseconds since the epoch.
Example	`epoch_ns(TIMESTAMP '2021-08-03 11:59:44.123456')`
Result	`1627991984123456000`

`epoch_us(timestamp)`

Description	Returns the total number of microseconds since the epoch.
Example	`epoch_us(TIMESTAMP '2021-08-03 11:59:44.123456')`
Result	`1627991984123456`

`epoch(timestamp)`

Description	Returns the total number of seconds since the epoch.
Example	`epoch('2022-11-07 08:43:04'::TIMESTAMP)`
Result	`1667810584.0`

> Unlike `epoch_ms`, `epoch_us` and `epoch_ns`, which return `BIGINT`, `epoch` returns a `DOUBLE` and therefore retains sub-second precision.

`extract(field FROM timestamp)`

Description	Get subfield from a timestamp.
Example	`extract('hour' FROM TIMESTAMP '1992-09-20 20:38:48')`
Result	`20`

`greatest(timestamp, timestamp)`

Description	The later of two timestamps.
Example	`greatest(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-09-20 20:38:48`

`isfinite(timestamp)`

Description	Returns true if the timestamp is finite, false otherwise.
Example	`isfinite(TIMESTAMP '1992-03-07')`
Result	`true`

`isinf(timestamp)`

Description	Returns true if the timestamp is infinite, false otherwise.
Example	`isinf(TIMESTAMP '-infinity')`
Result	`true`

`julian(timestamp)`

Description	Extract the Julian Day number from a timestamp.
Example	`julian(TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`2448704.043091706`

`last_day(timestamp)`

Description	The last day of the month.
Example	`last_day(TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-03-31`

least(timestamp, timestamp)

Description	The earlier of two timestamps.
Example	`least(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-03-22 01:02:03.1234`

make_timestamp(bigint, bigint, bigint, bigint, bigint, double)

Description	The timestamp for the given parts.
Example	`make_timestamp(1992, 9, 20, 13, 34, 27.123456)`
Result	`1992-09-20 13:34:27.123456`

make_timestamp(microseconds)

Description	Converts microseconds since the epoch to a timestamp.
Example	`make_timestamp(1667810584123456)`
Result	`2022-11-07 08:43:04.123456`

make_timestamp_ms(milliseconds)

Description	Converts milliseconds since the epoch to a timestamp.
Example	`make_timestamp_ms(1667810584123)`
Result	`2022-11-07 08:43:04.123`

make_timestamp_ns(nanoseconds)

Description	Converts nanoseconds since the epoch to a timestamp.
Example	`make_timestamp_ns(1667810584123456789)`
Result	`2022-11-07 08:43:04.123456789`

monthname(timestamp)

Description	The (English) name of the month.
Example	`monthname(TIMESTAMP '1992-09-20')`
Result	`September`

`strftime(timestamp, format)`

Description	Converts timestamp to string according to the format string.
Example	`strftime(timestamp '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`Wed, 1 January 1992 - 08:38:40 PM`

`strptime(text, format-list)`

Description	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
Example	`strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`
Result	`2023-04-15 10:56:00`

`strptime(text, format)`

Description	Converts the string `text` to timestamp according to the format string. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
Example	`strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`1992-01-01 20:38:40`

`time_bucket(bucket_width, timestamp[, offset])`

Description	Truncate `timestamp` to a grid of width `bucket_width`. The grid includes `2000-01-01 00:00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00[ + offset]`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '10 minutes', TIMESTAMP '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`
Result	`1992-04-20 15:25:00`

`time_bucket(bucket_width, timestamp[, origin])`

Description	Truncate `timestamp` to a grid of width `bucket_width`. The grid includes the `origin` timestamp, which defaults to `2000-01-01 00:00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 weeks', TIMESTAMP '1992-04-20 15:26:00', TIMESTAMP '1992-04-01 00:00:00')`
Result	`1992-04-15 00:00:00`

`try_strptime(text, format-list)`

Description	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Returns `NULL` on failure.
Example	`try_strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`
Result	`2023-04-15 10:56:00`

`try_strptime(text, format)`

Description	Converts the string `text` to timestamp according to the format string. Returns `NULL` on failure.
Example	`try_strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`1992-01-01 20:38:40`

Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP` types.

Name	Description
`generate_series(timestamp, timestamp, interval)`	Generate a table of timestamps in the closed range, stepping by the interval.
`range(timestamp, timestamp, interval)`	Generate a table of timestamps in the half open range, stepping by the interval.

> Infinite values are not allowed as table function bounds.

`generate_series(timestamp, timestamp, interval)`

Description	Generate a table of timestamps in the closed range, stepping by the interval.
Example	`generate_series(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

`range(timestamp, timestamp, interval)`

Description	Generate a table of timestamps in the half open range, stepping by the interval.
Example	`range(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

Timestamp with Time Zone Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP WITH TIME ZONE` (or `TIMESTAMPTZ`) values. See also the related `TIMESTAMP` functions.

Time zone support is provided by the built-in ICU extension.

In the examples below, the current time zone is presumed to be `America/Los_Angeles` using the Gregorian calendar.

Built-In Timestamp with Time Zone Functions

The table below shows the available scalar functions for `TIMESTAMPTZ` values. Since these functions do not involve binning or display, they are always available.

Name	Description
`current_timestamp`	Current date and time (start of current transaction).
`get_current_timestamp()`	Current date and time (start of current transaction).
`greatest(timestamptz, timestamptz)`	The later of two timestamps.
`isfinite(timestamptz)`	Returns true if the timestamp with time zone is finite, false otherwise.
`isinf(timestamptz)`	Returns true if the timestamp with time zone is infinite, false otherwise.
`least(timestamptz, timestamptz)`	The earlier of two timestamps.
`now()`	Current date and time (start of current transaction).
`timetz_byte_comparable(timetz)`	Converts a `TIME WITH TIME ZONE` to a `UBIGINT` sort key.
`to_timestamp(double)`	Converts seconds since the epoch to a timestamp with time zone.
`transaction_timestamp()`	Current date and time (start of current transaction).

`current_timestamp`

Description	Current date and time (start of current transaction).
Example	`current_timestamp`
Result	`2022-10-08 12:44:46.122-07`

`get_current_timestamp()`

Description	Current date and time (start of current transaction).
Example	`get_current_timestamp()`
Result	`2022-10-08 12:44:46.122-07`

`greatest(timestamptz, timestamptz)`

Description	The later of two timestamps.
Example	`greatest(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-09-20 20:38:48-07`

`isfinite(timestamptz)`

Description	Returns true if the timestamp with time zone is finite, false otherwise.
Example	`isfinite(TIMESTAMPTZ '1992-03-07')`
Result	`true`

`isinf(timestamptz)`

Description	Returns true if the timestamp with time zone is infinite, false otherwise.
Example	`isinf(TIMESTAMPTZ '-infinity')`
Result	`true`

`least(timestamptz, timestamptz)`

Description	The earlier of two timestamps.
Example	`least(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-03-22 01:02:03.1234-08`

`now()`

Description	Current date and time (start of current transaction).
Example	`now()`
Result	`2022-10-08 12:44:46.122-07`

`timetz_byte_comparable(timetz)`

Description	Converts a `TIME WITH TIME ZONE` to a `UBIGINT` sort key.
Example	`timetz_byte_comparable('18:18:16.21-07:00'::TIMETZ)`
Result	`2494691656335442799`

`to_timestamp(double)`

Description	Converts seconds since the epoch to a timestamp with time zone.

Example	`to_timestamp(1284352323.5)`
Result	`2010-09-12 21:32:03.5-07`

`transaction_timestamp()`

Description	Current date and time (start of current transaction).
Example	`transaction_timestamp()`
Result	`2022-10-08 12:44:46.122-07`

Timestamp with Time Zone Strings

With no time zone extension loaded, `TIMESTAMPTZ` values will be cast to and from strings using offset notation. This will let you specify an instant correctly without access to time zone information. For portability, `TIMESTAMPTZ` values will always be displayed using GMT offsets:

```
SELECT '2022-10-08 13:13:34-07'::TIMESTAMPTZ;
```

```
2022-10-08 20:13:34+00
```

If a time zone extension such as ICU is loaded, then a time zone can be parsed from a string and cast to a representation in the local time zone:

```
SELECT '2022-10-08 13:13:34 Europe/Amsterdam'::TIMESTAMPTZ::VARCHAR;
```

```
2022-10-08 04:13:34-07 -- the offset will differ based on your local time zone
```

ICU Timestamp with Time Zone Operators

The table below shows the available mathematical operators for `TIMESTAMP WITH TIME ZONE` values provided by the ICU extension.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIMESTAMPTZ '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`-`	subtraction of `TIMESTAMPTZ`s	`TIMESTAMPTZ '1992-03-27' - TIMESTAMPTZ '1992-03-22'`	`5 days`
`-`	subtraction of an `INTERVAL`	`TIMESTAMPTZ '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`

Adding to or subtracting from infinite values produces the same infinite value.

Addition and subtraction of intervals follows the semantics of the ICU Calendar add function, which the `icu` extension implements natively since DuckDB 2.0. For positive intervals (forwards in time) the fields are incremented from most to least significant. For negative intervals (backwards in time) the fields are decremented from least to most significant. This produces the same results as Postgres, but does not match some more recent calendar RFCs.

ICU Timestamp with Time Zone Functions

The table below shows the ICU provided scalar functions for `TIMESTAMP WITH TIME ZONE` values.

Name	Description
`age(timestamptz, timestamptz)`	Subtract arguments, resulting in the time difference between the two timestamps.
`age(timestamptz)`	Subtract from current_date.
`date_diff(part, starttimestamptz, endtimestamptz)`	The number of `part` boundaries between `starttimestamptz` and `endtimestamptz` inclusive of the larger timestamp and exclusive of the smaller timestamp.
`date_part([part, ...], timestamp)`	Get the listed subfields as a `struct`. The list must be constant.
`date_part(part, timestamp)`	Get subfield (equivalent to `extract`).
`date_sub(part, starttimestamptz, endtimestamptz)`	The signed length of the interval between `starttimestamptz` and `endtimestamptz`, truncated to whole multiples of `part`.
`date_trunc(part, timestamptz)`	Truncate to specified precision.
`epoch_ns(timestamptz)`	Converts a timestamptz to nanoseconds since the epoch.
`epoch_us(timestamptz)`	Converts a timestamptz to microseconds since the epoch.
`extract(field FROM timestamptz)`	Get subfield from a `TIMESTAMP WITH TIME ZONE`.
`last_day(timestamptz)`	The last day of the month.
`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)`	The `TIMESTAMP WITH TIME ZONE` for the given parts and time zone.
`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)`	The `TIMESTAMP WITH TIME ZONE` for the given parts in the current time zone.
`make_timestamptz(microseconds)`	The `TIMESTAMP WITH TIME ZONE` for the given µs since the epoch.
`strftime(timestamptz, format)`	Converts a `TIMESTAMP WITH TIME ZONE` value to string according to the format string.
`strptime(text, format)`	Converts string to `TIMESTAMP WITH TIME ZONE` according to the format string if `%Z` is specified.
`time_bucket(bucket_width, timestamptz[, offset])`	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01 00:00:00+00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00[ + offset]`. Note that `2000-01-03` is a Monday.
`time_bucket(bucket_width, timestamptz[, origin])`	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00+00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00`. Note that `2000-01-03` is a Monday.

Name	Description
time_bucket(bucket_width, timestamptz[, timezone])	Truncate timestamptz to a grid of width bucket_width. The grid is anchored at the origin timestamp, which defaults to 2000-01-01 00:00:00 in the provided timezone when bucket_width is a number of months or coarser units, else 2000-01-03 00:00:00 in the provided timezone. The default timezone is 'UTC'. Note that 2000-01-03 is a Monday.

age(timestamptz, timestamptz)

Description	Subtract arguments, resulting in the time difference between the two timestamps.
Example	age(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '1992-09-20')
Result	8 years 6 months 20 days

age(timestamptz)

Description	Subtract from current_date.
Example	age(TIMESTAMP '1992-09-20')
Result	29 years 1 month 27 days 12:39:00.844

date_diff(part, starttimestamptz, endtimestamptz)

Description	The signed number of part boundaries between starttimestamptz and endtimestamptz, inclusive of the larger timestamp and exclusive of the smaller timestamp.
Example	date_diff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')
Result	2

date_part([part, ...], timestamptz)

Description	Get the listed subfields as a struct. The list must be constant.
Example	date_part(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')
Result	{year: 1992, month: 9, day: 20}

date_part(part, timestamptz)

Description	Get subfield (equivalent to *extract*).
Example	date_part('minute', TIMESTAMPTZ '1992-09-20 20:38:40')
Result	38

`date_sub(part, starttimestamptz, endtimestamptz)`

Description	The signed length of the interval between `starttimestamptz` and `endtimestamptz`, truncated to whole multiples of `part`.
Example	`date_sub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	`1`

`date_trunc(part, timestamptz)`

Description	Truncate to specified precision.
Example	`date_trunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`epoch_ns(timestamptz)`

Description	Converts a timestamptz to nanoseconds since the epoch.
Example	`epoch_ns('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123456000`

`epoch_us(timestamptz)`

Description	Converts a timestamptz to microseconds since the epoch.
Example	`epoch_us('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123456`

`extract(field FROM timestamptz)`

Description	Get subfield from a `TIMESTAMP WITH TIME ZONE`.
Example	`extract('hour' FROM TIMESTAMPTZ '1992-09-20 20:38:48')`
Result	`20`

`last_day(timestamptz)`

Description	The last day of the month.
Example	`last_day(TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-03-31`

`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given parts and time zone.
Example	`make_timestamptz(1992, 9, 20, 15, 34, 27.123456, 'CET')`
Result	`1992-09-20 06:34:27.123456-07`

`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given parts in the current time zone.
Example	`make_timestamptz(1992, 9, 20, 13, 34, 27.123456)`
Result	`1992-09-20 13:34:27.123456-07`

`make_timestamptz(microseconds)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given µs since the epoch.
Example	`make_timestamptz(1667810584123456)`
Result	`2022-11-07 00:43:04.123456-08`

`strftime(timestamptz, format)`

Description	Converts a `TIMESTAMP WITH TIME ZONE` value to string according to the format string.
Example	`strftime(timestamptz '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`Wed, 1 January 1992 - 08:38:40 PM`

`strptime(text, format)`

Description	Converts string to `TIMESTAMP WITH TIME ZONE` according to the format string if `%Z` is specified.
Example	`strptime('Wed, 1 January 1992 - 08:38:40 PST', '%a, %-d %B %Y - %H:%M:%S %Z')`
Result	`1992-01-01 08:38:40-08`

`time_bucket(bucket_width, timestamptz[, offset])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01 00:00:00+00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00[ + offset]`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '10 minutes', TIMESTAMPTZ '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`
Result	`1992-04-20 15:25:00-07`

`time_bucket(bucket_width, timestamptz[, origin])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00+00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 weeks', TIMESTAMPTZ '1992-04-20 15:26:00-07', TIMESTAMPTZ '1992-04-01 00:00:00-07')`
Result	`1992-04-15 00:00:00-07`

`time_bucket(bucket_width, timestamptz[, timezone])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00` in the provided `timezone` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00` in the provided `timezone`. The default timezone is `'UTC'`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 days', TIMESTAMPTZ '1992-04-20 15:26:00-07', 'Europe/Berlin')`
Result	`1992-04-19 15:00:00-07` (=`1992-04-20 00:00:00 Europe/Berlin`)

There are also dedicated extraction functions to get the subfields.

ICU Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP WITH TIME ZONE` types.

Name	Description
`generate_series(timestamptz, timestamptz, interval)`	Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval.
`range(timestamptz, timestamptz, interval)`	Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp), stepping by the interval.

> Infinite values are not allowed as table function bounds.

`generate_series(timestamptz, timestamptz, interval)`

Description	Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval.
Example	`generate_series(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

`range(timestamptz, timestamptz, interval)`

Description	Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp), stepping by the interval.
Example	`range(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

ICU Timestamp Without Time Zone Functions

The table below shows the ICU provided scalar functions that operate on plain `TIMESTAMP` values. These functions assume that the `TIMESTAMP` is a "local timestamp".

A local timestamp is effectively a way of encoding the part values from a time zone into a single value. They should be used with caution because the produced values can contain gaps and ambiguities thanks to daylight savings time. Often the same functionality can be implemented more reliably using the `struct` variant of the `date_part` function.

Name	Description
`current_localtime()`	Returns a `TIME` whose GMT bin values correspond to local time in the current time zone.
`current_localtimestamp()`	Returns a `TIMESTAMP` whose GMT bin values correspond to local date and time in the current time zone.
`localtime`	Synonym for the `current_localtime()` function call.
`localtimestamp`	Synonym for the `current_localtimestamp()` function call.
`timezone(text, timestamp)`	Use the date parts of the timestamp in GMT to construct a timestamp in the given time zone. Effectively, the argument is a "local" time.
`timezone(text, timestamptz)`	Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a "local" time.

`current_localtime()`

Description	Returns a `TIME` whose GMT bin values correspond to local time in the current time zone.
Example	`current_localtime()`
Result	`08:47:56.497`

`current_localtimestamp()`

Description	Returns a `TIMESTAMP` whose GMT bin values correspond to local date and time in the current time zone.
Example	`current_localtimestamp()`
Result	`2022-12-17 08:47:56.497`

`localtime`

Description	Synonym for the `current_localtime()` function call.
Example	`localtime`

Result	`08:47:56.497`

`localtimestamp`

Description	Synonym for the `current_localtimestamp()` function call.
Example	`localtimestamp`
Result	`2022-12-17 08:47:56.497`

`timezone(text, timestamp)`

Description	Use the date parts of the timestamp in GMT to construct a timestamp in the given time zone. Effectively, the argument is a "local" time.
Example	`timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40')`
Result	`2001-02-16 19:38:40-08`

`timezone(text, timestamptz)`

Description	Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a "local" time.
Example	`timezone('America/Denver', TIMESTAMPTZ '2001-02-16 20:38:40-05')`
Result	`2001-02-16 18:38:40`

At Time Zone

The `AT TIME ZONE` syntax is syntactic sugar for the (two argument) `timezone` function listed above:

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver' AS ts;
```

```
2001-02-16 19:38:40-08
```

```
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver' AS ts;
```

```
2001-02-16 18:38:40
```

Note that numeric timezones are not allowed:

```
SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE '0200' AS ts;
```

```
Not implemented Error: Unknown TimeZone '0200'
```

Infinities

Functions applied to infinite dates will either return the same infinite dates (e.g., `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite temporal value, the result will be `NULL`.

Calendars

The ICU extension also supports non-Gregorian calendars. If such a calendar is current, then the display and binning operations will use that calendar.

Daylight Saving Time (DST) Transitions

When adding calendar intervals such as `INTERVAL '1 day'` to a `TIMESTAMPTZ`, the resulting local timestamp may fall on a non-existent time during daylight saving time transitions.

DuckDB follows PostgreSQL behavior and adjusts the result forward to the next valid timestamp.

Example:

```
SET timezone = 'Europe/Amsterdam';
```

```
SELECT TIMESTAMPTZ '2025-03-29 02:30:00+01' + INTERVAL '1 day';
```

```
2025-03-30 03:30:00+02
```

Union Functions

Name	Description
`union.tag`	Dot notation serves as an alias for `union_extract`.
`union_extract(union, 'tag')`	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected.
`union_value(tag := any)`	Create a single member `UNION` containing the argument value. The tag of the value will be the bound variable name.
`union_tag(union)`	Retrieve the currently selected tag of the union as an Enum.

`union.tag`

Description	Dot notation serves as an alias for `union_extract`.
Example	`(union_value(k := 'hello')).k`
Result	`hello`

`union_extract(union, 'tag')`

Description	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected.
Example	`union_extract(s, 'k')`
Result	`hello`

`union_value(tag := any)`

Description	Create a single member `UNION` containing the argument value. The tag of the value will be the bound variable name.
Example	`union_value(k := 'hello')`
Result	`hello` (of type `UNION(k VARCHAR)`)

`union_tag(union)`

Description	Retrieve the currently selected tag of the union as an Enum.
Example	`union_tag(union_value(k := 'foo'))`
Result	`'k'`

Utility Functions

Scalar Utility Functions

The functions below are difficult to categorize into specific function types and are broadly useful.

Name	Description
`alias(column)`	Return the name of the column.
`can_cast_implicitly(source_value, target_value)`	Whether or not we can implicitly cast from the types of the source value to the target value.
`checkpoint(database)`	Synchronize WAL with file for (optional) database without interrupting transactions.
`coalesce(expr, ...)`	Return the first expression that evaluates to a non-`NULL` value. Accepts 1 or more parameters. Each expression can be a column, literal value, function result, or many others.
`constant_or_null(arg1, arg2)`	If `arg2` is `NULL`, return `NULL`. Otherwise, return `arg1`.
`count_if(x)`	Aggregate function; returns the number of rows where the `BOOLEAN` argument `x` is `true`.
`create_sort_key(parameters...)`	Constructs a binary-comparable sort key based on a set of input parameters and sort qualifiers.
`current_catalog()`	Return the name of the currently active catalog. Default is memory.
`current_database()`	Return the name of the currently active database.
`current_query()`	Return the current query as a string.
`current_query_id()`	Returns the identifier of the current query as a `UBIGINT`.
`current_schema()`	Return the name of the currently active schema. Default is main.
`current_schemas(boolean)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.
`current_setting('setting_name')`	Return the current value of the configuration setting.
`currval('sequence_name')`	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.
`duckdb_format_sql(sql[, config])`	Formats the `sql` string, e.g., by placing each clause on a separate line and normalizing the case of keywords. The optional `config` is a `MAP(VARCHAR, VARCHAR)` with the keys `'indent_size'`, `'inline_threshold'` and `'keyword_case'` (`'upper'`, `'lower'` or `'preserve'`). Provided by the `autocomplete` extension.
`error(message)`	Throws the given error `message`.
`equi_width_bins(min, max, bincount, nice := false)`	Returns the upper boundaries of a partition of the interval `[min, max]` into `bin_count` equal-sized subintervals (for use with, e.g., `histogram`). If `nice = true`, then `min`, `max` and `bincount` may be adjusted to produce more aesthetically pleasing results.

Name	Description
`force_checkpoint(database)`	Synchronize WAL with file for (optional) database interrupting transactions.
`gen_random_uuid()`	Return a random UUID similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`get_block_size(database)`	Returns the block size of the attached database with the given name.
`get_type(expression)`	Returns the data type of the result of the `expression` as a value of type `TYPE`.
`getenv(var)`	Returns the value of the environment variable `var`. Only available in the command line client.
`getvariable('variable_name')`	Returns the value of the SQL variable named `variable_name`, or `NULL` if it is not set.
`hash(value)`	Returns a `UBIGINT` with a hash of `value`. The used hash function may change across DuckDB versions.
`icu_sort_key(string, collator)`	Surrogate sort key used to sort special characters according to the specific locale. Collator parameter is optional. Only available when the ICU extension is installed.
`if(a, b, c)`	Ternary conditional operator.
`ifnull(expr, other)`	A two-argument version of coalesce.
`in_search_path(database_name, schema_name)`	Returns whether or not the given database and schema are in the search path.
`is_histogram_other_bin(arg)`	Returns `true` when `arg` is the "catch-all element" of its datatype for the purpose of the `histogram_exact` function, which is equal to the "right-most boundary" of its datatype for the purpose of the `histogram` function.
`make_type(name, ...)`	Constructs a value of type `TYPE` from a type name and optional type parameters.
`md5(string)`	Returns the MD5 hash of the `string` as a `VARCHAR`.
`md5_number(string)`	Returns the MD5 hash of the `string` as a `UHUGEINT`.
`md5_number_lower(string)`	Returns the lower 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`md5_number_upper(string)`	Returns the upper 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`nextval('sequence_name')`	Return the following value of the sequence.
`nullif(a, b)`	Return `NULL` if `a = b`, else return `a`. Equivalent to `CASE WHEN a = b THEN NULL ELSE a END`.
`parse_formatted_bytes(string)`	Parse a human-readable byte size string (e.g., `'16 KiB'`) into a `UBIGINT` number of bytes. Throws an error on invalid input.
`pg_typeof(expression)`	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.
`query(query_string)`	Table function that parses and executes the query defined in *`query_string`*. Only constant strings are allowed. Warning: this function allows invoking arbitrary queries, potentially altering the database state.
`query_table(tbl_name)`	Table function that returns the table given in *`tbl_name`*.
`query_table(tbl_names, [by_name])`	Table function that returns the union of tables given in *`tbl_names`*. If the optional *`by_name`* parameter is set to `true`, it uses `UNION ALL BY NAME` semantics.
`read_blob(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a `BLOB`. See the `read_blob` guide for more details.
`read_text(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a `VARCHAR`. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.

Name	Description
`replace_type(value, source_type, target_type)`	Recursively casts every field of `value` that has type `source_type` to `target_type`. The types are supplied as `NULL`-typed values, e.g., `NULL::VARCHAR`.
`sha1(string)`	Returns a `VARCHAR` with the SHA-1 hash of the `string`.
`sha256(string)`	Returns a `VARCHAR` with the SHA-256 hash of the `string`.
`sleep_ms(milliseconds)`	Pause execution for the specified number of milliseconds. Returns `NULL`.
`stats(expression)`	Returns a `VARIANT` with statistics about the expression. Expression can be a column, constant, or SQL expression.
`txid_current()`	Returns the current transaction's identifier, a `UBIGINT` value. It will assign a new one if the current transaction does not have one already.
`typeof(expression)`	Returns the name of the data type of the result of the expression.
`uuid()`	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`uuidv4()`	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`uuidv7()`	Return a random UUIDv7 similar to this: `81964ebe-00b1-7e1d-b0f9-43c29b6fb8f5`.
`uuid_extract_timestamp(uuidv7)`	Extracts `TIMESTAMP WITH TIME ZONE` from a UUIDv7 value.
`uuid_extract_version(uuid)`	Extracts UUID version (4 or 7).
`version()`	Return the currently active version of DuckDB in this format.

`alias(column)`

Description	Return the name of the column.
Example	`alias(column1)`
Result	`column1`

`can_cast_implicitly(source_value, target_value)`

Description	Whether or not we can implicitly cast from the types of the source value to the target value.
Example	`can_cast_implicitly(1::BIGINT, 1::SMALLINT)`
Result	`false`

`checkpoint(database)`

Description	Synchronize WAL with file for (optional) database without interrupting transactions.
Example	`checkpoint(my_db)`
Result	success Boolean

coalesce(expr, ...)

Description	Return the first expression that evaluates to a non-NULL value. Accepts 1 or more parameters. Each expression can be a column, literal value, function result, or many others.
Example	`coalesce(NULL, NULL, 'default_string')`
Result	`default_string`

constant_or_null(arg1, arg2)

Description	If `arg2` is `NULL`, return `NULL`. Otherwise, return `arg1`.
Example	`constant_or_null(42, NULL)`
Result	`NULL`

count_if(x)

Description	Aggregate function; returns the number of rows where the `BOOLEAN` argument `x` is `true`.
Example	`count_if(42 > 0)`
Result	`1`

create_sort_key(parameters...)

Description	Constructs a binary-comparable sort key based on a set of input parameters and sort qualifiers.
Example	`create_sort_key('abc', 'ASC NULLS FIRST');`
Result	`\x02bcd\x00`

current_catalog()

Description	Return the name of the currently active catalog. Default is memory.
Example	`current_catalog()`
Result	`memory`

current_database()

Description	Return the name of the currently active database.
Example	`current_database()`
Result	`memory`

`current_query()`

Description	Return the current query as a string.
Example	`current_query()`
Result	`SELECT current_query();`

`current_query_id()`

Description	Returns the identifier of the current query as a `UBIGINT`.
Example	`current_query_id()`
Result	various

`current_schema()`

Description	Return the name of the currently active schema. Default is main.
Example	`current_schema()`
Result	`main`

`current_schemas(boolean)`

Description	Return list of schemas. Pass a parameter of `true` to include implicit schemas.
Example	`current_schemas(true)`
Result	`['temp', 'main', 'pg_catalog']`

`current_setting('setting_name')`

Description	Return the current value of the configuration setting.
Example	`current_setting('access_mode')`
Result	`automatic`

`currval('sequence_name')`

Description	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.
Example	`currval('my_sequence_name')`
Result	`1`

`duckdb_format_sql(sql[, config])`

Description	Formats the `sql` string, e.g., by placing each clause on a separate line and normalizing the case of keywords. The optional `config` is a `MAP(VARCHAR, VARCHAR)` with the keys `'indent_size'`, `'inline_threshold'` and `'keyword_case'` (`'upper'`, `'lower'` or `'preserve'`). Provided by the `autocomplete` extension.
Example	`duckdb_format_sql('select a,b from tbl where x=1')`
Result	`SELECT a, b\nFROM tbl\nWHERE x = 1` (where `\n` denotes a newline)

`error(message)`

Description	Throws the given error `message`.
Example	`error('access_mode')`

`equi_width_bins(min, max, bincount, nice := false)`

Description	Returns the upper boundaries of a partition of the interval `[min, max]` into `bin_count` equal-sized subintervals (for use with, e.g., `histogram`). If `nice = true`, then `min`, `max` and `bincount` may be adjusted to produce more aesthetically pleasing results.
Example	`equi_width_bins(0.1, 2.7, 4, true)`
Result	`[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]`

`force_checkpoint(database)`

Description	Synchronize WAL with file for (optional) database interrupting transactions.
Example	`force_checkpoint(my_db)`
Result	success Boolean

`gen_random_uuid()`

Description	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`gen_random_uuid()`
Result	various

`get_block_size(database)`

Description	Returns the block size of the attached database with the given name.
Example	`get_block_size('file_db')`
Result	`262144`

`get_type(expression)`

Description	Returns the data type of the result of the `expression` as a value of type `TYPE`.
Example	`get_type([1, 2])`
Result	`INTEGER[]`

`getenv(var)`

Description	Returns the value of the environment variable `var`. Only available in the command line client.
Example	`getenv('HOME')`
Result	`/path/to/user/home`

`getvariable('variable_name')`

Description	Returns the value of the SQL variable named `variable_name`, or `NULL` if it is not set. See the SET VARIABLE statement for more details.
Example	`getvariable('my_var')`
Result	various

`hash(value)`

Description	Returns a `UBIGINT` with the hash of the `value`. The used hash function may change across DuckDB versions.
Example	`hash('🦆')`
Result	`4164431626903154684`

`icu_sort_key(string, collator)`

Description	Surrogate sort key used to sort special characters according to the specific locale. Collator parameter is optional. Only available when the ICU extension is installed.
Example	`icu_sort_key('ö', 'DE')`
Result	`470145960106`

`if(a, b, c)`

Description	Ternary conditional operator; returns b if a, else returns c. Equivalent to `CASE WHEN a THEN b ELSE c END`.
Example	`if(2 > 1, 3, 4)`
Result	`3`

`ifnull(expr, other)`

Description	A two-argument version of coalesce.
Example	`ifnull(NULL, 'default_string')`
Result	`default_string`

`in_search_path(database_name, schema_name)`

Description	Returns whether or not the given database and schema are in the search path.
Example	`in_search_path('memory', 'main')`
Result	`true`

`is_histogram_other_bin(arg)`

Description	Returns `true` when `arg` is the ”catch-all element” of its datatype for the purpose of the `histogram_exact` function, which is equal to the ”right-most boundary” of its datatype for the purpose of the `histogram` function.
Example	`is_histogram_other_bin('')`
Result	`true`

`make_type(name, ...)`

Description	Constructs a value of type TYPE from a type name and optional type parameters.
Example	`make_type('DECIMAL', 10, 2)`
Result	`DECIMAL(10,2)`

`md5(string)`

Description	Returns the MD5 hash of the `string` as a VARCHAR.
Example	`md5('abc')`
Result	`900150983cd24fb0d6963f7d28e17f72`

`md5_number(string)`

Description	Returns the MD5 hash of the `string` as a UHUGEINT.
Example	`md5_number('abc')`
Result	`152195979970564155685860391459828531600`

`md5_number_lower(string)`

Description	Returns the lower 8 bytes of the MD5 hash of `string` as a `UBIGINT`.
Example	`md5_number_lower('abc')`
Result	`8250560606382298838`

`md5_number_upper(string)`

Description	Returns the upper 8 bytes of the MD5 hash of `string` as a `UBIGINT`.
Example	`md5_number_upper('abc')`
Result	`12704604231530709392`

`nextval('sequence_name')`

Description	Return the following value of the sequence.
Example	`nextval('my_sequence_name')`
Result	2

`nullif(a, b)`

Description	Return `NULL` if a = b, else return a. Equivalent to `CASE WHEN a = b THEN NULL ELSE a END`.
Example	`nullif(1+1, 2)`
Result	`NULL`

`parse_formatted_bytes(string)`

Description	Parse a human-readable byte size string (e.g., `'16 KiB'`) into a `UBIGINT` number of bytes. Throws an error on invalid input.
Example	`parse_formatted_bytes('1.5 GiB')`
Result	`1610612736`

`pg_typeof(expression)`

Description	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.
Example	`pg_typeof('abc')`
Result	`varchar`

`query(query_string)`

Description	Table function that parses and executes the query defined in `query_string`. Only constant strings are allowed. Warning: this function allows invoking arbitrary queries, potentially altering the database state.
Example	`query('SELECT 42 AS x')`
Result	`42`

`query_table(tbl_name)`

Description	Table function that returns the table given in `tbl_name`.
Example	`query_table('t1')`
Result	(the rows of `t1`)

`query_table(tbl_names, [by_name])`

Description	Table function that returns the union of tables given in `tbl_names`. If the optional `by_name` parameter is set to `true`, it uses `UNION ALL BY NAME` semantics.
Example	`query_table(['t1', 't2'])`
Result	(the union of the two tables)

`read_blob(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
Example	`read_blob('hello.bin')`
Result	`hello\x0A`

`read_text(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
Example	`read_text('hello.txt')`
Result	`hello\n`

`replace_type(value, source_type, target_type)`

Description	Recursively casts every field of `value` that has type `source_type` to `target_type`. The source and target types are supplied as `NULL`-typed values, e.g., `NULL::VARCHAR`. Useful for converting the field types of a deeply nested value (e.g., a `STRUCT` or `LIST`) in one call.
Example	`replace_type([{'a': '42'}], NULL::VARCHAR, NULL::INTEGER)`

Result	`[{'a': 42}]`

`sha1(string)`

Description	Returns a VARCHAR with the SHA-1 hash of the `string`.
Example	`sha1('🦆')`
Result	`949bf843dc338be348fb9525d1eb535d31241d76`

`sha256(string)`

Description	Returns a VARCHAR with the SHA-256 hash of the `string`.
Example	`sha256('🦆')`
Result	`d7a5c5e0d1d94c32218539e7e47d4ba9c3c7b77d61332fb60d633dde89e473fb`

`sleep_ms(milliseconds)`

Description	Pause execution for the specified number of milliseconds. Returns `NULL`.
Example	`sleep_ms(500)`
Result	`NULL`

`stats(expression)`

Description	Returns a `VARIANT` with statistics about the expression. Expression can be a column, constant, or SQL expression.
Example	`stats(5)`
Result	`{'approx_unique': 1, 'has_no_null': true, 'has_null': false, 'max': 5, 'min': 5}`

`txid_current()`

Description	Returns the current transaction's identifier, a `UBIGINT` value. It will assign a new one if the current transaction does not have one already.
Example	`txid_current()`
Result	various

`typeof(expression)`

Description	Returns the name of the data type of the result of the expression.
Example	`typeof('abc')`

Result	VARCHAR

`uuid()`

Description	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`uuid()`
Result	various

`uuidv4()`

Description	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`uuidv4()`
Result	various

`uuidv7()`

Description	Return a random UUIDv7 similar to this: `81964ebe-00b1-7e1d-b0f9-43c29b6fb8f5`.
Example	`uuidv7()`
Result	various

`uuid_extract_timestamp(uuidv7)`

Description	Extracts `TIMESTAMP WITH TIME ZONE` from a UUIDv7 value.
Example	`uuid_extract_timestamp(uuidv7())`
Result	`2025-04-19 15:51:20.07+00`

`uuid_extract_version(uuid)`

Description	Extracts UUID version (4 or 7).
Example	`uuid_extract_version(uuidv7())`
Result	7

`version()`

Description	Return the currently active version of DuckDB in this format.
Example	`version()`
Result	various

Utility Table Functions

A table function is used in place of a table in a FROM clause.

Name	Description
`generate_series(start, stop, step)`	Returns a table of values between `start` and `stop` (inclusive), incrementing by `step`.
`glob(search_path)`	Return filenames found at the location indicated by the *search_path* in a single column named `file`. The *search_path* may contain glob pattern matching syntax.
`range(start, stop, step)`	Returns a table of values between `start` and `stop` (exclusive), incrementing by `step`.
`repeat(value, count)`	Returns a table with `count` rows, each containing `value`.
`repeat_row(varargs, num_rows)`	Returns a table with `num_rows` rows, each containing the fields defined in `varargs`.
`sql_tokenize(sql)`	Splits the `sql` string into tokens. Returns a table with the columns `start` (the byte offset of the token), `token_type` (e.g., `KEYWORD`, `IDENTIFIER`, `NUMBER_LITERAL`) and `word`. Provided by the `autocomplete` extension.
`unnest(list)`	Returns a table with one row per element of `list`.

`generate_series(start, stop, step)`

Description	Returns a table of values between `start` and `stop`, incrementing by `step`. The `stop` parameter is inclusive, and the `stop` and `step` arguments are optional. See the list variant for the available argument combinations.
Example	`generate_series(2, 5)`
Result	4 rows: `2, 3, 4, 5`

`glob(search_path)`

Description	Return filenames found at the location indicated by the *search_path* in a single column named `file`. The *search_path* may contain glob pattern matching syntax.
Example	`glob('*')`
Result	(table of filenames)

`range(start, stop, step)`

Description	Returns a table of values between `start` and `stop`, incrementing by `step`. The `stop` parameter is exclusive, and the `start` and `step` arguments are optional. See the list variant for the available argument combinations.
Example	`range(2, 5)`
Result	3 rows: `2, 3, 4`

`repeat(value, count)`

Description	Returns a table with `count` rows, each containing `value`.
Example	`repeat('foo', 3)`
Result	3 rows of `'foo'`

`repeat_row(varargs, num_rows)`

Description	Returns a table with `num_rows` rows, each containing the fields defined in `varargs`.
Example	`repeat_row(1, 2, 'foo', num_rows = 3)`
Result	3 rows of `1, 2, 'foo'`

`sql_tokenize(sql)`

Description	Splits the `sql` string into tokens. Returns a table with the columns `start` (the byte offset of the token), `token_type` (e.g., `KEYWORD`, `IDENTIFIER`, `NUMBER_LITERAL`) and `word`. Provided by the `autocomplete` extension.
Example	`sql_tokenize('SELECT 42 AS x')`
Result	4 rows: `(0, KEYWORD, SELECT), (7, NUMBER_LITERAL, 42), (10, KEYWORD, AS), (13, IDENTIFIER, x)`

`unnest(list)`

Description	Returns a table with one row per element of `list`. See Unnesting for details, including recursive unnesting of nested lists and structs.
Example	`unnest([1, 2, 3])`
Result	3 rows: `1, 2, 3`

Variant Functions

This section describes functions that operate on the `VARIANT` type.

> New Most `variant_*` functions were introduced in DuckDB 2.0. The `variant_extract`, `variant_normalize` and `variant_typeof` functions were already available in DuckDB 1.5.

Name	Description
`variant_typeof(v)`	Return the full type of a `VARIANT` value, including nested structure.
`variant_type(v)`	Return the top-level type category of a `VARIANT` value.
`variant_keys(v)`	Return the keys of a `VARIANT` object as a list.
`variant_extract(v, field)`	Extract a nested value by object field name or list index. Returns a `VARIANT`.
`variant_extract_string(v, field)`	Extract a nested value as a `VARCHAR`.

Name	Description
`variant_array_length(v)`	Return the number of elements in a `VARIANT` array.
`variant_exists(v, key)`	Return whether the given key exists in a `VARIANT` object.
`variant_contains(haystack, needle)`	Return whether the `needle` `VARIANT` is contained in the `haystack` `VARIANT`.
`variant_comparator(v)`	Return a binary sort key that orders a `VARIANT` value by the logical ordering of `VARIANT` values.
`variant_normalize(v)`	Return a canonical representation of a `VARIANT` value, so that values representing the same data share the same form.
`variant_group_object(key, value)`	Aggregate. Return a `VARIANT` object built from all `key`, `value` pairs in the group.
`variant_group_array(x)`	Aggregate. Collect all values of `x` into a `VARIANT` array.

`variant_typeof(v)`

Description	Return the full type of a `VARIANT` value, including nested structure.
Example	`variant_typeof('{"a": 1}'::JSON::VARIANT)`
Result	`OBJECT(a)`

`variant_type(v)`

Description	Return the top-level type category of a `VARIANT` value.
Example	`variant_type('{"a": 1}'::JSON::VARIANT)`
Result	`OBJECT`

`variant_keys(v)`

Description	Return the keys of a `VARIANT` object as a list.
Example	`variant_keys('{"a": 1, "b": 2}'::JSON::VARIANT)`
Result	`[a, b]`

`variant_extract(v, field)`

Description	Extract a nested value by object field name or list index. Returns a `VARIANT`.
Example	`variant_extract('{"name": "Fry"}'::JSON::VARIANT, 'name')`
Result	`Fry`

`variant_extract_string(v, field)`

Description	Extract a nested value as a VARCHAR.

Example	`variant_extract_string('{"name": "Fry"}'::JSON::VARIANT, 'name')`
Result	`Fry`

`variant_array_length(v)`

Description	Return the number of elements in a `VARIANT` array.
Example	`variant_array_length('[1, 2, 3]'::JSON::VARIANT)`
Result	`3`

`variant_exists(v, key)`

Description	Return whether the given key exists in a `VARIANT` object.
Example	`variant_exists('{"a": 1}'::JSON::VARIANT, 'a')`
Result	`true`

`variant_contains(haystack, needle)`

Description	Return whether the `needle` `VARIANT` is contained in the `haystack` `VARIANT`.
Example	`variant_contains('{"user": {"id": 42}}'::JSON::VARIANT, '{"user": {"id": 42}}'::JSON::VARIANT)`
Result	`true`

`variant_comparator(v)`

Description	Return a binary sort key (a `BLOB`) that orders a `VARIANT` value according to the logical ordering of `VARIANT` values. This is used internally so that `VARIANT`s compare and sort consistently with the corresponding native DuckDB types, rather than by their internal byte representation.
Example	`variant_comparator({'a': 1, 'b': 2}::VARIANT)`
Result	A `BLOB` sort key

Values are ordered first by type bucket (for example, booleans before numbers, numbers before strings) and then by value within a bucket, so that sorting a `VARIANT` column matches the order of the underlying values:

```
SELECT i
FROM range(-2, 2) t(i)
ORDER BY i::VARIANT;
```

i
-2
-1
0

i
1

`variant_normalize(v)`

Description	Return a canonical representation of a `VARIANT` value. Two `VARIANT` values that represent the same data normalize to the same result, regardless of how they were constructed or physically stored. This is useful when comparing or storing `VARIANT` values.
Example	`variant_normalize({'b': [1, 2, 3], 'a': 42}::VARIANT)`
Result	A normalized `VARIANT`

`variant_group_object(key, value)`

Description	Aggregate function. Returns a `VARIANT` object containing all `key`, `value` pairs in the group, where `key` is a `VARCHAR` and `value` is a `VARIANT`. A `NULL` key or a duplicate key raises an error.
Example	`variant_group_object('duck', 'goose'::VARIANT)`
Result	`{'duck': goose}`

> New The `variant_group_object` aggregate was added in DuckDB 2.0.

`variant_group_array(x)`

Description	Collects all values of `x` into a `VARIANT` array, preserving order. Defined as a macro equivalent to `list(x)::VARIANT`, so it has aggregate semantics and also works as a window function. `NULL` values are kept as `VARIANT` nulls.
Example	`variant_group_array(i)`
Result	`[0, 1, 2, 3]`

> New The `variant_group_array` function was added in DuckDB 2.0.

Example

The `variant_*` functions are useful for inspecting and filtering semi-structured data stored in a `VARIANT` column:

```
CREATE TABLE events (payload VARIANT);
INSERT INTO events
VALUES ('{"user": {"id": 42, "tags": ["a", "b"]}}'::JSON::VARIANT);

SELECT variant_type(payload), variant_keys(payload)
FROM events;

SELECT *
FROM events
WHERE variant_contains(payload, {'user': {'id': 42}}::VARIANT);
```

Window Functions

DuckDB supports window functions, which can use multiple rows to calculate a value for each row. Window functions are blocking operators, i.e., they require their entire input to be buffered, making them one of the most memory-intensive operators in SQL.

Window functions are available in SQL since SQL:2003 and are supported by major SQL database systems.

Examples

Generate a `row_number` column to enumerate rows:

```
SELECT row_number() OVER ()
FROM sales;
```

> **Tip.** If you only need a number for each row in a table, you can use the `rowid` pseudocolumn.

Generate a `row_number` column to enumerate rows, ordered by `time`:

```
SELECT row_number() OVER (ORDER BY time)
FROM sales;
```

Generate a `row_number` column to enumerate rows, ordered by `time` and partitioned by `region`:

```
SELECT row_number() OVER (PARTITION BY region ORDER BY time)
FROM sales;
```

Compute the difference between the current and the previous-by-`time` `amount`:

```
SELECT amount - lag(amount) OVER (ORDER BY time)
FROM sales;
```

Compute the percentage of the total `amount` of sales per `region` for each row:

```
SELECT amount / sum(amount) OVER (PARTITION BY region)
FROM sales;
```

Syntax

Window functions can be used in the `SELECT`, `ORDER BY`, and `QUALIFY` clauses. To share `OVER` specifications between functions, use the statement's `WINDOW` clause and use the `OVER <window_name>`{:.language-sql .highlight} syntax.

General-Purpose Window Functions

The table below shows the available general window functions.

Name	Description
`cume_dist([ORDER BY ordering])`	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows.
`dense_rank()`	The rank of the current row *without gaps;* this function counts peer groups.
`fill(expr [ ORDER BY ordering])`	Fill in missing values using linear interpolation with `ORDER BY` as the X-axis.
`first_value(expr[ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.

Name	Description
`lag(expr[, offset[, default]][ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is `offset` rows (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) before the current row within the window frame; if there is no such row, instead return `default` (which must be of the same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `NULL`.
`last_value(expr[ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
`lead(expr[, offset[, default]][ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is `offset` rows after the current row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) within the window frame; if there is no such row, instead return `default` (which must be of the same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `NULL`.
`nth_value(expr, nth[ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the nth row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame (counting from 1); `NULL` if no such row.
`ntile(num_buckets[ ORDER BY ordering])`	An integer ranging from 1 to `num_buckets`, dividing the partition as equally as possible.
`percent_rank([ORDER BY ordering])`	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`.
`rank([ORDER BY ordering])`	The rank of the current row *with gaps;* same as `row_number` of its first peer.
`row_number([ORDER BY ordering])`	The number of the current row within the partition, counting from 1.

`cume_dist([ORDER BY ordering])`

Description	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows. If an `ORDER BY` clause is specified, the distribution is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`DOUBLE`
Example	`cume_dist()`

`dense_rank()`

Description	The rank of the current row *without gaps;* this function counts peer groups.
Return type	`BIGINT`
Example	`dense_rank()`
Aliases	`rank_dense()`

`fill(expr[ ORDER BY ordering])`

Description	Replaces NULL values of expr with a linear interpolation based on the closest non-NULL values and the sort values. Both values must support arithmetic and there must be only one ordering key. For missing values at the ends, linear extrapolation is used. Failure to interpolate results in the NULL value being retained.
Return type	Same type as expr
Example	fill(column)

first_value(expr[ORDER BY ordering][IGNORE NULLS])

Description	Returns expr evaluated at the row that is the first row (with a non-null value of expr if IGNORE NULLS is set) of the window frame. If an ORDER BY clause is specified, the first row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as expr
Example	first_value(column)

lag(expr[, offset[, default]][ORDER BY ordering][IGNORE NULLS])

Description	Returns expr evaluated at the row that is offset rows (among rows with a non-null value of expr if IGNORE NULLS is set) before the current row within the window frame; if there is no such row, instead return default (which must be of the same type as expr). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to NULL. If an ORDER BY clause is specified, the lagged row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as expr
Example	lag(column, 3, 0)

last_value(expr[ORDER BY ordering][IGNORE NULLS])

Description	Returns expr evaluated at the row that is the last row (among rows with a non-null value of expr if IGNORE NULLS is set) of the window frame. If omitted, offset defaults to 1 and default to NULL. If an ORDER BY clause is specified, the last row is determined within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as expr
Example	last_value(column)

lead(expr[, offset[, default]][ORDER BY ordering][IGNORE NULLS])

Description	Returns expr evaluated at the row that is offset rows after the current row (among rows with a non-null value of expr if IGNORE NULLS is set) within the window frame; if there is no such row, instead return default (which must be of the same type as expr). Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to NULL. If an ORDER BY clause is specified, the leading row number is computed within the frame using the provided ordering instead of the frame ordering.

Return type	Same type as `expr`
Example	`lead(column, 3, 0)`

`nth_value(expr, nth[ ORDER BY ordering][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the nth row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame (counting from 1); `NULL` if no such row. If an `ORDER BY` clause is specified, the nth row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as `expr`
Example	`nth_value(column, 2)`

`ntile(num_buckets[ ORDER BY ordering])`

Description	An integer ranging from 1 to `num_buckets`, dividing the partition as equally as possible. If an `ORDER BY` clause is specified, the ntile is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`BIGINT`
Example	`ntile(4)`

`percent_rank([ORDER BY ordering])`

Description	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`. If an `ORDER BY` clause is specified, the relative rank is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`DOUBLE`
Example	`percent_rank()`

`rank([ORDER BY ordering])`

Description	The rank of the current row *with gaps*; same as `row_number` of its first peer. If an `ORDER BY` clause is specified, the rank is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`BIGINT`
Example	`rank()`

`row_number([ORDER BY ordering])`

Description	The number of the current row within the partition, counting from 1. If an `ORDER BY` clause is specified, the row number is computed within the frame using the provided ordering instead of the frame ordering.

Return type	BIGINT
Example	row_number()

Aggregate Window Functions

All aggregate functions can be used in a windowing context, including the optional FILTER clause. The first and last aggregate functions are shadowed by the respective general-purpose window functions, with the minor consequence that the FILTER clause is not available for these but IGNORE NULLS is.

DISTINCT Arguments

All aggregate window functions support using a DISTINCT clause for the arguments. When the DISTINCT clause is provided, only distinct values are considered in the computation of the aggregate. This is typically used in combination with the COUNT aggregate to get the number of distinct elements; but it can be used together with any aggregate function in the system. There are some aggregates that are insensitive to duplicate values (e.g., min, max) and for them this clause is parsed and ignored.

```
-- Count the number of distinct users at a given point in time
SELECT count(DISTINCT name) OVER (ORDER BY time) FROM sales;
-- Concatenate those distinct users into a list
SELECT list(DISTINCT name) OVER (ORDER BY time) FROM sales;
```

ORDER BY Arguments

All aggregate window functions support using an ORDER BY argument clause that is *different* from the window ordering. When the ORDER BY argument clause is provided, the values being aggregated are sorted before applying the function. Usually this is not important, but there are some order-sensitive aggregates that can have indeterminate results (e.g., mode, list and string_agg). These can be made deterministic by ordering the arguments. For order-insensitive aggregates, this clause is parsed and ignored.

```
-- Compute the modal value up to each time, breaking ties in favor of the most recent value.
SELECT mode(value ORDER BY time DESC) OVER (ORDER BY time) FROM sales;
```

The SQL standard does not provide for using ORDER BY with general-purpose window functions, but we have extended all of these functions (except dense_rank) to accept this syntax and use framing to restrict the range that the secondary ordering applies to.

```
-- Compare each athlete's time in an event with the best time to date
SELECT
    event,
    date,
    athlete,
    time,
    first_value(time ORDER BY time ASC) OVER w AS record_time,
    first_value(athlete ORDER BY time ASC) OVER w AS record_athlete,
FROM meet_results
WINDOW w AS (PARTITION BY event ORDER BY datetime)
ORDER BY ALL;
```

Note that there is no comma separating the arguments from the ORDER BY clause.

Nulls

All general-purpose window functions that accept IGNORE NULLS respect nulls by default. This default behavior can optionally be made explicit via RESPECT NULLS.

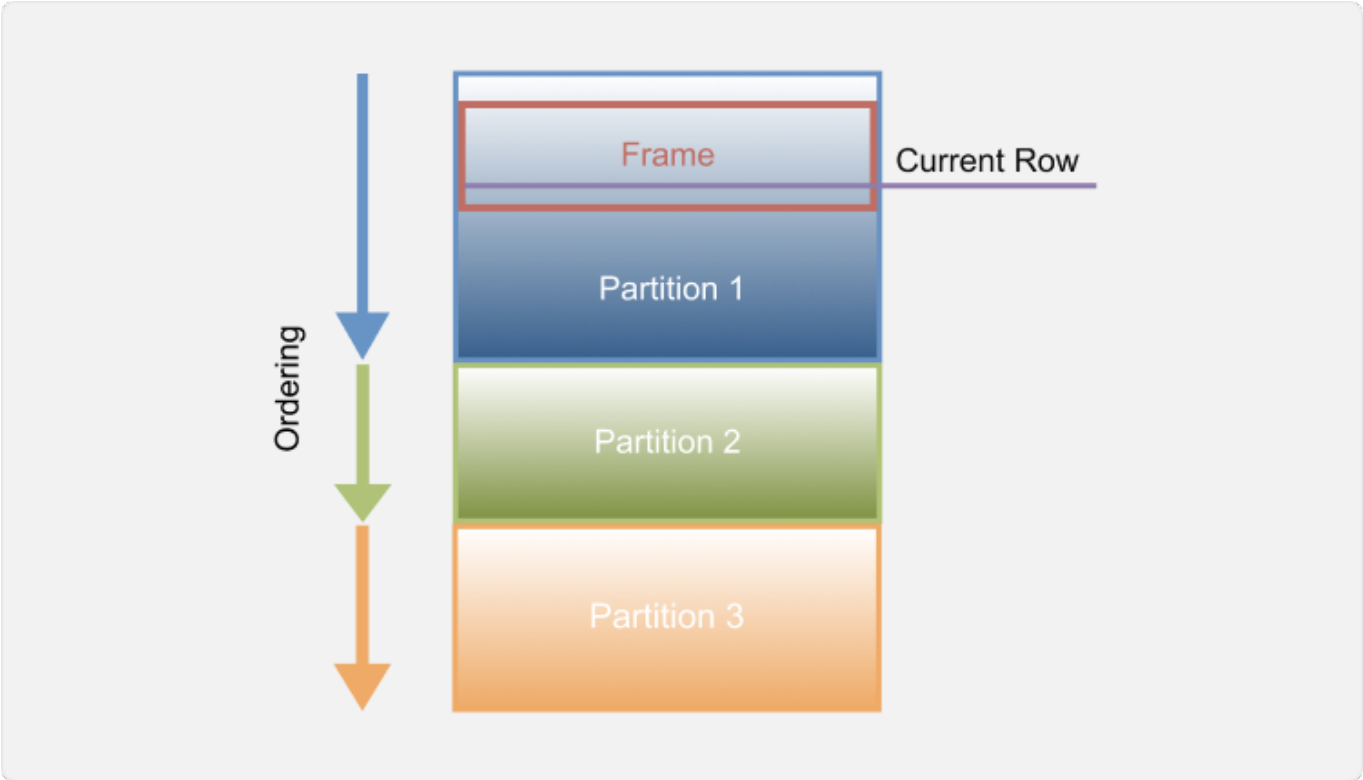
In contrast, all aggregate window functions (except for list and its aliases, which retain NULL values unless a FILTER condition excludes those rows, e.g., list(x) FILTER (WHERE x IS NOT NULL)) ignore nulls and do not accept RESPECT NULLS. For example,

`sum(column) OVER (ORDER BY time) AS cumulativeColumn` computes a cumulative sum where rows with a `NULL` value of `column` have the same value of `cumulativeColumn` as the row that precedes them.

Evaluation

Windowing works by breaking a relation up into independent *partitions*, *ordering* those partitions, and then computing a new column for each row as a function of the nearby values. Some window functions depend only on the partition boundary and the ordering, but a few (including all the aggregates) also use a *frame*. Frames are specified as a number of rows on either side (*preceding* or *following*) of the *current row*. The distance can be specified as a number of *rows*, as a *range* of values using the partition's ordering value and a distance, or as a number of *groups* (sets of rows with the same sort value).

The full syntax is shown in the diagram at the top of the page, and this diagram visually illustrates computation environment:



Partition and Ordering

Partitioning breaks the relation up into independent, unrelated pieces. Partitioning is optional, and if none is specified then the entire relation is treated as a single partition. Window functions cannot access values outside of the partition containing the row they are being evaluated at.

Ordering is also optional, but without it the results of general-purpose window functions and order-sensitive aggregate functions, and the order of framing are not well-defined. Each partition is ordered using the same ordering clause.

Here is a table of power generation data, available as a CSV file (`power-plant-generation-history.csv`). To load the data, run:

```
CREATE TABLE "Generation History" AS
    FROM 'power-plant-generation-history.csv';
```

After partitioning by plant and ordering by date, it will have this layout:

Plant	Date	MWh
Boston	2019-01-02	564337
Boston	2019-01-03	507405
Boston	2019-01-04	528523
Boston	2019-01-05	469538
Boston	2019-01-06	474163
Boston	2019-01-07	507213
Boston	2019-01-08	613040
Boston	2019-01-09	582588
Boston	2019-01-10	499506
Boston	2019-01-11	482014
Boston	2019-01-12	486134
Boston	2019-01-13	531518
Worcester	2019-01-02	118860
Worcester	2019-01-03	101977
Worcester	2019-01-04	106054
Worcester	2019-01-05	92182
Worcester	2019-01-06	94492
Worcester	2019-01-07	99932
Worcester	2019-01-08	118854
Worcester	2019-01-09	113506
Worcester	2019-01-10	96644
Worcester	2019-01-11	93806
Worcester	2019-01-12	98963
Worcester	2019-01-13	107170

In what follows, we shall use this table (or small sections of it) to illustrate various pieces of window function evaluation.

The simplest window function is `row_number()`. This function just computes the 1-based row number within the partition using the query:

```
SELECT
    "Plant",
    "Date",
    row_number() OVER (PARTITION BY "Plant" ORDER BY "Date") AS "Row"
FROM "Generation History"
ORDER BY 1, 2;
```

The result will be the following:

Plant	Date	Row
Boston	2019-01-02	1
Boston	2019-01-03	2
Boston	2019-01-04	3

Plant	Date	Row
...	...	...
Worcester	2019-01-02	1
Worcester	2019-01-03	2
Worcester	2019-01-04	3
...	...	...

Note that even though the function is computed with an ORDER BY clause, the result does not have to be sorted, so the SELECT also needs to be explicitly sorted if that is desired.

Framing

Framing specifies a set of rows relative to each row where the function is evaluated. The distance from the current row is given as an expression either PRECEDING or FOLLOWING the current row in the order specified by the ORDER BY clause in the OVER specification. This distance can either be specified as an integral number of ROWS or GROUPS, or as a RANGE delta expression. It is invalid for a frame to start after it ends. For a RANGE specification, there must be only one ordering expression and it must support subtraction unless only the sentinel boundary values UNBOUNDED PRECEDING / UNBOUNDED FOLLOWING / CURRENT ROW are used. Using the EXCLUDE clause, rows comparing equal to the current row in the specified ordering expression (so-called peers) can be excluded from the frame.

The default frame is unbounded (i.e., the entire partition) when no ORDER BY clause is present and RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW when an ORDER BY clause is present. By default, the CURRENT ROW boundary value (but not the CURRENT ROW in the EXCLUDE clause) means the current row and all its peers when RANGE or GROUP framing are used but it means only the current row when ROWS framing is used.

ROWS Framing Here is a simple ROW frame query, using an aggregate function:

```
SELECT points,
    sum(points) OVER (
        ROWS BETWEEN 1 PRECEDING
                 AND 1 FOLLOWING) AS we
FROM results;
```

This query computes the sum of each point and the points on either side of it:

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

Notice that at the edge of the partition, there are only two values added together. This is because frames are cropped to the edge of the partition.

RANGE Framing Returning to the power data, suppose the data is noisy. We might want to compute a 7 day moving average for each plant to smooth out the noise. To do this, we can use this window query:

```
SELECT "Plant", "Date",
    avg("MWh") OVER (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                  AND INTERVAL 3 DAYS FOLLOWING)
        AS "MWh 7-day Moving Average"
FROM "Generation History"
ORDER BY 1, 2;
```

This query partitions the data by `Plant` (to keep the different power plants' data separate), orders each plant's partition by `Date` (to put the energy measurements next to each other), and uses a RANGE frame of three days on either side of each day for the `avg` (to handle any missing days). This is the result:

Plant	Date	MWh 7-day Moving Average
Boston	2019-01-02	517450.75
Boston	2019-01-03	508793.20
Boston	2019-01-04	508529.83
...	...	...
Boston	2019-01-13	499793.00
Worcester	2019-01-02	104768.25
Worcester	2019-01-03	102713.00
Worcester	2019-01-04	102249.50
...	...	...

GROUPS Framing The third type of framing counts *groups* of rows relative the current row. A *group* in this framing is a set of values with identical ORDER BY values. If we assume that power is being generated on every day, we can use GROUPS framing to compute the moving average of all power generated in the system without having to resort to date arithmetic:

```
SELECT "Date", "Plant",
    avg("MWh") OVER (
        ORDER BY "Date" ASC
        GROUPS BETWEEN 3 PRECEDING
                   AND 3 FOLLOWING)
        AS "MWh 7-day Moving Average"
FROM "Generation History"
ORDER BY 1, 2;
```

Date	Plant	MWh 7-day Moving Average
2019-01-02	Boston	311109.500
2019-01-02	Worcester	311109.500
2019-01-03	Boston	305753.100
2019-01-03	Worcester	305753.100
2019-01-04	Boston	305389.667
2019-01-04	Worcester	305389.667
...	...	...

Date	Plant	MWh 7-day Moving Average
2019-01-12	Boston	309184.900
2019-01-12	Worcester	309184.900
2019-01-13	Boston	299469.375
2019-01-13	Worcester	299469.375

Notice how the values for each date are the same.

EXCLUDE Clause EXCLUDE is an optional modifier to the frame clause for excluding rows around the CURRENT ROW. This is useful when you want to compute some aggregate value of nearby rows to see how the current row compares to it.

In the following example, we want to know how an athlete's time in an event compares to the average of all the times recorded for their event within ±10 days:

```
SELECT
    event,
    date,
    athlete,
    avg(time) OVER w AS recent,
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY date
    RANGE BETWEEN INTERVAL 10 DAYS PRECEDING AND INTERVAL 10 DAYS FOLLOWING
        EXCLUDE CURRENT ROW
)
ORDER BY event, date, athlete;
```

There are four options for EXCLUDE that specify how to treat the current row:

- CURRENT ROW – exclude just the current row
- GROUP – exclude the current row and all its "peers" (rows that have the same ORDER BY value)
- TIES – exclude all peer rows, but *not* the current row (this makes a hole on either side)
- NO OTHERS – don't exclude anything (the default)

Exclusion is implemented for both windowed aggregates as well as for the `first`, `last` and `nth_value` functions.

`WINDOW` Clauses

Multiple different OVER clauses can be specified in the same SELECT, and each will be computed separately. Often, however, we want to use the same layout for multiple window functions. The WINDOW clause can be used to define a *named* window that can be shared between multiple window functions:

```
SELECT "Plant", "Date",
    min("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    avg("MWh") OVER seven AS "MWh 7-day Moving Average",
    max("MWh") OVER seven AS "MWh 7-day Moving Maximum"
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

The three window functions will also share the data layout, which will improve performance.

Multiple windows can be defined in the same WINDOW clause by comma-separating them:

```
SELECT "Plant", "Date",
    min("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    avg("MWh") OVER seven AS "MWh 7-day Moving Average",
    max("MWh") OVER seven AS "MWh 7-day Moving Maximum",
    min("MWh") OVER three AS "MWh 3-day Moving Minimum",
    avg("MWh") OVER three AS "MWh 3-day Moving Average",
    max("MWh") OVER three AS "MWh 3-day Moving Maximum"
FROM "Generation History"
WINDOW
    seven AS (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                  AND INTERVAL 3 DAYS FOLLOWING),
    three AS (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 1 DAYS PRECEDING
        AND INTERVAL 1 DAYS FOLLOWING)
ORDER BY 1, 2;
```

The queries above do not use a number of clauses commonly found in select statements, like `WHERE`, `GROUP BY`, etc. For more complex queries you can find where `WINDOW` clauses fall in the canonical order of the `SELECT statement`.

Filtering the Results of Window Functions Using `QUALIFY`

Window functions are executed after the `WHERE` and `HAVING` clauses have been already evaluated, so it's not possible to use these clauses to filter the results of window functions The `QUALIFY` clause avoids the need for a subquery or `WITH` clause to perform this filtering.

Box and Whisker Queries

All aggregates can be used as windowing functions, including the complex statistical functions. These function implementations have been optimized for windowing, and we can use the window syntax to write queries that generate the data for moving box-and-whisker plots:

```
SELECT "Plant", "Date",
    min("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    quantile_cont("MWh", [0.25, 0.5, 0.75]) OVER seven
        AS "MWh 7-day Moving IQR",
    max("MWh") OVER seven AS "MWh 7-day Moving Maximum",
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

Constraints

In SQL, constraints can be specified for tables. Constraints enforce certain properties over data that is inserted into a table. Constraints can be specified along with the schema of the table as part of the CREATE TABLE statement. In certain cases, constraints can also be added to a table using the ALTER TABLE statement, but this is not currently supported for all constraints.

> **Warning.** Constraints have a strong impact on performance: they slow down loading and updates but speed up certain queries. Please consult the Performance Guide for details.

Syntax

Check Constraint

Check constraints allow you to specify an arbitrary Boolean expression. Any columns that *do not* satisfy this expression violate the constraint. For example, we could enforce that the name column does not contain spaces using the following CHECK constraint.

```
CREATE TABLE students (name VARCHAR CHECK (NOT contains(name, ' ')));
INSERT INTO students VALUES ('this name contains spaces');
```

```
Constraint Error:
CHECK constraint failed on table students with expression CHECK((NOT contains("name", ' ')))
```

Not Null Constraint

A not-null constraint specifies that the column cannot contain any NULL values. By default, all columns in tables are nullable. Adding NOT NULL to a column definition enforces that a column cannot contain NULL values.

```
CREATE TABLE students (name VARCHAR NOT NULL);
INSERT INTO students VALUES (NULL);
```

```
Constraint Error:
NOT NULL constraint failed: students.name
```

Primary Key and Unique Constraint

Primary key or unique constraints define a column, or set of columns, that are a unique identifier for a row in the table. The constraint enforces that the specified columns are *unique* within a table, i.e., that at most one row contains the given values for the set of columns.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO students VALUES (1, 'Student 2');
```

```
Constraint Error:
Duplicate key "id: 1" violates primary key constraint
```

```
CREATE TABLE students (id INTEGER, name VARCHAR, PRIMARY KEY (id, name));
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO students VALUES (1, 'Student 2');
INSERT INTO students VALUES (1, 'Student 1');
```

```
Constraint Error:
Duplicate key "id: 1, name: Student 1" violates primary key constraint
```

To enforce this property efficiently, an ART index is automatically created for every primary key or unique constraint that is defined in the table.

Primary key constraints and unique constraints are identical except for two points:

- A table can only have one primary key constraint defined, but many unique constraints
- A primary key constraint also enforces the keys to not be `NULL`.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR, email VARCHAR UNIQUE);
INSERT INTO students VALUES (1, 'Student 1', 'student1@uni.com');
INSERT INTO students VALUES (2, 'Student 2', 'student1@uni.com');
```

```
Constraint Error:
Duplicate key "email: student1@uni.com" violates unique constraint.
```

```
INSERT INTO students (id, name) VALUES (3, 'Student 3');
INSERT INTO students (name, email) VALUES ('Student 3', 'student3@uni.com');
```

```
Constraint Error:
NOT NULL constraint failed: students.id
```

> **Warning.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, leading to constraint errors such as `violates primary key constraint` and `violates unique constraint`. See the Indexes page for more details.

You can also define a uniqueness constraint on multiple columns:

```
CREATE TABLE integers (i INTEGER, j INTEGER, k INTEGER, UNIQUE (i, j));
INSERT INTO integers VALUES (1, 2, 3);
INSERT INTO integers VALUES (1, 4, 5);
INSERT INTO integers VALUES (1, 2, 5);
```

```
Constraint Error:
Duplicate key "i: 1, j: 2" violates unique constraint.
```

Foreign Keys

Foreign keys define a column, or set of columns, that refer to a primary key or unique constraint from *another* table. The constraint enforces that the key exists in the other table.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE subjects (id INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE exams (
    exam_id INTEGER PRIMARY KEY,
    subject_id INTEGER REFERENCES subjects(id),
    student_id INTEGER REFERENCES students(id),
    grade INTEGER
);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO subjects VALUES (1, 'CS 101');
INSERT INTO exams VALUES (1, 1, 1, 10);
INSERT INTO exams VALUES (2, 1, 2, 10);
```

```
Constraint Error:
Violates foreign key constraint because key "id: 2" does not exist in the referenced table
```

To enforce this property efficiently, an ART index is automatically created for every foreign key constraint that is defined in the table.

> **Warning.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, leading to constraint errors such as `violates primary key constraint` and `violates unique constraint`. See the indexes section for more details.

Indexes

Index Types

DuckDB has two built-in index types. Indexes can also be defined via extensions.

Min-Max Index (Zonemap)

A min-max index (also known as zonemap or block range index) is *automatically created* for columns of all general-purpose data types.

Adaptive Radix Tree (ART)

An Adaptive Radix Tree (ART) is mainly used to ensure primary key constraints and to speed up point and very highly selective (i.e., < 0.1%) queries. ART indexes can be created manually using the `CREATE INDEX` clause and they are automatically created for columns with a `UNIQUE` or `PRIMARY KEY` constraint.

> **Warning.** ART indexes must currently be able to fit in memory during index creation. Avoid creating ART indexes if the index does not fit in memory during index creation.

Indexes Defined by Extensions

DuckDB supports R-trees for spatial indexing via the `spatial` extension.

Persistence

Both min-max indexes and ART indexes are persisted on disk.

`CREATE INDEX` and `DROP INDEX` Statements

To create an ART index, use the `CREATE INDEX` statement. To drop an ART index, use the `DROP INDEX` statement.

Limitations of ART Indexes

ART indexes create a secondary copy of the data in a second location. Maintaining that second copy complicates processing. Thus, certain limitations currently apply when it comes to modifying data that is also stored in secondary indexes.

> As expected, indexes have a strong effect on performance, slowing down loading and updates, but speeding up certain queries. Please consult the Performance Guide for details.

Constraint Checking in UPDATE Statements

UPDATE statements on indexed columns and columns that cannot be updated in place are transformed into a DELETE of the original row followed by an INSERT of the updated row. This rewrite has performance implications, particularly for wide tables, as entire rows are rewritten instead of only the affected columns.

Additionally, it causes the following constraint-checking limitation of UPDATE statements. The same limitation exists in other DBMSs, like PostgreSQL.

In the example below, note how the number of rows exceeds DuckDB's standard vector size, which is 2048 by default. The UPDATE statement is rewritten into a DELETE, followed by an INSERT. This rewrite happens per chunk of data (2048 rows) moving through DuckDB's processing pipeline. When updating `i = 2047` to `i = 2048`, we do not yet know that 2048 becomes 2049, and so forth. That is because we have not yet seen that chunk. Thus, we throw a constraint violation.

```
CREATE TABLE my_table (i INTEGER PRIMARY KEY);
INSERT INTO my_table SELECT range FROM range(3_000);
UPDATE my_table SET i = i + 1;
```

```
Constraint Error:
Duplicate key "i: 2048" violates primary key constraint.
```

A workaround is to split the UPDATE into a `DELETE ... RETURNING ...` followed by an INSERT, with some additional logic to (temporarily) store the result of the DELETE. All statements should be run inside a transaction via BEGIN, and eventually COMMIT.

Here's an example of how that could look like in the command line client.

```
CREATE TABLE my_table (i INTEGER PRIMARY KEY);
INSERT INTO my_table SELECT range FROM range(3_000);

BEGIN;
CREATE TEMP TABLE tmp AS SELECT i FROM my_table;
DELETE FROM my_table;
INSERT INTO my_table SELECT i FROM tmp;
DROP TABLE tmp;
COMMIT;
```

In other clients, you might be able to fetch the result of `DELETE ... RETURNING ...`. Then, you can use that result in a subsequent `INSERT ...` statement, or potentially make use of DuckDB's `Appender` (if available in the client).

Over-Eager Constraint Checking in Foreign Keys

This limitation occurs if you meet the following conditions:

- A table has a `FOREIGN KEY` constraint.
- There is an UPDATE on a composite payload column (e.g., a LIST or a STRUCT), the corresponding `PRIMARY KEY` table, which DuckDB rewrites into a DELETE followed by an INSERT.
- The to-be-deleted row exists in the foreign key table.

If these hold, you'll encounter an unexpected constraint violation:

```
CREATE TABLE pk_table (id INTEGER PRIMARY KEY, payload VARCHAR[]);
INSERT INTO pk_table VALUES (1, ['hello']);
CREATE TABLE fk_table (id INTEGER REFERENCES pk_table(id));
INSERT INTO fk_table VALUES (1);
UPDATE pk_table SET payload = ['world'] WHERE id = 1;
```

```
Constraint Error:
Violates foreign key constraint because key "id: 1" is still referenced by a foreign key in a different
table. If this is an unexpected constraint violation, please refer to our foreign key limitations in the
documentation
```

The reason for this is that DuckDB does not yet support "looking ahead". During the INSERT, it is unaware it will reinsert the foreign key value as part of the UPDATE rewrite.

Meta Queries

Information Schema

The views in the `information_schema` are SQL-standard views that describe the catalog entries of the database. These views can be filtered to obtain information about a specific column or table. DuckDB's implementation is based on PostgreSQL's information schema.

Tables

`character_sets`: Character Sets

Column	Description	Type	Example
`character_set_catalog`	Currently not implemented – always `NULL`.	`VARCHAR`	`NULL`
`character_set_schema`	Currently not implemented – always `NULL`.	`VARCHAR`	`NULL`
`character_set_name`	Name of the character set, currently implemented as showing the name of the database encoding.	`VARCHAR`	`'UTF8'`
`character_repertoire`	Character repertoire, showing UCS if the encoding is `UTF8`, else just the encoding name.	`VARCHAR`	`'UCS'`
`form_of_use`	Character encoding form, same as the database encoding.	`VARCHAR`	`'UTF8'`
`default_collate_catalog`	Name of the database containing the default collation (always the current database).	`VARCHAR`	`'my_db'`
`default_collate_schema`	Name of the schema containing the default collation.	`VARCHAR`	`'pg_catalog'`
`default_collate_name`	Name of the default collation.	`VARCHAR`	`'ucs_basic'`

`columns`: Columns

The view that describes the catalog information for columns is `information_schema.columns`. It lists the columns present in the database and has the following layout:

Column	Description	Type	Example
`table_catalog`	Name of the database containing the table (always the current database).	VARCHAR	`'my_db'`
`table_schema`	Name of the schema containing the table.	VARCHAR	`'main'`

Column	Description	Type	Example
`table_name`	Name of the table.	`VARCHAR`	`'widgets'`
`column_name`	Name of the column.	`VARCHAR`	`'price'`
`ordinal_position`	Ordinal position of the column within the table (count starts at 1).	`INTEGER`	`5`
`column_default`	Default expression of the column.	`VARCHAR`	`1.99`
`is_nullable`	`YES` if the column is possibly nullable, `NO` if it is known not nullable.	`VARCHAR`	`'YES'`
`data_type`	Data type of the column.	`VARCHAR`	`'DECIMAL(18, 2)'`
`character_maximum_length`	If `data_type` identifies a character or bit string type, the declared maximum length; `NULL` for all other data types or if no maximum length was declared.	`INTEGER`	`255`
`character_octet_length`	If `data_type` identifies a character type, the maximum possible length in octets (bytes) of a datum; `NULL` for all other data types. The maximum octet length depends on the declared character maximum length (see above) and the character encoding.	`INTEGER`	`1073741824`
`numeric_precision`	If `data_type` identifies a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. For all other data types, this column is `NULL`.	`INTEGER`	`18`
`numeric_scale`	If `data_type` identifies a numeric type, this column contains the (declared or implicit) scale of the type for this column. The scale indicates the number of digits after the decimal point. For all other data types, this column is `NULL`.	`INTEGER`	`2`
`datetime_precision`	If `data_type` identifies a date, time, timestamp, or interval type, this column contains the (declared or implicit) fractional seconds precision of the type for this column, that is, the number of decimal digits maintained following the decimal point in the seconds value. No fractional seconds are currently supported in DuckDB. For all other data types, this column is `NULL`.	`INTEGER`	`0`
`is_generated`	`ALWAYS` if the column is a generated column, `NEVER` otherwise. Available as of DuckDB 2.0.	`VARCHAR`	`'NEVER'`
`generation_expression`	If the column is a generated column, the expression used to compute it; `NULL` otherwise. Available as of DuckDB 2.0.	`VARCHAR`	`NULL`

> New The `is_generated` and `generation_expression` columns were introduced in DuckDB 2.0.

`constraint_column_usage`: Constraint Column Usage

This view describes all columns in the current database that are used by some constraint. For a check constraint, this view identifies the columns that are used in the check expression. For a not-null constraint, this view identifies the column that the constraint is defined on. For a foreign key constraint, this view identifies the columns that the foreign key references. For a unique or primary key constraint, this view identifies the constrained columns.

Column	Description	Type	Example
`table_catalog`	Name of the database that contains the table that contains the column that is used by some constraint (always the current database)	`VARCHAR`	`'my_db'`
`table_schema`	Name of the schema that contains the table that contains the column that is used by some constraint	`VARCHAR`	`'main'`
`table_name`	Name of the table that contains the column that is used by some constraint	`VARCHAR`	`'widgets'`
`column_name`	Name of the column that is used by some constraint	`VARCHAR`	`'price'`
`constraint_catalog`	Name of the database that contains the constraint (always the current database)	`VARCHAR`	`'my_db'`
`constraint_schema`	Name of the schema that contains the constraint	`VARCHAR`	`'main'`
`constraint_name`	Name of the constraint	`VARCHAR`	`'exam_id_students_id_fkey'`

`key_column_usage`: Key Column Usage

Column	Description	Type	Example
`constraint_catalog`	Name of the database that contains the constraint (always the current database).	`VARCHAR`	`'my_db'`
`constraint_schema`	Name of the schema that contains the constraint.	`VARCHAR`	`'main'`
`constraint_name`	Name of the constraint.	`VARCHAR`	`'exams_exam_id_fkey'`
`table_catalog`	Name of the database that contains the table that contains the column that is restricted by this constraint (always the current database).	`VARCHAR`	`'my_db'`
`table_schema`	Name of the schema that contains the table that contains the column that is restricted by this constraint.	`VARCHAR`	`'main'`
`table_name`	Name of the table that contains the column that is restricted by this constraint.	`VARCHAR`	`'exams'`
`column_name`	Name of the column that is restricted by this constraint.	`VARCHAR`	`'exam_id'`

Column	Description	Type	Example
`ordinal_position`	Ordinal position of the column within the constraint key (count starts at 1).	`INTEGER`	`1`
`position_in_unique_constraint`	For a foreign-key constraint, ordinal position of the referenced column within its unique constraint (count starts at 1); otherwise `NULL`.	`INTEGER`	`1`

`referential_constraints`: Referential Constraints

Column	Description	Type	Example
`constraint_catalog`	Name of the database containing the constraint (always the current database).	`VARCHAR`	`'my_db'`
`constraint_schema`	Name of the schema containing the constraint.	`VARCHAR`	`main`
`constraint_name`	Name of the constraint.	`VARCHAR`	`exam_id_students_id_fkey`
`unique_constraint_catalog`	Name of the database that contains the unique or primary key constraint that the foreign key constraint references.	`VARCHAR`	`'my_db'`
`unique_constraint_schema`	Name of the schema that contains the unique or primary key constraint that the foreign key constraint references.	`VARCHAR`	`'main'`
`unique_constraint_name`	Name of the unique or primary key constraint that the foreign key constraint references.	`VARCHAR`	`'students_id_pkey'`
`match_option`	Match option of the foreign key constraint. Always `NONE`.	`VARCHAR`	`NONE`
`update_rule`	Update rule of the foreign key constraint. Always `NO ACTION`.	`VARCHAR`	`NO ACTION`
`delete_rule`	Delete rule of the foreign key constraint. Always `NO ACTION`.	`VARCHAR`	`NO ACTION`

`schemata`: Database, Catalog and Schema

The top level catalog view is `information_schema.schemata`. It lists the catalogs and the schemas present in the database and has the following layout:

Column	Description	Type	Example
`catalog_name`	Name of the database that the schema is contained in.	`VARCHAR`	`'my_db'`
`schema_name`	Name of the schema.	`VARCHAR`	`'main'`
`schema_owner`	Name of the owner of the schema. Not yet implemented.	`VARCHAR`	`'duckdb'`

Column	Description	Type	Example
default_character_set_catalog	Applies to a feature not available in DuckDB.	VARCHAR	NULL
default_character_set_schema	Applies to a feature not available in DuckDB.	VARCHAR	NULL
default_character_set_name	Applies to a feature not available in DuckDB.	VARCHAR	NULL
sql_path	Applies to a feature not available in DuckDB.	VARCHAR	NULL

`tables`: Tables and Views

The view that describes the catalog information for tables and views is `information_schema.tables`. It lists the tables present in the database and has the following layout:

Column	Description	Type	Example
table_catalog	The catalog the table or view belongs to.	VARCHAR	'my_db'
table_schema	The schema the table or view belongs to.	VARCHAR	'main'
table_name	The name of the table or view.	VARCHAR	'widgets'
table_type	The type of table. One of: `BASE TABLE`, `LOCAL TEMPORARY`, `VIEW`.	VARCHAR	'BASE TABLE'
self_referencing_column_name	Applies to a feature not available in DuckDB.	VARCHAR	NULL
reference_generation	Applies to a feature not available in DuckDB.	VARCHAR	NULL
user_defined_type_catalog	If the table is a typed table, the name of the database that contains the underlying data type (always the current database), else `NULL`. Currently unimplemented.	VARCHAR	NULL
user_defined_type_schema	If the table is a typed table, the name of the schema that contains the underlying data type, else `NULL`. Currently unimplemented.	VARCHAR	NULL
user_defined_type_name	If the table is a typed table, the name of the underlying data type, else `NULL`. Currently unimplemented.	VARCHAR	NULL
is_insertable_into	`YES` if the table is insertable into, `NO` if not (Base tables are always insertable into, views not necessarily.)	VARCHAR	'YES'
is_typed	`YES` if the table is a typed table, `NO` if not.	VARCHAR	'NO'
commit_action	Not yet implemented.	VARCHAR	'NO'

`table_constraints`: Table Constraints

Column	Description	Type	Example
`constraint_catalog`	Name of the database that contains the constraint (always the current database).	`VARCHAR`	`'my_db'`
`constraint_schema`	Name of the schema that contains the constraint.	`VARCHAR`	`'main'`
`constraint_name`	Name of the constraint.	`VARCHAR`	`'exams_exam_id_fkey'`
`table_catalog`	Name of the database that contains the table (always the current database).	`VARCHAR`	`'my_db'`
`table_schema`	Name of the schema that contains the table.	`VARCHAR`	`'main'`
`table_name`	Name of the table.	`VARCHAR`	`'exams'`
`constraint_type`	Type of the constraint: `CHECK`, `FOREIGN KEY`, `PRIMARY KEY`, or `UNIQUE`.	`VARCHAR`	`'FOREIGN KEY'`
`is_deferrable`	`YES` if the constraint is deferrable, `NO` if not.	`VARCHAR`	`'NO'`
`initially_deferred`	`YES` if the constraint is deferrable and initially deferred, `NO` if not.	`VARCHAR`	`'NO'`
`enforced`	Always `YES`.	`VARCHAR`	`'YES'`
`nulls_distinct`	If the constraint is a unique constraint, then `YES` if the constraint treats `NULL`s as distinct or `NO` if it treats `NULL`s as not distinct, otherwise `NULL` for other types of constraints.	`VARCHAR`	`'YES'`

Catalog Functions

Several functions are also provided to see details about the catalogs and schemas that are configured in the database.

Function	Description	Example	Result
`current_catalog()`	Return the name of the currently active catalog. Default is memory.	`current_catalog()`	`'memory'`
`current_schema()`	Return the name of the currently active schema. Default is main.	`current_schema()`	`'main'`
`current_schemas(boolean)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.	`current_schemas(true)`	`['main', 'main', 'main', 'pg_catalog']`

DuckDB_% Metadata Functions

DuckDB offers a collection of table functions that provide metadata about the current database. These functions reside in the `main` schema and their names are prefixed with `duckdb_`.

The resultset returned by a `duckdb_` table function may be used just like an ordinary table or view. For example, you can use a `duckdb_` function call in the `FROM` clause of a `SELECT` statement, and you may refer to the columns of its returned resultset elsewhere in the statement, for example in the `WHERE` clause.

Table functions are still functions, and you should write parentheses after the function name to call it to obtain its returned resultset:

```
SELECT * FROM duckdb_settings();
```

Alternatively, you may execute table functions also using the CALL-syntax:

```
CALL duckdb_settings();
```

In this case too, the parentheses are mandatory.

> For some of the duckdb_% functions, there is also an identically named view available, which also resides in the main schema. Typically, these views do a SELECT on the duckdb_ table function with the same name, while filtering out those objects that are marked as internal. We mention it here, because if you accidentally omit the parentheses in your duckdb_ table function call, you might still get a result, but from the identically named view.

Example:

The duckdb_views() *table function* returns all views, including those marked internal:

```
SELECT * FROM duckdb_views();
```

The duckdb_views *view* returns views that are not marked as internal:

```
SELECT * FROM duckdb_views;
```

duckdb_available_metrics

> New The duckdb_available_metrics() function was introduced in DuckDB 2.0.

The duckdb_available_metrics() function lists the metrics that can be collected by the profiler, e.g., using the tracked_metrics setting.

Column	Description	Type
metric_name	The name of the metric, e.g., query.cpu_time.	VARCHAR
metric_type	The type of the metric's value, e.g., double, uint64, string or map.	VARCHAR
description	Human readable text that describes the metric.	VARCHAR
unit	The unit of the metric, e.g., seconds or rows. Empty for unitless metrics.	VARCHAR

duckdb_columns

The duckdb_columns() function provides metadata about the columns available in the DuckDB instance.

Column	Description	Type
database_name	The name of the database that contains the column object.	VARCHAR
database_oid	Internal identifier of the database that contains the column object.	BIGINT
schema_name	The SQL name of the schema that contains the table object that defines this column.	VARCHAR
schema_oid	Internal identifier of the schema object that contains the table of the column.	BIGINT
table_name	The SQL name of the table that defines the column.	VARCHAR

Column	Description	Type
`table_oid`	Internal identifier (name) of the table object that defines the column.	`BIGINT`
`column_name`	The SQL name of the column.	`VARCHAR`
`column_index`	The unique position of the column within its table.	`INTEGER`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`internal`	`true` if this column is built-in, `false` if it is user-defined.	`BOOLEAN`
`column_default`	The default value of the column (expressed in SQL)	`VARCHAR`
`is_nullable`	`true` if the column can hold `NULL` values; `false` if the column cannot hold `NULL`-values.	`BOOLEAN`
`is_generated`	`true` if the column is a generated column, `false` otherwise. Available as of DuckDB 2.0.	`BOOLEAN`
`generation_expression`	If the column is a generated column, the expression used to compute it; `NULL` otherwise. Available as of DuckDB 2.0.	`VARCHAR`
`data_type`	The name of the column data type.	`VARCHAR`
`data_type_id`	The internal identifier of the column data type.	`BIGINT`
`character_maximum_length`	Always `NULL`. DuckDB text types do not enforce a value length restriction based on a length type parameter.	`INTEGER`
`numeric_precision`	The number of units (in the base indicated by `numeric_precision_radix`) used for storing column values. For integral and approximate numeric types, this is the number of bits. For decimal types, this is the number of digit positions.	`INTEGER`
`numeric_precision_radix`	The number-base of the units in the `numeric_precision` column. For integral and approximate numeric types, this is `2`, indicating the precision is expressed as a number of bits. For the `decimal` type this is `10`, indicating the precision is expressed as a number of decimal positions.	`INTEGER`
`numeric_scale`	Applicable to `decimal` type. Indicates the maximum number of fractional digits (i.e., the number of digits that may appear after the decimal separator).	`INTEGER`

The `information_schema.columns` system view provides a more standardized way to obtain metadata about database columns, but the `duckdb_columns` function also returns metadata about DuckDB internal objects. (In fact, `information_schema.columns` is implemented as a query on top of `duckdb_columns()`)

`duckdb_connection_count`

The `duckdb_connection_count()` function returns the number of open connections to the DuckDB instance.

Column	Description	Type
`count`	The number of open connections.	`UBIGINT`

`duckdb_constraints`

The `duckdb_constraints()` function provides metadata about the constraints available in the DuckDB instance.

Column	Description	Type
database_name	The name of the database that contains the constraint.	VARCHAR
database_oid	Internal identifier of the database that contains the constraint.	BIGINT
schema_name	The SQL name of the schema that contains the table on which the constraint is defined.	VARCHAR
schema_oid	Internal identifier of the schema object that contains the table on which the constraint is defined.	BIGINT
table_name	The SQL name of the table on which the constraint is defined.	VARCHAR
table_oid	Internal identifier (name) of the table object on which the constraint is defined.	BIGINT
constraint_index	Indicates the position of the constraint as it appears in its table definition.	BIGINT
constraint_type	Indicates the type of constraint. Applicable values are CHECK, FOREIGN KEY, PRIMARY KEY, NOT NULL, UNIQUE.	VARCHAR
constraint_text	The definition of the constraint expressed as a SQL-phrase. (Not necessarily a complete or syntactically valid DDL-statement.)	VARCHAR
expression	If constraint is a check constraint, the definition of the condition being checked, otherwise NULL.	VARCHAR
constraint_column_indexes	An array of table column indexes referring to the columns that appear in the constraint definition.	BIGINT[]
constraint_column_names	An array of table column names appearing in the constraint definition.	VARCHAR[]
constraint_name	The name of the constraint.	VARCHAR
referenced_table	The table referenced by the constraint.	VARCHAR
referenced_column_names	The column names referenced by the constraint.	VARCHAR[]

The information_schema.referential_constraints and information_schema.table_constraints system views provide a more standardized way to obtain metadata about constraints, but the duckdb_constraints function also returns metadata about DuckDB internal objects. (In fact, information_schema.referential_constraints and information_schema.table_constraints are implemented as a query on top of duckdb_constraints())

duckdb_coordinate_systems

The duckdb_coordinate_systems() function lists the available coordinate reference systems (CRS) for use with the GEOMETRY data type.

Column	Description	Type
database_name	The name of the database that stores the CRS definition.	VARCHAR
database_oid	Internal identifier of the database that stores the CRS definition.	BIGINT
schema_name	The SQL name of the schema that stores the CRS definition.	VARCHAR
schema_oid	Internal identifier of the schema that stores the CRS definition.	BIGINT
crs_oid	Unique internal identifier for the CRS.	BIGINT

Column	Description	Type
`crs_name`	A unique name that identifies the CRS. (Typically, this value is identical to the `auth_name` and `auth_code`, separated by a colon)	`VARCHAR`
`auth_name`	The name of the authorithy that defined the CRS.	`VARCHAR`
`auth_code`	The code that defined the CRS within its authority.	`VARCHAR`
`projjson`	The projjson encoding of the CRS.	`VARCHAR`
`wkt2_2019`	The Well-Known Text (WKT) representation of the CRS.	`VARCHAR`

This function lists both built-in CRSs as well as the ones that are installed by extensions (such as the `SPATIAL` extension.

`duckdb_databases`

The `duckdb_databases()` function lists the databases that are accessible from within the current DuckDB process. Apart from the database associated at startup, the list also includes databases that were attached later on to the DuckDB process.

Column	Description	Type
`database_name`	The name of the database, or the alias if the database was attached using an ALIAS-clause.	`VARCHAR`
`database_oid`	The internal identifier of the database.	`VARCHAR`
`path`	The file path associated with the database.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`true` indicates a system or built-in database. `false` indicates a user-defined database.	`BOOLEAN`
`type`	The type indicates the type of RDBMS implemented by the attached database. For DuckDB databases, that value is `duckdb`.	`VARCHAR`
`readonly`	Denotes whether the database is read-only.	`BOOLEAN`
`options`	The options used in the ATTACH statement, as a map of option names to their string values.	`MAP(VARCHAR, VARCHAR)`

`duckdb_dependencies`

The `duckdb_dependencies()` function provides metadata about the dependencies available in the DuckDB instance.

Column	Description	Type
`classid`	Always 0	`BIGINT`
`objid`	The internal id of the object.	`BIGINT`
`objsubid`	Always 0	`INTEGER`
`refclassid`	Always 0	`BIGINT`
`refobjid`	The internal id of the dependent object.	`BIGINT`
`refobjsubid`	Always 0	`INTEGER`

Column	Description	Type
deptype	The type of dependency. Either regular (n) or automatic (a).	VARCHAR

duckdb_extension_repositories

> New The duckdb_extension_repositories() function was introduced in DuckDB 2.0.

The duckdb_extension_repositories() function provides metadata about the extension repositories known to the DuckDB instance, including the built-in ones (e.g., core, core_nightly and community) and those created with CREATE EXTENSION REPOSITORY.

Column	Description	Type
repository_name	The name of the repository.	VARCHAR
prefix	The URL prefix (or path) from which extensions are installed.	VARCHAR
type	The type of the repository, e.g., CORE or COMMUNITY.	VARCHAR
key_fingerprints	The SHA-256 fingerprints of the public keys pinned for the repository. NULL for built-in repositories.	VARCHAR[]
public_keys	The public keys pinned for the repository. NULL for built-in repositories.	VARCHAR[]

duckdb_extensions

The duckdb_extensions() function provides metadata about the extensions available in the DuckDB instance.

Column	Description	Type
extension_name	The name of the extension.	VARCHAR
loaded	true if the extension is loaded, false if it's not loaded.	BOOLEAN
installed	true if the extension is installed, false if it's not installed.	BOOLEAN
install_path	(BUILT-IN) if the extension is built-in, otherwise, the filesystem path where the binary that implements the extension resides.	VARCHAR
description	Human readable text that describes the extension's functionality.	VARCHAR
aliases	List of alternative names for this extension.	VARCHAR[]
extension_version	The version of the extension (vX.Y.Z for stable versions and 6-character hash for unstable versions).	VARCHAR
install_mode	The installation mode that was used to install the extension: UNKNOWN, REPOSITORY, CUSTOM_PATH, STATICALLY_LINKED, NOT_INSTALLED, NULL.	VARCHAR
installed_from	Name of the repository the extension was installed from, e.g., community or core_nightly. The empty string denotes the core repository.	VARCHAR

duckdb_functions

The duckdb_functions() function provides metadata about the functions (including macros) available in the DuckDB instance.

Column	Description	Type
database_name	The name of the database that contains this function.	VARCHAR
database_oid	Internal identifier of the database containing the index.	BIGINT
schema_name	The SQL name of the schema where the function resides.	VARCHAR
function_name	The SQL name of the function.	VARCHAR
alias_of	The name of the function this is an alias of, or NULL if this is not an alias.	VARCHAR
function_type	The function kind. Value is one of: table, scalar, aggregate, pragma, macro, table_macro	VARCHAR
description	Description of this function, or NULL if none is registered.	VARCHAR
comment	A comment created by the COMMENT ON statement.	VARCHAR
tags	A map of string key–value pairs.	MAP(VARCHAR, VARCHAR)
return_type	The logical data type name of the returned value. Applicable for scalar and aggregate functions.	VARCHAR
parameters	If the function has parameters, the list of parameter names.	VARCHAR[]
parameter_types	If the function has parameters, a list of logical data type names corresponding to the parameter list.	VARCHAR[]
varargs	The name of the data type in case the function has a variable number of arguments, or NULL if the function does not have a variable number of arguments.	VARCHAR
macro_definition	If this is a macro, the SQL expression that defines it.	VARCHAR
has_side_effects	false if this is a pure function. true if this function changes the database state (like sequence functions nextval() and curval()).	BOOLEAN
internal	true if the function is built-in (defined by DuckDB or an extension), false if it was defined using the CREATE MACRO statement.	BOOLEAN
extension_name	The name of the extension that provides this function, or NULL if none.	VARCHAR
function_oid	The internal identifier for this function.	BIGINT
examples	Examples of using the function. Used to generate the documentation.	VARCHAR[]
stability	The stability of the function (CONSISTENT, VOLATILE, CONSISTENT_WITHIN_QUERY or NULL)	VARCHAR
categories	Categories of the function. Used to generate the documentation.	VARCHAR[]

> New The extension_name column was introduced in DuckDB 2.0.

duckdb_grammar_extensions

> New The duckdb_grammar_extensions() function was introduced in DuckDB 2.0.

The duckdb_grammar_extensions() function lists the extensions to the SQL grammar of the PEG parser that are currently registered, e.g., by extensions.

Column	Description	Type
`name`	The name of the grammar extension.	`VARCHAR`
`description`	Human readable text that describes the grammar extension.	`VARCHAR`

`duckdb_indexes`

The `duckdb_indexes()` function provides metadata about secondary indexes available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this index.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the index.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the table with the secondary index.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object.	`BIGINT`
`index_name`	The SQL name of this secondary index.	`VARCHAR`
`index_oid`	The object identifier of this index.	`BIGINT`
`table_name`	The name of the table with the index.	`VARCHAR`
`table_oid`	Internal identifier (name) of the table object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`is_unique`	`true` if the index was created with the `UNIQUE` modifier, `false` if it was not.	`BOOLEAN`
`is_primary`	Always `false`.	`BOOLEAN`
`expressions`	Always `NULL`.	`VARCHAR`
`sql`	The definition of the index, expressed as a `CREATE INDEX` SQL statement.	`VARCHAR`

Note that `duckdb_indexes` only provides metadata about secondary indexes, i.e., those indexes created by explicit `CREATE INDEX` statements. Primary keys, foreign keys, and `UNIQUE` constraints are maintained using indexes, but their details are included in the `duckdb_constraints()` function.

`duckdb_keywords`

The `duckdb_keywords()` function provides metadata about DuckDB's keywords and reserved words.

Column	Description	Type
`keyword_name`	The keyword.	`VARCHAR`

Column	Description	Type
`keyword_category`	The category of the keyword: `reserved` (cannot be used as an identifier without quoting), `unreserved` (can be used as an identifier), `column_name` (usable as a column name but not as a type or function name), or `type_function` (usable as a type or function name but not as a column name). See Keywords and Identifiers.	`VARCHAR`

`duckdb_log_contexts`

The `duckdb_log_contexts()` function provides information on the contexts of DuckDB log entries.

Column	Description	Type
`context_id`	The identifier of the context. The `context_id` column in the `duckdb_logs` table is a foreign key that points to this column.	`UBIGINT`
`scope`	The scope of the context (`connection`, `database` or `file_opener`).	`VARCHAR`
`connection_id`	The identifier of the connection.	`UBIGINT`
`transaction_id`	The identifier of the transaction.	`UBIGINT`
`query_id`	The identifier of the query.	`UBIGINT`
`thread_id`	The identifier of the thread.	`UBIGINT`

`duckdb_logs`

The `duckdb_logs()` function returns a table of DuckDB log entries.

Column	Description	Type
`context_id`	The identifier of the context of the log entry. Foreign key to the `duckdb_log_contexts` table.	`UBIGINT`
`timestamp`	The timestamp of the log entry.	`TIMESTAMP`
`type`	The type of the log entry.	`VARCHAR`
`log_level`	The level of the log entry (`TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` or `FATAL`).	`VARCHAR`
`message`	The message of the log entry.	`VARCHAR`

`duckdb_memory`

The `duckdb_memory()` function provides metadata about DuckDB's buffer manager.

Column	Description	Type
`tag`	The memory tag. It has one of the following values: `BASE_TABLE`, `HASH_TABLE`, `PARQUET_READER`, `CSV_READER`, `ORDER_BY`, `ART_INDEX`, `COLUMN_DATA`, `METADATA`, `OVERFLOW_STRINGS`, `IN_MEMORY_TABLE`, `ALLOCATOR`, `EXTENSION`.	`VARCHAR`
`memory_usage_bytes`	The memory used (in bytes).	`BIGINT`
`temporary_storage_bytes`	The disk storage used (in bytes).	`BIGINT`

`duckdb_optimizers`

The `duckdb_optimizers()` function provides metadata about the optimization rules (e.g., `expression_rewriter`, `filter_pushdown`) available in the DuckDB instance. These can be selectively turned off using `PRAGMA disabled_optimizers`.

Column	Description	Type
`name`	The name of the optimization rule.	`VARCHAR`

`duckdb_prepared_statements`

The `duckdb_prepared_statements()` function provides metadata about the prepared statements that exist in the current DuckDB session.

Column	Description	Type
`name`	The name of the prepared statement.	`VARCHAR`
`statement`	The SQL statement.	`VARCHAR`
`parameter_types`	The expected parameter types for the statement's parameters. Currently returns `UNKNOWN` for all parameters.	`VARCHAR[]`
`result_types`	The types of the columns in the table returned by the prepared statement.	`VARCHAR[]`

`duckdb_profiling_settings`

The `duckdb_profiling_settings()` macro returns the current profiling-related settings from `duckdb_settings()`.

Column	Description	Type
`name`	The name of the profiling setting.	`VARCHAR`
`value`	The current value of the setting.	`VARCHAR`
`description`	A description of the setting.	`VARCHAR`

duckdb_schemas

The `duckdb_schemas()` function provides metadata about the schemas available in the DuckDB instance.

Column	Description	Type
`oid`	Internal identifier of the schema object.	`BIGINT`
`database_name`	The name of the database that contains this schema.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the schema.	`BIGINT`
`schema_name`	The SQL name of the schema.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`true` if this is an internal (built-in) schema, `false` if this is a user-defined schema.	`BOOLEAN`
`sql`	Always `NULL`	`VARCHAR`
`parent_schema`	The name of the parent schema for a nested schema, or `NULL`.	`VARCHAR`
`parent_schema_oid`	Internal identifier of the parent schema, or `NULL`.	`BIGINT`

The `information_schema.schemata` system view provides a more standardized way to obtain metadata about database schemas.

duckdb_secret_types

The `duckdb_secret_types()` lists secret types that are supported in the current DuckDB session.

Column	Description	Type
`type`	The name of the secret type, e.g., `s3`.	`VARCHAR`
`default_provider`	The default secret provider, e.g., `config`.	`VARCHAR`
`extension`	The extension that registered the secret type, e.g., `aws`.	`VARCHAR`

duckdb_secret_type_parameters

> New The `duckdb_secret_type_parameters()` function was introduced in DuckDB 2.0. It lists the parameters that each secret type and provider accepts.

Column	Description	Type
`type`	The name of the secret type, e.g., `s3`.	`VARCHAR`
`provider`	The secret provider, e.g., `config`.	`VARCHAR`
`name`	The name of the parameter.	`VARCHAR`
`parameter_type`	The data type of the parameter.	`VARCHAR`
`description`	A description of the parameter. Currently always `NULL`.	`VARCHAR`

duckdb_secrets

The `duckdb_secrets()` function provides metadata about the secrets available in the DuckDB instance.

Column	Description	Type
`name`	The name of the secret.	`VARCHAR`
`type`	The type of the secret, e.g., S3, GCS, R2, AZURE.	`VARCHAR`
`provider`	The provider of the secret.	`VARCHAR`
`persistent`	Denotes whether the secret is persistent.	`BOOLEAN`
`storage`	The backend for storing the secret.	`VARCHAR`
`scope`	The scope of the secret.	`VARCHAR[]`
`secret_string`	Returns the content of the secret as a string. Sensitive pieces of information, e.g., the access key, are redacted.	`VARCHAR`

duckdb_sequences

The `duckdb_sequences()` function provides metadata about the sequences available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this sequence	`VARCHAR`
`database_oid`	Internal identifier of the database containing the sequence.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the sequence object.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the sequence object.	`BIGINT`
`sequence_name`	The SQL name that identifies the sequence within the schema.	`VARCHAR`
`sequence_oid`	The internal identifier of this sequence object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`temporary`	Whether this sequence is temporary. Temporary sequences are transient and only visible within the current connection.	`BOOLEAN`
`start_value`	The initial value of the sequence. This value will be returned when `nextval()` is called for the very first time on this sequence.	`BIGINT`
`min_value`	The minimum value of the sequence.	`BIGINT`
`max_value`	The maximum value of the sequence.	`BIGINT`
`increment_by`	The value that is added to the current value of the sequence to draw the next value from the sequence.	`BIGINT`
`cycle`	Whether the sequence should start over when drawing the next value would result in a value outside the range.	`BOOLEAN`
`last_value`	The last value drawn from the sequence using `nextval(...)`, or `NULL` if no value was ever drawn.	`BIGINT`
`sql`	The definition of this object, expressed as SQL DDL-statement.	`VARCHAR`

Attributes like `temporary`, `start_value` etc. correspond to the various options available in the `CREATE SEQUENCE` statement and are documented there in full. Note that the attributes will always be filled out in the `duckdb_sequences` resultset, even if they were not explicitly specified in the `CREATE SEQUENCE` statement.

> 1. The `last_value` column contains the last value that was drawn from the sequence, or `NULL` if a value was never drawn from the sequence.
> 2. If the sequence cycles, then the sequence will start over from the boundary of its range, not necessarily from the value specified as start value.

duckdb_settings

The `duckdb_settings()` function provides metadata about the settings available in the DuckDB instance.

Column	Description	Type
`name`	Name of the setting.	`VARCHAR`
`value`	Current value of the setting.	`VARCHAR`
`description`	A description of the setting.	`VARCHAR`
`input_type`	The logical data type of the setting's value.	`VARCHAR`
`scope`	The scope of the setting (`LOCAL` or `GLOBAL`).	`VARCHAR`

> New As of DuckDB 2.0, `duckdb_settings()` accepts the `debug`, `deprecated`, and `include_all` named parameters.

By default, `duckdb_settings()` returns only regular settings and omits debug-only and deprecated ones. The `debug` and `deprecated` parameters select which category to return instead: for example, `duckdb_settings(deprecated := true)` returns the deprecated settings and `duckdb_settings(debug := true)` returns the debug settings. To return every setting, pass `include_all := true`, which cannot be combined with the other two parameters:

```
SELECT * FROM duckdb_settings(include_all := true);
```

The various settings are described in the configuration page.

duckdb_tables

The `duckdb_tables()` function provides metadata about the base tables available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this table	`VARCHAR`
`database_oid`	Internal identifier of the database containing the table.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the base table.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the base table.	`BIGINT`
`table_name`	The SQL name of the base table.	`VARCHAR`
`table_oid`	Internal identifier of the base table object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`false` if this is a user-defined table.	`BOOLEAN`

Column	Description	Type
`temporary`	Whether this is a temporary table. Temporary tables are not persisted and only visible within the current connection.	`BOOLEAN`
`has_primary_key`	`true` if this table object defines a `PRIMARY KEY`.	`BOOLEAN`
`estimated_size`	The estimated number of rows in the table.	`BIGINT`
`column_count`	The number of columns defined by this object.	`BIGINT`
`index_count`	The number of indexes associated with this table. This number includes all secondary indexes, as well as internal indexes generated to maintain `PRIMARY KEY` and/or `UNIQUE` constraints.	`BIGINT`
`check_constraint_count`	The number of check constraints active on columns within the table.	`BIGINT`
`sql`	The definition of this object, expressed as SQL `CREATE TABLE`-statement.	`VARCHAR`

The `information_schema.tables` system view provides a more standardized way to obtain metadata about database tables that also includes views. But the resultset returned by `duckdb_tables` contains a few columns that are not included in `information_schema.tables`.

`duckdb_temporary_files`

The `duckdb_temporary_files()` function provides metadata about the temporary files DuckDB has written to disk, to offload data from memory. This function mostly exists for debugging and testing purposes.

Column	Description	Type
`path`	The name of the temporary file.	`VARCHAR`
`size`	The size in bytes of the temporary file.	`BIGINT`

`duckdb_triggers`

> New The `duckdb_triggers()` function was introduced in DuckDB 2.0.

The `duckdb_triggers()` function provides metadata about the triggers available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains the trigger.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains the trigger.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the trigger.	`VARCHAR`
`schema_oid`	Internal identifier of the schema that contains the trigger.	`BIGINT`
`trigger_name`	The SQL name of the trigger.	`VARCHAR`
`trigger_oid`	The object identifier of the trigger.	`BIGINT`
`table_name`	The name of the table the trigger is attached to.	`VARCHAR`

Column	Description	Type
`action_timing`	When the trigger fires relative to the triggering statement, e.g., `AFTER`.	`VARCHAR`
`event_manipulation`	The event that fires the trigger: `INSERT`, `UPDATE` or `DELETE`.	`VARCHAR`
`columns`	For `UPDATE OF <column>, ...` triggers, the list of columns. Empty otherwise.	`VARCHAR[]`
`for_each`	The granularity of the trigger: `ROW` or `STATEMENT`.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`temporary`	`true` if this is a temporary trigger, `false` otherwise.	`BOOLEAN`
`sql`	The definition of the trigger expressed as SQL, i.e., a `CREATE TRIGGER` statement.	`VARCHAR`

`duckdb_types`

The `duckdb_types()` function provides metadata about the data types available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this schema.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains the data type.	`BIGINT`
`schema_name`	The SQL name of the schema containing the type definition. Always `main`.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object.	`BIGINT`
`type_name`	The name or alias of this data type.	`VARCHAR`
`type_oid`	The internal identifier of the data type object. If `NULL`, then this is an alias of the type (as identified by the value in the `logical_type` column).	`BIGINT`
`type_size`	The number of bytes required to represent a value of this type in memory.	`BIGINT`
`logical_type`	The 'canonical' name of this data type. The same `logical_type` may be referenced by several types having different `type_names`.	`VARCHAR`
`type_category`	The category to which this type belongs. Data types within the same category generally expose similar behavior when values of this type are used in expressions. For example, the `NUMERIC` type_category includes integers, decimals and floating point numbers.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	Whether this is an internal (built-in) or a user object.	`BOOLEAN`
`labels`	Labels for categorizing types. Used for generating the documentation.	`VARCHAR[]`

duckdb_variables

The `duckdb_variables()` function provides metadata about the variables available in the DuckDB instance.

Column	Description	Type
`name`	The name of the variable, e.g., `x`.	`VARCHAR`
`value`	The value of the variable, e.g., `12`.	`VARCHAR`
`type`	The type of the variable, e.g., `INTEGER`.	`VARCHAR`

duckdb_views

The `duckdb_views()` function provides metadata about the views available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this view.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains this view.	`BIGINT`
`schema_name`	The SQL name of the schema where the view resides.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the view.	`BIGINT`
`view_name`	The SQL name of the view object.	`VARCHAR`
`view_oid`	The internal identifier of this view object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`true` if this is an internal (built-in) view, `false` if this is a user-defined view.	`BOOLEAN`
`temporary`	`true` if this is a temporary view. Temporary views are not persistent and are only visible within the current connection.	`BOOLEAN`
`column_count`	The number of columns defined by this view object.	`BIGINT`
`sql`	The definition of this object, expressed as SQL DDL-statement.	`VARCHAR`

The `information_schema.tables` system view provides a more standardized way to obtain metadata about database views that also includes base tables. But the resultset returned by `duckdb_views` contains also definitions of internal view objects as well as a few columns that are not included in `information_schema.tables`.

DuckDB's SQL Dialect

Overview

DuckDB's SQL dialect is based on PostgreSQL. DuckDB tries to closely match PostgreSQL's semantics, however, some use cases require slightly different behavior. For example, interchangeability with data frame libraries necessitates order preservation of inserts to be supported by default. These differences are documented in the pages below.

Indexing

DuckDB uses 1-based indexing except for JSON objects, which use 0-based indexing.

Examples

The index origin is 1 for strings, lists, etc.

```
SELECT list[1] AS element
FROM (SELECT ['first', 'second', 'third'] AS list);
```

element
varchar
first

The index origin is 0 for JSON objects.

```
SELECT json[1] AS element
FROM (SELECT '["first", "second", "third"]'::JSON AS json);
```

element
json
"second"

Friendly SQL

DuckDB offers several advanced SQL features and syntactic sugar to make SQL queries more concise. We refer to these colloquially as "friendly SQL".

> Several of these features were first introduced by DuckDB, while some are inspired by other systems. Many of the features originally introduced by DuckDB (e.g., `GROUP BY ALL`) have been since adapted by other systems.

> **Tip.** We have a Friendly SQL 2026 Calendar with a short explanation, an example, and an abstract illustration for 12 friendly SQL features.

Clauses

- Creating tables and inserting data:
  - `CREATE OR REPLACE TABLE`: avoid `DROP TABLE IF EXISTS` statements in scripts.
  - `CREATE TABLE ... AS SELECT` (CTAS): create a new table from the output of a table without manually defining a schema.
  - `INSERT INTO ... BY NAME`: this variant of the `INSERT` statement allows using column names instead of positions.
  - `INSERT OR IGNORE INTO ...`: insert the rows that do not result in a conflict due to `UNIQUE` or `PRIMARY KEY` constraints.
  - `INSERT OR REPLACE INTO ...`: insert the rows that do not result in a conflict due to `UNIQUE` or `PRIMARY KEY` constraints. For those that result in a conflict, replace the columns of the existing row to the new values of the to-be-inserted row.
- Describing tables and computing statistics:
  - `DESCRIBE`: provides a succinct summary of the schema of a table or query.
  - `SUMMARIZE`: returns summary statistics for a table or query.
- Making SQL clauses more compact and readable:
  - `FROM`-first syntax with an optional `SELECT` clause: DuckDB allows queries in the form of `FROM tbl` which selects all columns (performing a `SELECT *` statement).
  - `GROUP BY ALL`: omit the group-by columns by inferring them from the list of attributes in the `SELECT` clause.
  - `ORDER BY ALL`: shorthand to order on all columns (e.g., to ensure deterministic results).
  - `SELECT * EXCLUDE`: the `EXCLUDE` option allows excluding specific columns from the `*` expression.
  - `SELECT * REPLACE`: the `REPLACE` option allows replacing specific columns with different expressions in a `*` expression.
  - `UNION BY NAME`: perform the `UNION` operation along the names of columns (instead of relying on positions).
  - Prefix aliases in the `SELECT` and `FROM` clauses: write `x: 42` instead of `42 AS x` for improved readability.
  - Specifying a percentage of the table size for the `LIMIT` clause: write `LIMIT 10%` to return 10% of the query results.
  - `FETCH FIRST`: the SQL-standard alternative to `LIMIT`, e.g., `FETCH FIRST 5 ROWS ONLY`.
  - Grouping by `UNNEST`: use `unnest()` directly in the `GROUP BY` clause to group by the elements of a list.
- Transforming tables:
  - `PIVOT` to turn long tables to wide tables.
  - `UNPIVOT` to turn wide tables to long tables.
- Defining SQL-level variables:
  - `SET VARIABLE`
  - `RESET VARIABLE`
  - The `$name` shorthand for reading variables: write `$x` instead of `getvariable('x')`.
- Organizing the catalog:
  - Nested schemas: create schemas within schemas, e.g., `CREATE SCHEMA finance.reports`.
- Managing extensions and settings:
  - `INSTALL AND LOAD`: install and load an extension in a single statement.
  - `SHOW <setting>`: print the value of a configuration setting, e.g., `SHOW threads`.

Query Features

- Column aliases in `WHERE`, `GROUP BY`, and `HAVING`. (Note that column aliases cannot be used in the `ON` clause of `JOIN` clauses.)
- `COLUMNS()` expression can be used to execute the same expression on multiple columns:
  - with regular expressions
  - with `EXCLUDE` and `REPLACE`
  - with lambda functions

- Reusable column aliases (also known as "lateral column aliases"), e.g.: `SELECT i + 1 AS j, j + 2 AS k FROM range(0, 3) t(i)`
- Advanced aggregation features for analytical (OLAP) queries:
  - `FILTER` clause
  - `GROUPING SETS`, `GROUP BY CUBE`, `GROUP BY ROLLUP` clauses
- `count()` shorthand for `count(*)`
- `IN` operator for lists and maps
- Specifying column names for common table expressions (`WITH`)
- Data-modifying CTEs: use `INSERT`, `UPDATE`, `DELETE` and `COPY` statements in common table expressions
- Specifying column names in the `JOIN` clause
- Using `VALUES` in the `JOIN` clause
- Using `VALUES` in the anchor part of common table expressions
- `SWITCH` statements as syntactic sugar for the `CASE` expression

Literals and Identifiers

- Case-insensitivity while maintaining case of entities in the catalog
- Deduplicating identifiers
- Underscores as digit separators in numeric literals

Data Types

- `MAP` data type
- `UNION` data type

Data Import

- Auto-detecting the headers and schema of CSV files
- Directly querying CSV files and Parquet files
- Replacement scans:
  - You can load from files using the syntax `FROM 'my.csv'`, `FROM 'my.csv.gz'`, `FROM 'my.parquet'`, etc.
  - In Python, you can access Pandas data frames using `FROM df`.
- Filename expansion (globbing), e.g.: `FROM 'my-data/part-*.parquet'`

Functions and Expressions

- Dot operator for function chaining: `SELECT ('hello').upper()`
- String formatters: the `format()` function with the `fmt` syntax and the `printf() function`
- List comprehensions
- List slicing and indexing from the back (`[-1]`)
- String slicing
- `STRUCT.*` notation
- Creating `LIST` using square brackets
- Simple `LIST` and `STRUCT` creation
- Updating the schema of `STRUCT`s

Join Types

- ASOF joins
- LATERAL joins
- POSITIONAL joins
- NEAREST joins for top-k similarity search, e.g., over embeddings

Trailing Commas

DuckDB allows trailing commas, both when listing entities (e.g., column and table names) and when constructing LIST items. For example, the following query works:

```
SELECT
    42 AS x,
    ['a', 'b', 'c',] AS y,
    'hello world' AS z,
;
```

"Top-N in Group" Queries

Computing the "top-N rows in a group" ordered by some criteria is a common task in SQL that unfortunately often requires a complex query involving window functions and/or subqueries.

To aid in this, DuckDB provides the aggregate functions `max(arg, n)`, `min(arg, n)`, `arg_max(arg, val, n)`, `arg_min(arg, val, n)`, `max_by(arg, val, n)` and `min_by(arg, val, n)` to efficiently return the "top" n rows in a group based on a specific column in either ascending or descending order.

For example, let's use the following table:

```
SELECT * FROM t1;
```

grp varchar	val int32
a	2
a	1
b	5
b	4
a	3
b	6

We want to get a list of the top-3 `val` values in each group `grp`. The conventional way to do this is to use a window function in a subquery:

```
SELECT array_agg(rs.val), rs.grp
FROM
    (SELECT val, grp, row_number() OVER (PARTITION BY grp ORDER BY val DESC) AS rid
    FROM t1 ORDER BY val DESC) AS rs
WHERE rid < 4
GROUP BY rs.grp;
```

array_agg(rs.val) int32[]	grp varchar
[3, 2, 1]	a
[6, 5, 4]	b

But in DuckDB, we can do this much more concisely (and efficiently!):

```
SELECT max(val, 3) FROM t1 GROUP BY grp;
```

max(val, 3)
int32[]
[3, 2, 1]
[6, 5, 4]

Related Blog Posts

- "Friendlier SQL with DuckDB" blog post
- "Even Friendlier SQL with DuckDB" blog post
- "SQL Gymnastics: Bending SQL into Flexible New Shapes" blog post

Keywords and Identifiers

Identifiers

Similarly to other SQL dialects and programming languages, identifiers in DuckDB's SQL are subject to several rules.

- Unquoted identifiers need to conform to a number of rules:
  - They must not be a reserved keyword (see `duckdb_keywords()`), e.g., `CREATE TABLE select (i INTEGER)` will fail.
  - They must not start with a number or special character, e.g., `SELECT 123 AS 1col` is invalid.
  - They cannot contain whitespaces (including tabs and newline characters).
- Identifiers can be quoted using double-quote characters (`"`). Quoted identifiers can use any keyword, whitespace or special character, e.g., `"SELECT"` and `" § 🦤 ¶ "` are valid identifiers.
- Double quotes can be escaped by repeating the quote character, e.g., to create an identifier named `IDENTIFIER "X"`, use `"IDENTIFIER ""X"""`.

> New As of DuckDB 2.0, identifiers can also be quoted using backtick characters (`` ` ``), following exactly the same rules as double-quoted identifiers. A backtick is escaped by doubling it, e.g., `` `ab` `` denotes the identifier a`b. Backtick-quoted identifiers are never case-folded and work everywhere double-quoted identifiers do. Note that stored and round-tripped DDL (for example in `duckdb_views()`) still uses double quotes.

Deduplicating Identifiers

In some cases, duplicate identifiers can occur, e.g., column names may conflict when unnesting a nested data structure. In these cases, DuckDB automatically deduplicates column names by renaming them according to the following rules:

- For a column named <name>{:.language-sql .highlight}, the first instance is not renamed.
- Subsequent instances are renamed to <name>_<count>{:.language-sql .highlight}, where <count>{:.language-sql .highlight} starts at 1.

For example:

```
SELECT *
FROM (SELECT unnest({'a': 42, 'b': {'a': 88, 'b': 99}}, recursive := true));
```

a	a_1	b
42	88	99

Identifier Resolution

Identifiers can be *qualified* to remove ambiguity. A fully qualified column reference has the form `<database>.<schema>.<table>.<column>`, and any leading part can be omitted when the remaining name is unambiguous. Table names can likewise be qualified as `<database>.<schema>.<table>`.

When a name is not fully qualified, DuckDB resolves it using the catalog search path. The search path is controlled by the `schema` and `search_path` configuration options: `schema` sets a single default `<catalog>.<schema>` entry, while `search_path` sets a comma-separated list of such entries that are searched in order. Both options write to the same catalog search path, and each entry may be a bare schema or a `<catalog>.<schema>` pair.

For a detailed description of how database and schema names are resolved, including the default database and schema and how to change the search path, see Name Qualification on the ATTACH page.

Database Names

Database names are subject to the rules for identifiers.

Additionally, it is best practice to avoid DuckDB's two internal database schema names, `system` and `temp`. By default, persistent databases are named after their filename without the extension. Therefore, the filenames `system.db` and `temp.db` (as well as `system.duckdb` and `temp.duckdb`) result in the database names `system` and `temp`, respectively. If you need to attach to a database that has one of these names, use an alias, e.g.:

```
ATTACH 'temp.db' AS temp2;
USE temp2;
```

Rules for Case-Sensitivity

Keywords and Function Names

SQL keywords and function names are case-insensitive in DuckDB.

For example, the following two queries are equivalent:

```
select COS(Pi()) as CosineOfPi;
SELECT cos(pi()) AS CosineOfPi;
```

CosineOfPi
-1.0

Case-Sensitivity of Identifiers

Identifiers in DuckDB are always case-insensitive, similarly to PostgreSQL. However, unlike PostgreSQL (and some other major SQL implementations), DuckDB also treats quoted identifiers as case-insensitive.

Comparison of identifiers: Case-insensitivity is implemented using an ASCII-based comparison: `col_A` and `col_a` are equal but `col_á` is not equal to them.

```
SELECT col_A FROM (SELECT 'x' AS col_a); -- succeeds
SELECT col_á FROM (SELECT 'x' AS col_a); -- fails
```

Preserving cases: While DuckDB treats identifiers in a case-insensitive manner, it preserves the cases of these identifiers. That is, each character's case (uppercase/lowercase) is maintained as originally specified by the user even if a query uses different cases when referring to the identifier. For example:

```
CREATE TABLE tbl AS SELECT cos(pi()) AS CosineOfPi;
SELECT cosineofpi FROM tbl;
```

CosineOfPi
-1.0

To change this behavior, use the `preserve_identifier_case` configuration option, see below.

Case-Sensitivity of Keys in Nested Data Structures

The keys of MAPs are case-sensitive:

```
SELECT MAP(['key1'], [1]) = MAP(['KEY1'], [1]) AS equal;
```

```
false
```

The keys of UNIONs and STRUCTs are case-insensitive:

```
SELECT {'key1': 1} = {'KEY1': 1} AS equal;
```

```
true
```

```
SELECT union_value(key1 := 1) = union_value(KEY1 := 1) as equal;
```

```
true
```

Handling Conflicts In case of a conflict, when the same identifier is spelt with different cases, one will be selected randomly. For example:

```
CREATE TABLE t1 (idfield INTEGER, x INTEGER);
CREATE TABLE t2 (IdField INTEGER, y INTEGER);
INSERT INTO t1 VALUES (1, 123);
INSERT INTO t2 VALUES (1, 456);
SELECT * FROM t1 NATURAL JOIN t2;
```

idfield	x	y
1	123	456

Disabling Preserving Cases

> New In DuckDB 2.0, `preserve_identifier_case` is a VARCHAR option. Previously, it was a BOOLEAN option.

The `preserve_identifier_case` configuration option controls how non-quoted identifiers are folded. It accepts the following values:

- `'preserve_case'` (default): keep the case as written.
- `'lowercase'`: turn non-quoted identifiers into lowercase.
- `'uppercase'`: turn non-quoted identifiers into uppercase.

For backward compatibility, the Boolean values `true` and `false` are also accepted and map to `'preserve_case'` and `'lowercase'`, respectively.

For example, with `preserve_identifier_case` set to `'lowercase'`, all non-quoted identifiers are turned into lowercase:

```
SET preserve_identifier_case = 'lowercase';
CREATE TABLE tbl AS SELECT cos(pi()) AS CosineOfPi;
SELECT CosineOfPi FROM tbl;
```

cosineofpi
-1.0

With `preserve_identifier_case` set to `'uppercase'`, they are turned into uppercase, so the table above would be named `TBL` and its column `COSINEOFPI`.

> The setting is applied when a statement is parsed. Therefore, it only takes effect for statements parsed after the SET statement has been executed, e.g., in a subsequent query string.

Order Preservation

For many operations, DuckDB preserves the order of rows, similarly to data frame libraries such as Pandas.

Example

Take the following table for example:

```
CREATE TABLE tbl AS
    SELECT *
    FROM (VALUES (1, 'a'), (2, 'b'), (3, 'c')) t(x, y);

SELECT *
FROM tbl;
```

x	y
1	a
2	b
3	c

Let's take the following query that returns the rows where `x` is an odd number:

```
SELECT *
FROM tbl
WHERE x % 2 == 1;
```

x	y
1	a
3	c

Because the row `(1, 'a')` occurs before `(3, 'c')` in the original table, it is guaranteed to come before that row in this table too.

Clauses

The following clauses guarantee that the original row order is preserved:

- COPY (see Insertion Order)
- FROM with a single table

- `LIMIT`
- `OFFSET`
- `SELECT`
- `UNION ALL`
- `WHERE`
- Window functions with an empty `OVER` clause
- Common table expressions and table subqueries as long as they only contain the aforementioned components

> **Tip.** `row_number() OVER ()` allows turning the original row order into an explicit column that can be referenced in the operations that don't preserve row order by default. On materialized tables, the `rowid` pseudo-column can be used to the same effect.

The following operations **do not** guarantee that the row order is preserved:

- `FROM` with multiple tables and/or subqueries
- `JOIN`
- `UNION`
- `USING SAMPLE`
- Whole-table aggregation (the input order, that is, the order in which rows are fed into order-sensitive aggregate functions is not guaranteed unless explicitly specified in the aggregate function)
- `GROUP BY` (neither in- nor output order are guaranteed)
- `ORDER BY` (specifically, `ORDER BY` may not use a stable algorithm)
- Scalar subqueries

Insertion Order

By default, the following components preserve insertion order:

- CSV reader (`read_csv` function)
- JSON reader (`read_json` function)
- Parquet reader (`read_parquet` function)

The `preserve_insertion_order` setting governs whether this order is maintained as rows flow into a *sink*. A sink is any operation that consumes a result, namely:

- Writing to a file, for example with `COPY ... TO`
- Writing to a table, for example with `INSERT` or `CREATE TABLE ... AS SELECT`
- Returning a query result to the client

When the setting is disabled, DuckDB may reorder the rows produced by a query that has no `ORDER BY` clause before they reach the sink.

Preservation of insertion order is controlled by the `preserve_insertion_order` configuration option. This setting is `true` by default, indicating that the order should be preserved. To change this setting, use:

```
SET preserve_insertion_order = false;
```

Disabling this setting can reduce memory usage when importing or exporting datasets larger than memory. See the `preserve_insertion_order` option in the workload tuning guide for details.

PostgreSQL Compatibility

DuckDB's SQL dialect closely follows the conventions of the PostgreSQL dialect. The few exceptions to this are listed on this page.

Floating-Point Arithmetic

DuckDB and PostgreSQL handle floating-point arithmetic differently for division by zero. DuckDB conforms to the IEEE Standard for Floating-Point Arithmetic (IEEE 754) for both division by zero and operations involving infinity values. PostgreSQL returns an error for division by zero but aligns with IEEE 754 for handling infinity values. To show the differences, run the following SQL queries:

```
SELECT 1.0 / 0.0 AS x;
SELECT 0.0 / 0.0 AS x;
SELECT -1.0 / 0.0 AS x;
SELECT 'Infinity'::FLOAT / 'Infinity'::FLOAT AS x;
SELECT 1.0 / 'Infinity'::FLOAT AS x;
SELECT 'Infinity'::FLOAT - 'Infinity'::FLOAT AS x;
SELECT 'Infinity'::FLOAT - 1.0 AS x;
```

Expression	PostgreSQL	DuckDB	IEEE 754
1.0 / 0.0	error	Infinity	Infinity
0.0 / 0.0	error	NaN	NaN
-1.0 / 0.0	error	-Infinity	-Infinity
'Infinity' / 'Infinity'	NaN	NaN	NaN
1.0 / 'Infinity'	0.0	0.0	0.0
'Infinity' - 'Infinity'	NaN	NaN	NaN
'Infinity' - 1.0	Infinity	Infinity	Infinity

Division on Integers

When computing division on integers, PostgreSQL performs integer division, while DuckDB performs float division:

```
SELECT 1 / 2 AS x;
```

PostgreSQL returns `0`, while DuckDB returns `0.5`.

To perform integer division in DuckDB, use the `//` operator:

```
SELECT 1 // 2 AS x;
```

This returns `0`.

`UNION` of Boolean and Integer Values

The following query fails in PostgreSQL but successfully completes in DuckDB:

```
SELECT true AS x
UNION
SELECT 2;
```

PostgreSQL returns an error:

```
ERROR:  UNION types boolean and integer cannot be matched
```

DuckDB performs an enforced cast, therefore, it completes the query and returns the following:

x
1

x
2

Implicit Casting on Equality Checks

DuckDB performs implicit casting on equality checks, e.g., converting strings to numeric and boolean values. Therefore, there are several instances, where PostgreSQL throws an error while DuckDB successfully computes the result:

Expression	PostgreSQL	DuckDB
'1.1' = 1	error	true
'1.1' = 1.1	true	true
1 = 1.1	false	false
true = 'true'	true	true
true = 1	error	true
'true' = 1	error	error

Case Sensitivity for Quoted Identifiers

PostgreSQL is case-insensitive. The way PostgreSQL achieves case insensitivity is by lowercasing unquoted identifiers within SQL, whereas quoting preserves case, e.g., the following command creates a table named `mytable` but tries to query for `MyTaBLe` because quotes preserve the case.

```
CREATE TABLE MyTaBLe (x INTEGER);
SELECT * FROM "MyTaBLe";
```

```
ERROR:  relation "MyTaBLe" does not exist
```

PostgreSQL does not only treat quoted identifiers as case-sensitive; it treats all identifiers as case-sensitive, e.g., this also does not work:

```
CREATE TABLE "PreservedCase" (x INTEGER);
SELECT * FROM PreservedCase;
```

```
ERROR:  relation "preservedcase" does not exist
```

Therefore, case-insensitivity in PostgreSQL only works if you never use quoted identifiers with different cases.

For DuckDB, this behavior was problematic when interfacing with other tools (e.g., Parquet, Pandas) that are case-sensitive by default – since all identifiers would be lowercased all the time. Therefore, DuckDB achieves case insensitivity by making identifiers fully case insensitive throughout the system but *preserving their case*.

In DuckDB, the scripts above complete successfully:

```
CREATE TABLE MyTaBLe (x INTEGER);
SELECT * FROM "MyTaBLe";
CREATE TABLE "PreservedCase" (x INTEGER);
SELECT * FROM PreservedCase;
SELECT table_name FROM duckdb_tables();
```

table_name
MyTaBLe
PreservedCase

PostgreSQL's behavior of lowercasing identifiers is accessible using the `preserve_identifier_case` option:

```
SET preserve_identifier_case = 'lowercase';
CREATE TABLE MyTaBLe (x INTEGER);
SELECT table_name FROM duckdb_tables();
```

table_name
mytable

However, the case insensitive matching in the system for identifiers cannot be turned off.

Using Double Equality Sign for Comparison

DuckDB supports both = and == for equality comparison, while PostgreSQL only supports =.

```
SELECT 1 == 1 AS t;
```

DuckDB returns `true`, while PostgreSQL returns:

```
postgres=# SELECT 1 == 1 AS t;
ERROR:  operator does not exist: integer == integer
LINE 1: SELECT 1 == 1 AS t;
```

Note that the use of == is not encouraged due to its limited portability.

Vacuuming Tables

In PostgreSQL, the VACUUM statement garbage collects tables and analyzes tables. In DuckDB, the VACUUM statement is only used to rebuild statistics. For instruction on reclaiming space, refer to the "Reclaiming space" page.

Strings

Since version 1.3.0, DuckDB escapes characters such as `'` in strings serialized in nested data structures. PostgreSQL does not do this.

For an example, run:

```
SELECT ARRAY[''''];
```

PostgreSQL returns:

```
{'}
```

DuckDB returns:

```
['\'']
```

Functions

`regexp_extract` Function

Unlike PostgreSQL's `regexp_substr` function, DuckDB's `regexp_extract` returns empty strings instead of NULLs when there is no match.

`to_date` Function

DuckDB does not support the `to_date` PostgreSQL date formatting function. Instead, please use the `strptime` function.

`date_part` Function

Most parts extracted by the `date_part` function are returned as integers. Since there are no infinite integer values in DuckDB, `NULL`s are returned for infinite timestamps.

`pg_size_pretty` Function

DuckDB's `pg_size_pretty` function (an alias of `format_bytes`) formats sizes using binary (IEC) units such as `KiB` and `MiB`, whereas PostgreSQL uses `kB` and `MB`. For example, `pg_size_pretty(1024)` returns `1.0 KiB` in DuckDB and `1024 bytes` in PostgreSQL, and `pg_size_pretty(1048576)` returns `1.0 MiB` in DuckDB and `1024 kB` in PostgreSQL.

Resolution of Type Names in the Schema

For `CREATE TABLE` statements, DuckDB attempts to resolve type names in the schema where a table is created. For example:

```
CREATE SCHEMA myschema;
CREATE TYPE myschema.mytype AS ENUM ('as', 'df');
CREATE TABLE myschema.mytable (v mytype);
```

PostgreSQL returns an error on the last statement:

```
ERROR:  type "mytype" does not exist
LINE 1: CREATE TABLE myschema.mytable (v mytype);
```

DuckDB runs the statement and creates the table successfully, confirmed by the following query:

```
DESCRIBE myschema.mytable;
```

column_name	column_type	null	key	default	extra
v	ENUM('as', 'df')	YES	NULL	NULL	NULL

Exploiting Functional Dependencies for `GROUP BY`

PostgreSQL can exploit functional dependencies, such as `i -> j` in the following query:

```
CREATE TABLE tbl (i INTEGER, j INTEGER, PRIMARY KEY (i));
SELECT j
FROM tbl
GROUP BY i;
```

PostgreSQL runs the query.

DuckDB fails:

```
Binder Error:
column "j" must appear in the GROUP BY clause or must be part of an aggregate function.
Either add it to the GROUP BY list, or use "ANY_VALUE(j)" if the exact value of "j" is not important.
```

To work around this, add the other attributes or use the `GROUP BY ALL` clause.

Behavior of Regular Expression Match Operators

PostgreSQL supports the POSIX regular expression matching operators `~` (case-sensitive partial regex matching) and `~*` (case-insensitive partial regex matching) as well as their negated variants, `!~` and `!~*`, respectively.

By default, DuckDB follows the same semantics: `~` maps to `regexp_matches` (case-sensitive partial matching), `~*` performs case-insensitive partial matching, and `!~` and `!~*` are their negated variants. The results therefore match PostgreSQL:

Expression	PostgreSQL	DuckDB
`'aaa' ~ '(a|b)'`	true	true
`'AAA' ~* '(a|b)'`	true	true
`'aaa' !~ '(a|b)'`	false	false
`'AAA' !~* '(a|b)'`	false	false

The legacy behavior, in which `~` mapped to `regexp_full_match` and `~*` and `!~*` were unsupported, can be restored using the deprecated `regex_match_operator_semantics` setting with the value `full`.

SQL Quirks

Like all programming languages and libraries, DuckDB has its share of idiosyncrasies and inconsistencies.
Some are vestiges of our feathered friend's evolution; others are inevitable because we strive to adhere to the SQL Standard and specifically to PostgreSQL's dialect (see the "PostgreSQL Compatibility" page for exceptions). The rest may simply come down to different preferences, or we may even agree on what *should* be done but just haven't gotten around to it yet.

Acknowledging these quirks is the best we can do, which is why we have compiled below a list of examples.

Aggregating Empty Groups

On empty groups, the aggregate functions `sum`, `list`, and `string_agg` all return `NULL` instead of `0`, `[]` and `''`, respectively. This is dictated by the SQL Standard and obeyed by all SQL implementations we know. This behavior is inherited by the list aggregate `list_sum`, but not by the DuckDB original `list_dot_product` which returns `0` on empty lists.

0 vs. 1-Based Indexing

To comply with standard SQL, one-based indexing is used almost everywhere, e.g., array and string indexing and slicing, and window functions (`row_number`, `rank`, `dense_rank`). However, similarly to PostgreSQL, JSON features use a zero-based indexing.

Types

`UINT8` vs. `INT8`

`UINT8` and `INT8` are aliases to integer types of different widths:

- `UINT8` corresponds to `UTINYINT` because it's an *8-bit* unsigned integer
- `INT8` corresponds to `BIGINT` because it's an *8-byte* signed integer

Explanation: the `n` in the numeric type `INTn` and `UINTn` denote the width of the number in either bytes or bits. `INT1`, `INT2`, `INT4` correspond to the number of bytes, while `INT16`, `INT32` and `INT64` correspond to the number of bits. The same applies to `UINT` values. However, the value `n = 8` is a valid choice for both the number of bits and bytes. For unsigned values, `UINT8` corresponds to `UTINYINT` (8 bits). For signed values, `INT8` corresponds to `BIGINT` (8 bytes).

Expressions

Results That May Surprise You

Expression	Result	Note
`-2^2`	`4.0`	PostgreSQL compatibility means the unary minus has higher precedence than the exponentiation operator. Use additional parentheses, e.g., `-(2^2)` or the pow function, e.g., `-pow(2, 2)`, to avoid mistakes.
`'t' = true`	`true`	Compatible with PostgreSQL.
`1 = '1'`	`true`	Compatible with PostgreSQL.
`1 = ' 1'`	`true`	Compatible with PostgreSQL.
`1 = '01'`	`true`	Compatible with PostgreSQL.
`1 = ' 01 '`	`true`	Compatible with PostgreSQL.
`1 = true`	`true`	Not compatible with PostgreSQL.
`1 = '1.1'`	`true`	Not compatible with PostgreSQL.
`1 IN (0, NULL)`	`NULL`	Makes sense if you think of the `NULL`s in the input and output as `UNKNOWN`.
`1 in [0, NULL]`	`false`	
`concat('abc', NULL)`	`abc`	Compatible with PostgreSQL. `list_concat` behaves similarly.
`'abc' || NULL`	`NULL`	

NaN Values

`'NaN'::FLOAT = 'NaN'::FLOAT` and `'NaN'::FLOAT > 3` violate IEEE-754 but mean floating point data types have a total order, like all other data types (beware the consequences for `greatest` / `least`).

age Function

`age(x)` is `current_date - x` instead of `current_timestamp - x`. Another quirk inherited from PostgreSQL.

Extract Functions

`list_extract` / `map_extract_value` return `NULL` on non-existing keys (`map_extract` returns an empty list). `struct_extract` throws an error because keys of structs are like columns.

Clauses

Automatic Column Deduplication in SELECT

Column names are deduplicated with the first occurrence shadowing the others:

```
CREATE TABLE tbl AS SELECT 1 AS a;
SELECT a FROM (SELECT *, 2 AS a FROM tbl);
```

a
1

Case Insensitivity for SELECTing Columns

Due to case-insensitivity, it's not possible to use `SELECT a FROM 'file.parquet'` when a column called A appears before the desired column a in `file.parquet`.

`USING SAMPLE`

The `USING SAMPLE` clause is syntactically placed after the `WHERE` and `GROUP BY` clauses (same as the `LIMIT` clause) but is semantically applied before both (unlike the `LIMIT` clause).

Implicit Conversion of Identifiers in Table Function Arguments

DuckDB historically allowed a bare, unbound identifier to be used as a table function argument, implicitly converting it to a string literal. For example, `glob(no_such_file.csv)` was treated as `glob('no_such_file.csv')`.

> **Deprecated.** As of DuckDB 2.0, this implicit conversion is deprecated. By default DuckDB still converts the identifier but logs a warning; use an explicit string literal instead, for example `glob('no_such_file.csv')`.

The behavior is controlled by the `table_function_identifier_conversion` setting:

- `DEFAULT`: convert the identifier but log a warning.
- `ENABLE_IMPLICIT_STRING`: convert silently (the previous behavior).
- `DISABLE_IMPLICIT_STRING`: raise a binder error instead of converting.

```
SET table_function_identifier_conversion = 'DISABLE_IMPLICIT_STRING';
```

PEG Parser

> New Starting with DuckDB 2.0, the PEG parser is the default parser. In earlier versions it was an experimental, opt-in feature enabled with `enable_peg_parser()`, which is no longer required or available.

DuckDB uses a parser based on PEG (parsing expression grammars). It replaces the parser front end derived from PostgreSQL that earlier versions used. The PEG parser enables better suggestions, improved error messages with precise source locations, and allows extensions to extend the grammar.

The PEG grammar was first introduced in DuckDB 1.2 to power autocomplete in the command line client. DuckDB 1.5 shipped a complete but experimental version, and DuckDB 2.0 made it the default.

Switching parsers does not change the SQL dialect. DuckDB 2.0 replaced the parser implementation and rewrote its grammar, but the dialect itself and the rest of the query pipeline, including the binder, continue to operate on the same internal syntax tree.

Suggestions

The PEG parser is used to generate suggestions. You can cycle through the options using TAB:

```
«animals_db» D FROM ducks WHERE habitat IS
IS           ISNULL       ILIKE        ⍰IN⍰          INTERSECT    LIKE
```

Extensible Grammar

Because the grammar is part of the parser, extensions can extend the tokenizer, add grammar rules, and register custom matchers while reusing the rest of the DuckDB grammar. This makes it possible to combine the syntax of multiple extensions, and you can expect extensions that expose entirely new SQL syntax.

Performance

The parser uses packrat parsing, which memoizes intermediate results to avoid exponential backtracking on deeply nested expressions. As a tradeoff, parsing uses additional memory and has a slight performance overhead. This is in the range of milliseconds and is thus negligible for analytical queries.

Checking a Query

The `check_peg_parser` table function is a testing utility provided by the `autocomplete` extension. It runs the PEG grammar matcher on a query string without executing the query. It returns a single `BOOLEAN` column, `success`, and an empty result:

```
FROM check_peg_parser('SELECT 42');
```

```
┌─────────┐
│ success │
│ boolean │
└─────────┘
  0 rows
```

Note that `check_peg_parser` does not reliably report parse failures: in DuckDB 2.0.0, it also returns an empty result for invalid queries such as `SELEC 42`. To check whether a query parses without binding or executing it, use the `json_serialize_sql` function of the `json` extension, which returns a JSON object with `"error": true` for queries that fail to parse:

```sql
SELECT json_serialize_sql('SELEC 42')::JSON->>'error' AS parse_error;
```

parse_error
varchar
true

Legacy Parser

> New The `legacy_parser` extension was introduced in DuckDB 2.0.

For exceptional cases, DuckDB 2.0 can fall back to the legacy PostgreSQL-style parser through the `legacy_parser` extension. Setting the following option autoloads the extension and switches from the default PEG parser to the legacy parser:

```sql
SET enable_legacy_parser = true;
```

The legacy parser is intended as a bridge solution, for example to check a query on both parsers and determine whether a given behavior is tied to the parser or transformer rather than to downstream code. It provides a best-effort legacy experience and preserves several DuckDB 1.5 quirks:

- `~` and `!~` perform full regular expression matches, whereas DuckDB 2.0 defaults to partial matches.
- The `!` operator binds less tightly: `2 * 3!` returns `720` (`(2 * 3)!`), whereas DuckDB 2.0 returns `12`.
- Rendered SQL qualifies built-in functions, for example `main.struct_pack(x := 1)` in view definitions.
- `ORDER BY true` is accepted, whereas DuckDB 2.0 rejects a non-integer literal.
- Column constraints on `ALTER TABLE ... ADD COLUMN`, such as `UNIQUE`, are rejected by the transformer.
- The `preserve_identifier_case = 'uppercase'` option is a DuckDB 2.0 feature; the legacy grammar treats it as `'preserve_case'`.

> **Warning.** Newer and more powerful syntax, such as `INSTALL AND LOAD`, is rejected by the legacy parser.

Further Reading

- "A Preview of DuckDB v2.0" blog post
- "DuckDB v2.0: Your Database Deserves a Better Parser" blog post
- "Runtime-Extensible SQL Parsers Using PEG" blog post
- "Runtime-Extensible Parsers" (CIDR 2025 paper) by Hannes Mühleisen and Mark Raasveldt

Samples

Samples are used to randomly select a subset of a dataset.

Examples

Select a sample of exactly 5 rows from `tbl` using `reservoir` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 5;
```

Select a sample of *approximately* 10% of the table using `system` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 10%;
```

> **Warning.** By default, when you specify a percentage, each *vector* is included in the sample with that probability. If your table contains fewer than ~10k rows, it makes sense to specify the `bernoulli` sampling option instead, which applies the probability to each row independently. Even then, you'll sometimes get more and sometimes less than the specified percentage of the number of rows, but it is much less likely that you get no rows at all. To get exactly 10% of rows (up to rounding), you must use the `reservoir` sampling option.

Select a sample of *approximately* 10% of the table using `bernoulli` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 10 PERCENT (bernoulli);
```

Select a sample of *exactly* 10% (up to rounding) of the table using `reservoir` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 10 PERCENT (reservoir);
```

Select a sample of *exactly* 50 rows of the table using reservoir sampling with a fixed seed (100):

```
SELECT *
FROM tbl
USING SAMPLE reservoir(50 ROWS)
REPEATABLE (100);
```

Select a sample of *approximately* 20% of the table using `system` sampling with a fixed seed (377):

```
SELECT *
FROM tbl
USING SAMPLE 20% (system, 377);
```

Select a sample of *approximately* 20% of `tbl` **before** the join with `tbl2`:

```
SELECT *
FROM tbl TABLESAMPLE reservoir(20%), tbl2
WHERE tbl.i = tbl2.i;
```

Select a sample of *approximately* 20% of `tbl` **after** the join with `tbl2`:

```sql
SELECT *
FROM tbl, tbl2
WHERE tbl.i = tbl2.i
USING SAMPLE reservoir(20%);
```

Syntax

Samples allow you to randomly extract a subset of a dataset. Samples are useful for exploring a dataset faster, as often you might not be interested in the exact answers to queries, but only in rough indications of what the data looks like and what is in the data. Samples allow you to get approximate answers to queries faster, as they reduce the amount of data that needs to pass through the query engine.

DuckDB supports three different types of sampling methods: `reservoir`, `bernoulli` and `system`. By default, DuckDB uses `reservoir` sampling when an exact number of rows is sampled, and `system` sampling when a percentage is specified. The sampling methods are described in detail below.

Samples require a *sample size*, which is an indication of how many elements will be sampled from the total population. Samples can either be given as a percentage (`10%` or `10 PERCENT`) or as a fixed number of rows (`10` or `10 ROWS`). All three sampling methods support sampling over a percentage, but **only** reservoir sampling supports sampling a fixed number of rows.

Samples are probabilistic, that is to say, samples can be different between runs *unless* the seed is specifically specified. Specifying the seed *only* guarantees that the sample is the same if multi-threading is not enabled (i.e., `SET threads = 1`{:.language-sql .highlight}). In the case of multiple threads running over a sample, samples are not necessarily consistent even with a fixed seed.

Reproducible Samples

Sampling is probabilistic, so a query returns a different sample on each run unless the seed is fixed. The seed is a non-negative integer and can be specified in two equivalent ways:

- With the `REPEATABLE` keyword after the method form, e.g., `USING SAMPLE reservoir(50 ROWS) REPEATABLE (100)`.
- As the second value in the parenthesized method form, e.g., `USING SAMPLE 20% (system, 377)`.

As noted above, fixing the seed only guarantees a reproducible sample when DuckDB runs single-threaded (`SET threads = 1`{:.language-sql .highlight}). With multiple threads, samples are not necessarily consistent even with a fixed seed.

Sampling Methods

`reservoir`

Reservoir sampling is a stream sampling technique that selects a random sample by keeping a *reservoir* of size equal to the sample size, and randomly replacing elements as more elements come in. Reservoir sampling allows us to specify *exactly* how many elements we want in the resulting sample (by selecting the size of the reservoir). As a result, reservoir sampling *always* outputs the same amount of elements, unlike system and bernoulli sampling.

Reservoir sampling is only recommended for small sample sizes, and is not recommended for use with percentages. That is because reservoir sampling needs to materialize the entire sample and randomly replace tuples within the materialized sample. The larger the sample size, the higher the performance hit incurred by this process.

Reservoir sampling also incurs an additional performance penalty when multi-processing is used, since the reservoir is to be shared amongst the different threads to ensure unbiased sampling. This is not a big problem when the reservoir is very small, but becomes costly when the sample is large.

> **Best practice.** Avoid using reservoir sampling with large sample sizes if possible. Reservoir sampling requires the entire sample to be materialized in memory.

`bernoulli`

Bernoulli sampling can only be used when a sampling percentage is specified. It is rather straightforward: every row in the underlying table is included with a chance equal to the specified percentage. As a result, bernoulli sampling can return a different number of tuples even if the same percentage is specified. The *expected* number of rows is equal to the specified percentage of the table, but there will be some *variance*.

Because bernoulli sampling is completely independent (there is no shared state), there is no penalty for using bernoulli sampling together with multiple threads.

`system`

System sampling is a variant of bernoulli sampling with one crucial difference: every *vector* is included with a chance equal to the sampling percentage. This is a form of cluster sampling. System sampling is more efficient than bernoulli sampling, as no per-tuple selections have to be performed.

The *expected* number of rows is still equal to the specified percentage of the table, but the *variance* is `vectorSize` times higher. As such, system sampling is not suitable for datasets with fewer than ~10k rows, where it can happen that all rows will be filtered out, or all the data will be included, even when you ask for `50 PERCENT`.

Table Samples

The `TABLESAMPLE` and `USING SAMPLE` clauses use the same internal syntax (sample size, method, and optional seed), but they differ in where they are written and what they sample. The `TABLESAMPLE` clause is written directly after a table expression, following its alias if one is present, and samples only that table. The `USING SAMPLE` clause is written at the end of the `SELECT` statement body and samples the result of the entire `FROM` clause, after all joins have been resolved. This distinction is relevant when there are joins present in the query plan. `TABLESAMPLE` can be applied to any table expression in the `FROM` clause, including tables that participate in a join.

The `TABLESAMPLE` clause is essentially equivalent to creating a subquery with the `USING SAMPLE` clause, i.e., the following two queries are identical:

Sample 20% of `tbl` **before** the join:

```
SELECT *
FROM
    tbl TABLESAMPLE reservoir(20%),
    tbl2
WHERE tbl.i = tbl2.i;
```

Sample 20% of `tbl` **before** the join:

```
SELECT *
FROM
    (SELECT * FROM tbl USING SAMPLE reservoir(20%)) tbl,
    tbl2
WHERE tbl.i = tbl2.i;
```

Sample 20% **after** the join (i.e., sample 20% of the join result):

```
SELECT *
FROM tbl, tbl2
WHERE tbl.i = tbl2.i
USING SAMPLE reservoir(20%);
```

Configuration

Configuration

DuckDB has a number of configuration options that can be used to change the behavior of the system.

The configuration options can be set using either the SET statement or the PRAGMA statement. They can be reset to their original values using the RESET statement.

The values of configuration options can be queried via the `current_setting()` scalar function or using the `duckdb_settings()` table function. For example:

```
SELECT current_setting('memory_limit') AS memlimit;
```

Or:

```
SELECT value AS memlimit
FROM duckdb_settings()
WHERE name = 'memory_limit';
```

Examples

Set the memory limit of the system to 10 GB.

```
SET memory_limit = '10GB';
```

Configure the system to use 1 thread.

```
SET threads TO 1;
```

Turn logging on and set the logging level to `debug`. For additional details on logging levels, see Log Level.

In the CLI client, the default log storage (`shell_log_storage`) prints logs to the console but cannot be queried, so we switch to the `memory` log storage first:

```
SET logging_storage = 'memory';
SET enable_logging = true;
SET logging_level = 'debug';
```

Write a single log message with the `debug` level and a `connection` scope:

```
SELECT write_log('A new client has connected.', level := 'debug', scope := 'connection');
```

Write a single log message with a `debug` level and a `connection` scope and a custom `log_type`:

```
SELECT write_log(
        'A new duck has connected to the lake.',
        level := 'debug',
        scope := 'connection',
        log_type := 'duckdb.docs.example.quack'
    );
```

Check logs with the DEBUG log level:

```
SELECT * FROM duckdb_logs WHERE log_level = 'DEBUG';
```

Check logs with the `QueryLog` type:

```
SELECT * FROM duckdb_logs WHERE type = 'QueryLog';
```

Check current logging settings:

```sql
SELECT * FROM duckdb_settings() WHERE name LIKE '%logging%';
```

Enable printing of a progress bar during long-running queries:

```sql
SET enable_progress_bar = true;
```

Set the default null order to `NULLS LAST`:

```sql
SET default_null_order = 'nulls_last';
```

Return the current value of a specific setting:

```sql
SELECT current_setting('threads') AS threads;
```

threads
10

Query a specific setting:

```sql
SELECT *
FROM duckdb_settings()
WHERE name = 'threads';
```

name	value	description	input_type	scope
threads	1	The number of total threads used by the system.	BIGINT	GLOBAL

Show a list of all available settings:

```sql
SELECT *
FROM duckdb_settings();
```

Reset the memory limit of the system back to the default:

```sql
RESET memory_limit;
```

Secrets Manager

DuckDB has a Secrets manager, which provides a unified user interface for secrets across all backends (e.g., AWS S3) that use them.

Schema and Search Path

The `schema` and `search_path` configuration options both control the *search path*: the ordered list of schemas that DuckDB consults to resolve unqualified table, view, and function names. Despite the difference in their descriptions, both options operate on this same search path:

- `search_path` sets the entire search path as a comma-separated list of entries.
- `schema` is a convenience option that sets the search path to a single entry.

Each entry in the search path is a schema. An entry may be qualified with a catalog (attached database) name in the form `<catalog>.<schema>` to select a schema in a specific database; an unqualified entry refers to a schema in the current catalog. This is why the descriptions mention both schemas and catalogs: the path is a list of schemas, whose entries may optionally be catalog-qualified.

Asynchronous I/O Settings

> New Asynchronous I/O, along with the settings that tune it, was introduced in DuckDB 2.0.

The `async_threads` and `read_ahead_depth` settings control the size of the asynchronous thread pool and the read-ahead budget used when reading from object storage and HTTP servers. They are listed in the reference below and documented in more detail on the I/O page, together with the related `http_client_pool_capacity` setting.

Configuration Reference

Configuration options come with different default scopes: `GLOBAL` and `LOCAL`. Below is a list of all available configuration options by scope.

Global Configuration Options

Name	Description	Type	Default value
`Calendar`	The current calendar	`VARCHAR`	System (locale) calendar
`TimeZone`	The current time zone	`VARCHAR`	System (locale) timezone
`access_mode`	Access mode of the database (`AUTOMATIC`, `READ_ONLY` or `READ_WRITE`)	`VARCHAR`	`automatic`
`allocator_background_threads`	Whether to enable the allocator background threads.	`BOOLEAN`	`false`
`allocator_bulk_deallocation_flush_threshold`	If a bulk deallocation larger than this occurs, flush outstanding allocations.	`VARCHAR`	`512.0 MiB`
`allocator_flush_threshold`	Peak allocation threshold at which to flush the allocator after completing a task.	`VARCHAR`	`134217728B`
`allow_asterisks_in_http_paths`	Allow '*' character in URLs users can query	`BOOLEAN`	`false`
`allow_community_extensions`	Allow loading extensions build by the community.	`BOOLEAN`	`true`
`allow_extension_repositories`	Whether custom trusted extension repositories are 'allowed', 'forbidden' (which also distrusts existing repositories) or 'undecided' (the default: blocks adding new repositories, but keeps trusting existing ones). While the database is running the setting can only move from 'undecided' to 'allowed' or 'forbidden', or from 'allowed' to 'forbidden'.	`VARCHAR`	`undecided`
`allow_extensions_metadata_mismatch`	Allow to load extensions with no compatible metadata.	`BOOLEAN`	`false`
`allow_parser_override_extension`	Allow extensions to override the current parser.	`VARCHAR`	`DEFAULT`
`allow_persistent_secrets`	Allow the creation of persistent secrets that are stored and loaded on restarts.	`BOOLEAN`	`true`
`allow_unredacted_secrets`	Allow printing unredacted secrets.	`BOOLEAN`	`false`

Name	Description	Type	Default value
allow_unsigned_extensions	Allow to load extensions with invalid or missing signatures.	BOOLEAN	false
allowed_configs	List of configuration options that are ALWAYS allowed to be changed, even when lock_configuration is true.	VARCHAR[]	[]
allowed_directories	List of directories, and directory prefixes, that are ALWAYS allowed to be queried, even when enable_external_access is false.	VARCHAR[]	[]
allowed_paths	List of files that are ALWAYS allowed to be queried, even when enable_external_access is false.	VARCHAR[]	[]
approximate_join_order_threshold	The minimum number of tables in a join to determine the optimal join order approximately instead of exactly.	UBIGINT	12
arrow_large_buffer_size	Whether Arrow buffers for strings, blobs, UUIDs and bits should be exported using large buffers.	BOOLEAN	false
arrow_lossless_conversion	Whenever a DuckDB type does not have a clear native or canonical extension match in Arrow, export the types with a duckdb.type_name extension name.	BOOLEAN	false
arrow_output_list_view	Whether the export to the Arrow format should use ListView as the physical layout for LIST columns.	BOOLEAN	false
arrow_output_version	Whether strings should be produced by DuckDB in Utf8View format instead of Utf8.	VARCHAR	1.0
asof_loop_join_threshold	The maximum number of rows we need on the left side of an ASOF join to use a nested loop join.	UBIGINT	64
async_threads	The number of total async threads used by the system for tasks like I/O.	BIGINT	4 × # CPU cores (max. 256)
auto_checkpoint_skip_wal_threshold	The estimated WAL write size at which point we will skip writing to the WAL and only checkpoint. Skipping writing to the WAL means concurrent commits are blocked while the checkpoint is happening.	UBIGINT	100000
auto_fallback_to_full_download	Allows automatically falling back to full file downloads when possible.	BOOLEAN	true
autoinstall_extension_repository	Overrides the custom endpoint for extension installation on autoloading. Falls back to custom_extension_repository, if not specified.	VARCHAR	
autoinstall_known_extensions	Whether known extensions are allowed to be automatically installed when a query depends on them.	BOOLEAN	true
autoload_known_extensions	Whether known extensions are allowed to be automatically loaded when a query depends on them.	BOOLEAN	true
binary_as_string	In Parquet files, interpret binary data as a string.	BOOLEAN	false
block_allocator_memory	Physical memory that the block allocator is allowed to use. This memory is allocated once, never freed, and cannot be reduced.	VARCHAR	0 bytes
ca_cert_file	Path to a custom certificate file for self-signed certificates.	VARCHAR	
cache_local_files	Whether the external file cache also caches local files. Remote files are always cached.	BOOLEAN	false

Name	Description	Type	Default value
`catalog_error_max_schemas`	The maximum number of schemas in the catalog that the system will scan to generate the "did you mean..." error message.	`UBIGINT`	`100`
`checkpoint_on_detach`	Override checkpoint behavior when detaching a database. ENABLED requests a checkpoint, but the checkpoint does not occur if another connection still references the database. DISABLED never checkpoints, DEFAULT defers to the global checkpoint_on_shutdown setting.	`VARCHAR`	`DEFAULT`
`checkpoint_threshold`, `wal_autocheckpoint`	The WAL size threshold at which to automatically trigger a checkpoint, e.g., 1GB.	`VARCHAR`	`16.0 MiB`
`current_transaction_invalidation_policy`	Which types of exceptions invalidate the database for the current transaction.	`VARCHAR`	`STANDARD_POLICY`
`custom_extension_repository`	Overrides the custom endpoint for extension installation.	`VARCHAR`	
`custom_user_agent`	HTTP request metadata that can be set by callers/wrappers to tag their own traffic when DuckDB makes a request.	`VARCHAR`	
`default_block_size`	The default block size for new duckdb database files. New means that the file is being created and does not yet exist.	`UBIGINT`	`262144`
`default_collation`	The collation setting used when none is specified.	`VARCHAR`	
`default_null_order`, `null_order`	The NULL ordering used when none is specified (`NULLS_FIRST` or `NULLS_LAST`).	`VARCHAR`	`NULLS_LAST`
`default_order`	The order type used when none is specified (`ASC` or `DESC`).	`VARCHAR`	`ASCENDING`
`default_secret_storage`	Allows switching the default storage for secrets.	`VARCHAR`	`local_file`
`default_transaction_invalidation_policy`	When to invalidate transactions when errors occur. SYNTACTIC_ERRORS_DO_NOT_INVALIDATE means that parser and binder exceptions do not invalidate. `ALL_ERRORS_INVALIDATE_TRANSACTION` means all errors invalidate a transaction.	`VARCHAR`	`ALL_ERRORS_INVALIDATE_TRANSACTION`
`delim_join_as_cte`	Rewrite delim joins to materialized CTEs during dependent join flattening.	`BOOLEAN`	`true`
`dialect_compatibility_mode`	Enable SQL dialect compatibility for a certain engine, e.g., `SET dialect_compatibility_mode='spark'`.	`VARCHAR`	`NONE`
`disable_database_invalidation`	Disables invalidating the database instance when encountering a fatal error. Should be used with great care, as DuckDB cannot guarantee correct behavior after a fatal error.	`BOOLEAN`	`false`
`disable_parquet_prefetching`	Disable the prefetching mechanism in Parquet	`BOOLEAN`	`false`
`disable_timestamptz_casts`	Disable casting from timestamp to timestamptz.	`BOOLEAN`	`false`
`disabled_compression_methods`	Disable a specific, comma-separated set of compression methods.	`VARCHAR`	
`disabled_filesystems`	Disable specific file systems, thereby preventing their access, e.g., LocalFileSystem.	`VARCHAR`	

Name	Description	Type	Default value
disabled_log_types	Sets the list of disabled log types.	VARCHAR	
disabled_optimizers	Disable a specific, comma-separated set of optimizers.	VARCHAR	
duckdb_api	Specifies which client API/binding opened the database connection, e.g., cpp, capi, cli. The value gets folded into the HTTP User-Agent string DuckDB sends on any HTTP request.	VARCHAR	cli
dynamic_or_filter_threshold	The maximum amount of OR filters that are generated dynamically from a hash join.	UBIGINT	50
enable_external_access	Allow the database to access external state, e.g., through loading/installing modules, COPY TO/FROM, CSV readers, pandas replacement scans, etc.	BOOLEAN	true
enable_external_file_cache	Allow the database to cache external files like Parquet in memory.	BOOLEAN	true
enable_fsst_vectors	Allow scans on FSST compressed segments to emit compressed vectors to utilize late decompression.	BOOLEAN	false
enable_geoparquet_conversion	Attempt to decode/encode geometry data in/as GeoParquet files if the spatial extension is present.	BOOLEAN	true
enable_global_s3_configuration	Automatically fetch AWS credentials from environment variables.	BOOLEAN	true
enable_http_metadata_cache	Whether or not the global http metadata is used to cache HTTP metadata.	BOOLEAN	false
enable_logging	Enables the logger.	BOOLEAN	false (true in the CLI)
enable_macro_dependencies	Enables created MACROs to create dependencies on the referenced objects, such as tables.	BOOLEAN	false
enable_optimistic_write	Whether or not to optimistically write large appends to disk before committing. Disable this to keep bulk appends in memory, e.g., for in-memory benchmarks.	BOOLEAN	true
enable_optimizer	Whether the query optimizer is enabled.	BOOLEAN	true
enable_server_cert_verification	Enable server side certificate verification.	BOOLEAN	false
enable_view_dependencies	Enable created VIEWs to create dependencies on the referenced objects, such as tables.	BOOLEAN	false
enabled_log_types	Sets the list of enabled log types.	VARCHAR	
error_on_division_by_zero	Throw an error instead of returning NULL when dividing by zero.	BOOLEAN	true
errors_as_json	Output error messages as structured JSON instead of as a raw string.	BOOLEAN	false
experimental_metadata_reuse	EXPERIMENTAL: Re-use row group and table metadata when checkpointing.	BOOLEAN	true
explain_output	Configure the output of EXPLAIN statements (ALL, OPTIMIZED_ONLY, PHYSICAL_ONLY).	VARCHAR	PHYSICAL_ONLY
extension_directories	Set the directories to store extensions in.	VARCHAR[]	[]

Name	Description	Type	Default value
extension_repository_directory	Set the directory in which trusted extension repositories are stored. This is the trust anchor for user-provided repositories, so while signature checking is enabled (allow_unsigned_extensions=false) it can only be set at startup, not while the database is running.	VARCHAR	
external_file_cache_local_max_block_size	Maximum size in bytes of an external file cache block for local (non-remote) files. Larger reads are split into blocks of this size, which are fetched in parallel.	UBIGINT	16384
external_file_cache_remote_max_block_size	Maximum size in bytes of an external file cache block for remote files (e.g., HTTP/S3). Larger reads are split into blocks of this size, which are fetched in parallel.	UBIGINT	2097152
external_file_cache_remote_min_block_size	Reads of remote files are rounded out to aligned blocks of this size, so tiny reads do not each cost a request.	UBIGINT	4096
external_file_cache_request_sizing	How reads of cached files are sized: AUTO lets each reader choose, GRID reads the aligned blocks of the cache block size around each read, EXACT reads exactly the requested bytes. Reads of files that are not cached are always exact.	VARCHAR	AUTO
external_file_cache_spill	Whether evicted external file cache blocks of remote files spill to the temporary directory instead of being dropped, so that they are re-read from there rather than re-fetched from the source.	BOOLEAN	false
external_threads	The number of external threads that work on DuckDB tasks.	UBIGINT	1
extra_http_headers	Extra HTTP headers added to every request	MAP(VARCHAR, VARCHAR)	{}
file_search_path	A comma separated list of directories to search for input files.	VARCHAR	
force_column_metadata_reuse	Force re-use of row group metadata on a column-level when checkpointing on older storage versions 6 and 7. This breaks storage backward-compatibility with older DuckDB versions.	BOOLEAN	false
force_compression	Forces the usage of a specific compression method.	VARCHAR	auto
force_download_threshold	Forces upfront download of files smaller than the given size in bytes	UBIGINT	0
force_download	Forces upfront download of file	BOOLEAN	false
fsync_mode	How to synchronize database file changes to stable storage (STANDARD, NONE or FULL). NONE may cause database corruption after an operating system or hardware crash. FULL uses fullfsync on macOS.	VARCHAR	STANDARD
geometry_minimum_shredding_size	Minimum size of a row group to enable GEOMETRY shredding. Set to -1 to disable shredding entirely. Defaults to 1/4th of a row group.	BIGINT	30000
home_directory	Sets the home directory of the database instance.	VARCHAR	
http_client_pool_capacity	HTTP client pool capacity derived from CPU and async thread counts unless explicitly set	BIGINT	Derived automatically (max. 256)

Name	Description	Type	Default value
http_keep_alive	Keep alive connections. Setting this to false can help when running into connection failures	BOOLEAN	true
http_proxy_password	The password for the HTTP proxy.	VARCHAR	
http_proxy_username	The username for the HTTP proxy.	VARCHAR	
http_proxy	The HTTP proxy host. Defaults to the HTTP_PROXY environment variable.	VARCHAR	
http_retries	HTTP retries on I/O error	UBIGINT	3
http_retry_backoff	Backoff factor for exponentially increasing retry wait time	FLOAT	4.0
http_retry_wait_ms	Time between retries	UBIGINT	100
http_timeout	HTTP timeout read/write/connection/retry (in seconds)	UBIGINT	30
httpfs_client_implementation	Select which HTTP client implementation is used	VARCHAR	default
httpfs_connection_caching	Enable connection caching for HTTP requests	BOOLEAN	true
httpfs_enable_credential_refresh	Enable credential refresh for HTTPFS S3 secrets	BOOLEAN	true
ieee_floating_point_ops	Use IEEE 754 behavior for supported floating point operations, returning NAN/INF instead of errors/NULL.	BOOLEAN	true
ignore_unknown_crs	Ignore unknown Coordinate Reference Systems (CRS) when creating geometry types or importing geospatial data.	BOOLEAN	false
immediate_transaction_mode	Whether transactions should be started lazily when needed, or immediately when BEGIN TRANSACTION is called.	BOOLEAN	false
index_scan_max_count	The maximum index scan count sets a threshold for index scans. If fewer than MAX(index_scan_max_count, index_scan_percentage * total_row_count) rows match, we perform an index scan instead of a table scan.	UBIGINT	2048
index_scan_percentage	The index scan percentage sets a threshold for index scans. If fewer than MAX(index_scan_max_count, index_scan_percentage * total_row_count) rows match, we perform an index scan instead of a table scan.	DOUBLE	0.001
initial_column_segment_size	The initial memory (in bytes) reserved for the first transient column segment. Must be a power of two. Internally, we subtract the block header size (typically 8 bytes) for segments with or exceeding 1024 bytes. Subsequent segments double in size until reaching the block size.	UBIGINT	2048
integer_division	Whether or not the / operator defaults to integer division, or to floating point division.	BOOLEAN	false
json_geometry_format	How GEOMETRY values are written to JSON: 'wkt' for Well-Known Text, or 'geojson' for GeoJSON geometry objects. COPY ... TO ... (FORMAT GEOJSON) always writes GeoJSON regardless of this setting.	VARCHAR	wkt
late_materialization_max_rows	The maximum amount of rows in the LIMIT/SAMPLE operator for which we trigger late materialization.	UBIGINT	50

Name	Description	Type	Default value
legacy_disable_null_type	When enabled, prevent the NULL type from leaving the binder. This was the default behavior prior to DuckDB v2.0.	BOOLEAN	false
lock_configuration	Whether configurations can be altered.	BOOLEAN	false
log_query_path	Specifies the path to which queries should be logged. The default is NULL, meaning queries are not logged.	VARCHAR	
logging_level	Determines which 'severity' of log messages will be logged.	VARCHAR	INFO (WARNING in the CLI)
logging_mode	Determines which types of log messages are logged.	VARCHAR	LEVEL_ONLY
logging_storage	Set the logging storage, options are 'memory', 'stdout', 'file', ''.	VARCHAR	memory (shell_log_storage in the CLI)
max_execution_time	The maximum execution time per query in milliseconds (0 = no limit).	BIGINT	0
max_expression_depth	The maximum expression depth limit in the parser. WARNING: increasing this setting and using very deep expressions might lead to stack overflow errors.	UBIGINT	1000
max_memory, memory_limit	The maximum memory of the system (e.g., 1GB).	VARCHAR	80% of RAM
max_temp_directory_size	The maximum amount of data stored inside the 'temp_directory' (when set) (e.g., 1GB).	VARCHAR	90% of available disk space
max_vacuum_tasks	The maximum number of vacuum tasks to schedule during a checkpoint.	UBIGINT	100
merge_http_secret_into_s3_request	Merges HTTP secret parameters into S3 requests	BOOLEAN	true
merge_join_threshold	The maximum number of rows on either table to choose a merge join.	UBIGINT	1000
nested_loop_join_threshold	The maximum number of rows on either table to choose a nested loop join.	UBIGINT	5
order_by_non_integer_literal	Allow ordering by non-integer literals. Ordering by such literals has no effect. Enabling this relaxes the check to a silent no-op instead of a binder error.	BOOLEAN	false
ordered_aggregate_threshold	The number of rows to accumulate before sorting.	UBIGINT	262144
parquet_metadata_cache	Cache Parquet metadata - useful when reading the same files multiple times	BOOLEAN	false
parquet_prefetch_column_gap	Byte gap under which Parquet prefetch I/O ranges are coalesced (NULL lets the cost model adapt it)	UBIGINT	NULL
partitioned_write_flush_threshold	The number of rows threshold after which a thread state is flushed when writing using PARTITION_BY.	UBIGINT	524288
partitioned_write_max_open_files	The maximum amount of files the system can keep open before flushing to disk when writing using PARTITION_BY.	UBIGINT	100
password	The password to use. Kept for legacy compatibility.	VARCHAR	
perfect_ht_threshold	Threshold in bytes for when to use a perfect hash table.	UBIGINT	12

Name	Description	Type	Default value
pin_threads	Whether to pin threads to cores. Linux only, and the default is AUTO. AUTO means the setting is on when there are more than 64 cores.	VARCHAR	auto
pivot_filter_threshold	The threshold determining when a dedicated pivot operator switches from using filtered aggregates to LIST.	UBIGINT	20
pivot_limit	The maximum number of pivot columns in a pivot statement.	UBIGINT	100000
prefer_range_joins	Force the use of range joins with mixed predicates.	BOOLEAN	false
prefetch_all_parquet_files	(deprecated) Parquet files are now always prefetched, this setting has no effect	BOOLEAN	false
preserve_identifier_case	How to fold non-quoted identifiers. 'preserve_case' keeps the case as written, 'lowercase' lowercases them, 'uppercase' uppercases them.	VARCHAR	preserve_case
preserve_insertion_order	Whether or not to preserve insertion order. If set to false, then the system is allowed to re-order any results that do not contain ORDER BY clauses.	BOOLEAN	true
read_ahead_depth	The number of scan jobs prefetched ahead of decoding. -1 = automatic, with the backlog being bound by a memory budget, 0 = disabled.	BIGINT	-1
redact_http_logs	Redact HTTP header values in logs unless their names are explicitly allowed.	BOOLEAN	true
regex_match_operator_semantics	Configures whether regex match operators use partial or full string matching.	VARCHAR	partial
s3_allow_recursive_globbing	Whether globs on S3-like storage are optimized with recursive strategy (alternative is listing)	BOOLEAN	true
s3_enable_deprecated_url_parameters	Enable deprecated S3 URL parameters. NULL disables them by default; false permanently disables them for this database instance.	BOOLEAN	NULL
s3_list_concurrency	Maximum number of concurrent LIST requests issued while expanding an S3 glob. The window is also bounded by async_threads + 1 and shrinks automatically when the server throttles.	UBIGINT	256
s3_uploader_max_filesize	Maximum size of an S3 upload	VARCHAR	80GB
s3_uploader_max_parts_per_file	Maximum number of parts in an S3 multipart upload	UBIGINT	10000
s3_url_scheme_aliases	Additional URL schemes routed to the S3-compatible filesystem (e.g., ['oss', 'cos']). Can only be set globally; secrets for aliased schemes need an explicit SCOPE.	VARCHAR[]	[]
s3_version_id_pinning	Pin S3 reads to a specific object version for consistency	BOOLEAN	false
scalar_subquery_error_on_multiple_rows	Deprecated setting. Throw an error when a scalar subquery returns more than one row. This will become the default behavior. When disabled, an arbitrary row is returned instead.	BOOLEAN	true

Name	Description	Type	Default value
scheduler_process_partial	Partially process tasks before rescheduling. Enabling this allows for more scheduler fairness between separate queries.	BOOLEAN	false
secret_directory	Set the directory in which persistent secrets are stored.	VARCHAR	~/.duckdb/stored_secrets
show_behavior	Configures how SHOW resolves a bare identifier. 'auto' describes a table if one exists, else it describes a setting (deprecated). 'table' always describes a table, and 'setting' always describes a setting.	VARCHAR	AUTO
standard_vector_size	The compiled-in STANDARD_VECTOR_SIZE (read-only).	UBIGINT	2048
storage_block_prefetch	Configures the scenario in which to use storage block prefetching.	VARCHAR	REMOTE_ONLY
storage_compatibility_version	Sets the DuckDB version for which data is serialized to the DuckDB file during checkpointing. Can differ from the version of the DuckDB instance.	VARCHAR	latest
table_function_identifier_conversion	Configures the use of deprecated implicit conversion of unbound identifiers to strings in table function arguments.	VARCHAR	DEFAULT
temp_directory	Set the directory to which temporary files are written.	VARCHAR	<database_name>.tmp or .tmp (in in-memory mode)
temp_file_encryption	Encrypt all temporary files, if the database is encrypted.	BOOLEAN	false
threads, worker_threads	The number of total background threads used by the system.	BIGINT	# CPU cores
user, username	The username to use. Kept for legacy compatibility.	VARCHAR	
validate_external_file_cache	The cache validation mode. VALIDATE_ALL is the default, it validates all cache entries., VALIDATE_REMOTE only validates remote cache entries. NO_VALIDATION disables cache validation.	VARCHAR	VALIDATE_ALL
variant_minimum_shredding_size	Minimum size of a row group to have VARIANT shredding enabled. Set to -1 to disable shredding entirely. Defaults to 1/4th of the row group size.	BIGINT	30000
wal_autocheckpoint_entries	Trigger automatic checkpoint when WAL entry count reaches or exceeds N (0 = disabled).	UBIGINT	0
warnings_as_errors	Escalate all warnings to errors.	BOOLEAN	false
write_buffer_row_group_count	The amount of row groups to buffer in bulk ingestion prior to flushing them together. Reducing this setting can reduce memory consumption.	UBIGINT	5
write_buffer_row_group_memory_limit	The maximum data to buffer in row groups (in bytes) to buffer prior to flushing them together. When either this limit is reached, or write_buffer_row_group_count is reached, we flush the data to disk. Defaults to 20% of the memory limit divided by the thread count.	VARCHAR	413.9 MiB
zstd_min_string_length	The (average) minimum length at which to enable ZSTD compression, defaults to 4096.	UBIGINT	4096

Local Configuration Options

Name	Description	Type	Default value
`active_grammar_extensions`	The grammar extensions used by the parser.	`VARCHAR[]`	`[shell]`
`current_dialect`	The SQL dialect used by the parser.	`VARCHAR`	`NULL`
`enable_profiling`	Enables profiling and sets the output format, which can be `JSON`, `QUERY_TREE`, or `QUERY_TREE_OPTIMIZER`.	`VARCHAR`	`NULL`
`enable_progress_bar_print`	Controls the printing of the progress bar and requires 'enable_progress_bar' to be true.	`BOOLEAN`	`true`
`enable_progress_bar`	Enables the progress bar, which prints progress to the terminal during long queries.	`BOOLEAN`	`false` (`true` in the CLI)
`max_streaming_buffer_size`, `streaming_buffer_size`	The maximum number of bytes a streaming query result buffers (e.g., 1GB). Queued output stays under this cap, and each engine thread may hold, beyond the queue, the units it is still building or handing over from its current chunk.	`VARCHAR`	`10.0 MiB`
`operator_memory_limit`	The maximum memory available for query intermediates (sorts, hash tables) per connection (e.g., 256MB).	`VARCHAR`	`NULL`
`profile_output`, `profiling_output`	The file to which profiler output is written. When empty, the output is printed to the terminal.	`VARCHAR`	
`profiling_coverage`	The profiling coverage, either SELECT (only SELECT queries) or `ALL` (all queries are profiled).	`VARCHAR`	`SELECT`
`profiling_renderer_settings`	A map of settings passed to the renderer of the profiler output, e.g., {'max_extra_lines': 100}. Settings not recognized by the active renderer are ignored.	`MAP(VARCHAR, VARCHAR)`	`{}`
`progress_bar_time`	Configures how long (in milliseconds) a query needs to take before we start printing a progress bar.	`BIGINT`	`2000`
`schema`	Sets the default search schema. Equivalent to setting search_path to a single value.	`VARCHAR`	`main`
`search_path`	Sets the default catalog search path as a comma-separated list of values.	`VARCHAR`	
`tracked_metrics`	A list of metric glob patterns to enable for collection, e.g., ['query.*', *'optimizer.*'*].	`VARCHAR[]`	`[*]`

Deprecated Configuration Options

The following configuration options are deprecated and will be removed in a future release.

Name	Description	Type	Default value	Scope
`enable_curl_server_cert_verification`	Enable server side certificate verification for CURL backend.	`BOOLEAN`	`TRUE`	`GLOBAL`

Name	Description	Type	Default value	Scope
enable_object_cache	Deprecated legacy setting, does nothing.	BOOLEAN	false	GLOBAL
extension_directory	Deprecated setting. Use extension_directories instead	VARCHAR		GLOBAL
gcs_user_project	GCS billing project for Requester Pays	VARCHAR	NULL	GLOBAL
lambda_syntax	Deprecated setting. Configures the use of the deprecated single arrow operator (->) for lambda functions.	VARCHAR	DEFAULT	GLOBAL
legacy_metrics_format	Deprecated setting. When enabled, profiling output uses the legacy flat format instead of the current grouped format.	BOOLEAN	false	GLOBAL
old_implicit_casting	Deprecated setting. Used for compatibility for the pre-0.10 implicit VARCHAR casting behavior.	BOOLEAN	false	GLOBAL
produce_arrow_string_view	Deprecated setting. Controlled via enable_arrow_string_view.	BOOLEAN	false	GLOBAL
profiling_mode	Deprecated setting. Use enable_profiling and profiling_coverage instead.	VARCHAR	NULL	LOCAL
s3_access_key_id	S3 Access Key ID	VARCHAR	NULL	GLOBAL
s3_endpoint	S3 Endpoint	VARCHAR	NULL	GLOBAL
s3_kms_key_id	S3 KMS Key ID	VARCHAR	NULL	GLOBAL
s3_region	S3 Region	VARCHAR	NULL	GLOBAL
s3_requester_pays	S3 use requester pays mode	BOOLEAN	false	GLOBAL
s3_secret_access_key	S3 Access Key	VARCHAR	NULL	GLOBAL
s3_session_token	S3 Session Token	VARCHAR	NULL	GLOBAL
s3_url_compatibility_mode	Disable Globs and Query Parameters on S3 URLs	BOOLEAN	false	GLOBAL

Name	Description	Type	Default value	Scope
`s3_url_style`	S3 URL style	VARCHAR	`vhost`	GLOBAL
`s3_use_ssl`	S3 use SSL	BOOLEAN	`true`	GLOBAL
`vacuum_rebuild_indexes`	Deprecated compatibility setting: maximum table row count for rebuilding bound ART indexes when checkpoint vacuum cannot remap their row IDs. 0 disables the rebuild fallback but does not disable index remapping on v2.0 storage. The 'vacuum_rebuild_indexes' ATTACH option overrides this default per database.	UBIGINT	0	GLOBAL

Pragmas

The PRAGMA statement is a SQL extension adopted by DuckDB from SQLite. PRAGMA statements can be issued in a similar manner to regular SQL statements. PRAGMA commands may alter the internal state of the database engine, and can influence the subsequent execution or behavior of the engine.

Configure DuckDB settings using the SET statement. You can retrieve a setting's value using `SELECT current_setting('option_name')`.

> **Deprecated.** Assigning configuration settings with `PRAGMA name = value` is deprecated and emits a warning. Use `SET name = value{:.language-sql .highlight}` instead. The warning becomes an error when `warnings_as_errors` is enabled. Function-style PRAGMA calls (for example, `PRAGMA table_info('table_name')`) and the SQLite-compatible `PRAGMA table_info = 'table_name'` form are unaffected.

For DuckDB's built in configuration options, see the Configuration Reference. DuckDB extensions may register additional configuration options. These are documented in the respective extensions' documentation pages.

This page contains the supported PRAGMA settings.

Metadata

Schema Information List all databases:

```
PRAGMA database_list;
```

List all tables:

```
PRAGMA show_tables;
```

List all tables, with extra information, similarly to DESCRIBE:

```
PRAGMA show_tables_expanded;
```

To list all functions:

```
PRAGMA functions;
```

For queries targeting non-existing schemas, DuckDB generates "did you mean..." style error messages. When there are thousands of attached databases, these errors can take a long time to generate. To limit the number of schemas DuckDB looks through, use the `catalog_error_max_schemas` option:

```
SET catalog_error_max_schemas = 10;
```

Table Information Get info for a specific table:

```
PRAGMA table_info('table_name');
CALL pragma_table_info('table_name');
```

`table_info` returns information about the columns of the table with name `table_name`. The exact format of the table returned is given below:

```
cid INTEGER,        -- cid of the column
name VARCHAR,       -- name of the column
type VARCHAR,       -- type of the column
notnull BOOLEAN,    -- if the column is marked as NOT NULL
dflt_value VARCHAR, -- default value of the column, or NULL if not specified
pk BOOLEAN          -- part of the primary key or not
```

Database Size Get the file and memory size of each database:

```
PRAGMA database_size;
CALL pragma_database_size();
```

`database_size` returns information about the file and memory size of each database. The column types of the returned results are given below:

```
database_name VARCHAR, -- database name
database_size VARCHAR, -- total block count times the block size
block_size BIGINT,     -- database block size
total_blocks BIGINT,   -- total blocks in the database
used_blocks BIGINT,    -- used blocks in the database
free_blocks BIGINT,    -- free blocks in the database
wal_size VARCHAR,      -- write ahead log size
memory_usage VARCHAR,  -- memory used by the database buffer manager
memory_limit VARCHAR   -- maximum memory allowed for the database
```

Storage Information To get storage information:

```
PRAGMA storage_info('table_name');
CALL pragma_storage_info('table_name');
```

This call returns the following information for the given table:

Name	Type	Description
`row_group_id`	`BIGINT`	
`column_name`	`VARCHAR`	
`column_id`	`BIGINT`	
`column_path`	`VARCHAR`	
`segment_id`	`BIGINT`	
`segment_type`	`VARCHAR`	
`start`	`BIGINT`	The start row id of this chunk
`count`	`BIGINT`	The amount of entries in this storage chunk
`compression`	`VARCHAR`	Compression type used for this column – see the "Lightweight Compression in DuckDB" blog post
`stats`	`VARIANT`	
`has_updates`	`BOOLEAN`	
`persistent`	`BOOLEAN`	`false` if temporary table
`block_id`	`BIGINT`	Empty unless persistent
`block_offset`	`BIGINT`	Empty unless persistent
`additional_block_ids`	`BIGINT[]`	Empty unless persistent

> New As of DuckDB 2.0, when `pragma_storage_info` is called as a table function, it accepts an `include_segment_info` boolean parameter. When set to `true`, the result contains an additional `segment_info` VARCHAR column with per-segment detail for some compression methods, such as the `DICT_FSST` mode applied (for example, `DICTIONARY`):

```
SELECT compression, segment_info
FROM pragma_storage_info('table_name', include_segment_info := true);
```

See Storage for more information.

Show Databases The following statement is equivalent to the SHOW DATABASES statement:

```
PRAGMA show_databases;
```

Resource Management

Memory Limit Set the memory limit for the buffer manager:

```
SET memory_limit = '1GB';
```

The memory limit can also be specified as a percentage of the total system memory:

```
SET memory_limit = '60%';
```

> New As of DuckDB 2.0, `max_memory` (and its alias `memory_limit`) accepts a percentage of the total system memory. The percentage must be between 0 and 100 and is computed relative to the total system memory. Values that cannot be parsed as a valid percentage raise an error.

> **Warning.** The specified memory limit is only applied to the buffer manager. For most queries, the buffer manager handles the majority of the data processed. However, certain in-memory data structures such as vectors and query results are allocated outside of the buffer manager. Additionally, aggregate functions with complex state (e.g., `list`, `mode`, `quantile`, `string_agg`, and `approx` functions) use memory outside of the buffer manager. Therefore, the actual memory consumption can be higher than the specified memory limit.

Threads Set the amount of threads for parallel query execution:

```
SET threads = 4;
```

Maximum Execution Time

> New As of DuckDB 2.0, the `max_execution_time` setting bounds the wall-clock time of a query in milliseconds. When the limit is exceeded, the query is aborted with the error `Query exceeded maximum execution time`. The default is `0`, meaning no limit.

```
SET max_execution_time = 5000;
```

This is a per-connection setting with a global fallback: a local value overrides a value set with `SET GLOBAL`, and `RESET max_execution_time` restores the global value.

Collations

List all available collations:

```
PRAGMA collations;
```

Set the default collation to one of the available ones:

```
SET default_collation = 'nocase';
```

Default Ordering for NULLs

Set the default ordering for NULLs to be either `NULLS_FIRST`, `NULLS_LAST`, `NULLS_FIRST_ON_ASC_LAST_ON_DESC` or `NULLS_LAST_ON_ASC_FIRST_ON_DESC`:

```
SET default_null_order = 'NULLS_FIRST';
SET default_null_order = 'NULLS_LAST_ON_ASC_FIRST_ON_DESC';
```

Set the default result set ordering direction to `ASCENDING` or `DESCENDING`:

```
SET default_order = 'ASCENDING';
SET default_order = 'DESCENDING';
```

Ordering by Non-Integer Literals

By default, ordering by non-integer literals is not allowed:

```
SELECT 42 ORDER BY 'hello world';
```

```
-- Binder Error: ORDER BY non-integer literal has no effect.
```

To allow this behavior, use the `order_by_non_integer_literal` option:

```
SET order_by_non_integer_literal = true;
```

Implicit Casting to VARCHAR

Prior to version 0.10.0, DuckDB would automatically allow any type to be implicitly cast to VARCHAR during function binding. As a result it was possible to e.g., compute the substring of an integer without using an explicit cast. For version v0.10.0 and later an explicit cast is needed instead. To revert to the old behavior that performs implicit casting, set the `old_implicit_casting` variable to `true`. Note that this setting is deprecated and will be removed in a future release:

```
SET old_implicit_casting = true;
```

Python: Scan All Dataframes

Prior to version 1.1.0, DuckDB's replacement scan mechanism in Python scanned the global Python namespace. To revert to this old behavior, use the following setting:

```
SET python_scan_all_frames = true;
```

Information on DuckDB

Version Show DuckDB version:

```
PRAGMA version;
CALL pragma_version();
```

Platform `platform` returns an identifier for the platform the current DuckDB executable has been compiled for, e.g., `osx_arm64`. The format of this identifier matches the platform name as described in the extension loading explainer:

```
PRAGMA platform;
CALL pragma_platform();
```

User Agent The following statement returns the user agent information, e.g., `duckdb/v0.10.0(osx_arm64)`:

```
PRAGMA user_agent;
```

Metadata Information The following statement returns information on the metadata store (`block_id`, `total_blocks`, `free_blocks`, and `free_list`):

```
PRAGMA metadata_info;
```

Extension Versions The following statement returns the name, version, and install mode of each loaded extension (`extension_name`, `extension_version`, `install_mode`, and `installed_from`):

```
PRAGMA extension_versions;
```

Progress Bar

Show progress bar when running queries:

```
PRAGMA enable_progress_bar;
```

Or:

```
PRAGMA enable_print_progress_bar;
```

Don't show a progress bar for running queries:

```
PRAGMA disable_progress_bar;
```

Or:

```
PRAGMA disable_print_progress_bar;
```

EXPLAIN Output

The output of `EXPLAIN` can be configured to show only the physical plan.

The default configuration of `EXPLAIN`:

```
SET explain_output = 'physical_only';
```

To only show the optimized query plan:

```
SET explain_output = 'optimized_only';
```

To show all query plans:

```
SET explain_output = 'all';
```

Profiling

Enable Profiling

The following query enables profiling with the default format, `query_tree`. Independent of the format, `enable_profiling` is **mandatory** to enable profiling.

```
PRAGMA enable_profiling;
PRAGMA enable_profile;
```

Profiling Coverage

By default, the profiling coverage is set to `SELECT`. `SELECT` runs the profiler for each operator in the physical plan of a `SELECT` statement.

```
SET profiling_coverage = 'SELECT';
```

By default, the profiler **does not** emit profiling information for other statement types (`INSERT INTO`, `ATTACH`, etc.). To run the profiler for all statement types, change this setting to `ALL`.

```
SET profiling_coverage = 'ALL';
```

Profiling Format

The format of `enable_profiling` can be specified as `query_tree`, `json`, `query_tree_optimizer`, or `no_output`. Each format prints its output to the configured output, except `no_output`.

The default format is `query_tree`. It prints the physical query plan and the metrics of each operator in the tree.

```
SET enable_profiling = 'query_tree';
```

Alternatively, `json` returns the physical query plan as JSON:

```
SET enable_profiling = 'json';
```

> **Tip.** To visualize query plans, consider using the DuckDB execution plan visualizer developed by the Database Systems Research Group at the University of Tübingen.

To return the physical query plan, including optimizer and planner metrics:

```
SET enable_profiling = 'query_tree_optimizer';
```

Database drivers and other applications can also access profiling information through API calls, in which case users can disable any other output. Even though the parameter reads `no_output`, it is essential to note that this **only** affects printing to the configurable output. When accessing profiling information through API calls, it is still crucial to enable profiling:

```
SET enable_profiling = 'no_output';
```

Profiling Output

By default, DuckDB prints profiling information to the standard output. However, if you prefer to write the profiling information to a file, you can use the `profiling_output` setting to specify a filepath.

> **Warning.** The file contents will be overwritten for every newly issued query. Hence, the file will only contain the profiling information of the last run query:

```
SET enable_profiling = 'json';
SET profiling_output = '/path/to/file.json';
SET profile_output = '/path/to/file.json';
```

The file extension must match the profiling output format set in `enable_profiling` (e.g., `.json` for `json`), or be `.txt`.

Profiling Mode

> **Deprecated.** The `profiling_mode` setting is deprecated in DuckDB 2.0 and its value has no effect: detailed profiling information is always collected. To enable profiling, use `PRAGMA enable_profiling`. To select which metrics are collected, use the `tracked_metrics` setting.

Custom Metrics

> New The `tracked_metrics` setting was introduced in DuckDB 2.0, replacing the `custom_profiling_settings` PRAGMA used in earlier versions.

By default, profiling collects all metrics. To collect only a subset, use the `tracked_metrics` setting, which accepts a list of metric glob patterns. A metric is collected if its name matches any of the patterns. The default value, `['*']`, matches all metrics.

> **Note.** This only affects the metrics when `enable_profiling` is set to `json` or `no_output`. The `query_tree` and `query_tree_optimizer` always use a default set of metrics.

Metric names are grouped by stage and written in `group.metric` form, for example `query.cpu_time` or `operator.timing`. The following example collects only the query-level metrics and every operator metric:

```
SET tracked_metrics = ['query.*', 'operator.*'];
```

To list all available metric names, their types, and units, query the `duckdb_available_metrics()` table function:

```
SELECT metric_name, metric_type, unit
FROM duckdb_available_metrics();
```

The profiling documentation contains an overview of the available metrics.

Disable Profiling

To disable profiling:

```
PRAGMA disable_profiling;
PRAGMA disable_profile;
```

Query Optimization

Optimizer To disable the query optimizer:

```
PRAGMA disable_optimizer;
```

To enable the query optimizer:

```
PRAGMA enable_optimizer;
```

Selectively Disabling Optimizers The `disabled_optimizers` option allows selectively disabling optimization steps. For example, to disable `filter_pushdown` and `statistics_propagation`, run:

```
SET disabled_optimizers = 'filter_pushdown,statistics_propagation';
```

The available optimizations can be queried using the `duckdb_optimizers()` table function.

To re-enable the optimizers, run:

```
SET disabled_optimizers = '';
```

> **Warning.** The `disabled_optimizers` option should only be used for debugging performance issues and should be avoided in production.

Logging

Set a path for query logging:

```
SET log_query_path = '/tmp/duckdb_log/';
```

Disable query logging:

```
SET log_query_path = '';
```

Full-Text Search Indexes

The `create_fts_index` and `drop_fts_index` options are only available when the `fts` extension is loaded. Their usage is documented on the Full-Text Search extension page.

Object Cache

> **Deprecated.** The `enable_object_cache` and `disable_object_cache` pragmas are deprecated in DuckDB 2.0 and no longer have any effect.

Enable caching of objects for e.g., Parquet metadata:

```
PRAGMA enable_object_cache;
```

Disable caching of objects:

```
PRAGMA disable_object_cache;
```

Checkpointing

Compression During checkpointing, the existing column data + any new changes get compressed. There exist a couple of settings to influence which compression functions are considered.

Force Compression Prefer using this compression method over any other method if possible:

```
SET force_compression = 'bitpacking';
```

Disabled Compression Methods Avoid using any of the listed compression methods from the comma separated list:

```
SET disabled_compression_methods = 'fsst,rle';
```

Force Checkpoint When `CHECKPOINT` is called when no changes are made, force a checkpoint regardless:

```
PRAGMA force_checkpoint;
```

Checkpoint on Shutdown Run a `CHECKPOINT` on successful shutdown and delete the WAL, to leave only a single database file behind:

```
PRAGMA enable_checkpoint_on_shutdown;
```

Don't run a `CHECKPOINT` on shutdown:

```
PRAGMA disable_checkpoint_on_shutdown;
```

Checkpoint on Detach

> New As of DuckDB 2.0, the `checkpoint_on_detach` setting controls whether a `CHECKPOINT` runs when a database is detached:

```
SET checkpoint_on_detach = 'ENABLED';
```

The accepted values are `ENABLED` (request a checkpoint on detach), `DISABLED` (never checkpoint on detach), and `DEFAULT` (defer to the global `checkpoint_on_shutdown` behavior). A requested checkpoint still does not occur while another connection references the database. This is a per-session setting.

Temp Directory for Spilling Data to Disk

By default, DuckDB uses a temporary directory named `<database_file_name>.tmp{:.language-sql .highlight}` to spill to disk, located in the same directory as the database file. To change this, use:

```
SET temp_directory = '/path/to/temp_dir.tmp/';
```

Returning Errors as JSON

The `errors_as_json` option can be set to obtain error information in raw JSON format. For certain errors, extra information or decomposed information is provided for easier machine processing. For example:

```
SET errors_as_json = true;
```

Then, running a query that results in an error produces a JSON output:

```
SELECT * FROM nonexistent_tbl;
```

```
{
   "exception_type":"Catalog",
   "exception_message":"Table with name nonexistent_tbl does not exist!\nDid you mean \"temp.information_
   schema.tables\"?",
   "name":"nonexistent_tbl",
   "candidates":"temp.information_schema.tables",
   "position":"14",
   "type":"Table",
   "error_subtype":"MISSING_ENTRY"
}
```

IEEE Floating-Point Operation Semantics

DuckDB follows IEEE floating-point operation semantics. If you would like to turn this off, run:

```
SET ieee_floating_point_ops = false;
```

In this case, floating point division by zero (e.g., `1.0 / 0.0`, `0.0 / 0.0` and `-1.0 / 0.0`) throws an error. To return `NULL` instead, also set `error_on_division_by_zero = false` (see below).

> New The semantics of integer division by zero changed in DuckDB 2.0.

Integer division by zero (`1 // 0`, `1 % 0`, `INTERVAL '1' DAY / 0`) errors by default. To return `NULL`, similarly to DuckDB 1.x, use:

```
SET error_on_division_by_zero = false;
```

> **Deprecated.** The `error_on_division_by_zero` setting is deprecated and will be removed in a future release. Setting it prints a deprecation warning in the CLI. To return `NULL` for a specific expression, wrap it in `TRY` instead, e.g., `TRY(1 // 0)`.

Query Verification (for Development)

The following PRAGMAs are mostly used for development and internal testing.

> **Deprecated.** The `enable_verification` and `disable_verification` pragmas are deprecated in DuckDB 2.0 and no longer have any effect.

Enable query verification:

```
PRAGMA enable_verification;
```

Disable query verification:

```
PRAGMA disable_verification;
```

Enable force parallel query processing:

```
PRAGMA verify_parallelism;
```

Disable force parallel query processing:

```
PRAGMA disable_verify_parallelism;
```

Block Sizes

When persisting a database to disk, DuckDB writes to a dedicated file containing a list of blocks holding the data. In the case of a file that only holds very little data, e.g., a small table, the default block size of 256 kB might not be ideal. Therefore, DuckDB's storage format supports different block sizes.

There are a few constraints on possible block size values.

- Must be a power of two.
- Must be greater or equal to 16384 (16 kB).
- Must be lesser or equal to 262144 (256 kB).

You can set the default block size for all new DuckDB files created by an instance like so:

```
SET default_block_size = '16384';
```

It is also possible to set the block size on a per-file basis, see ATTACH for details.

Secrets Manager

The **Secrets manager** provides a unified user interface for secrets across all backends that use them. Secrets can be scoped, so different storage prefixes can have different secrets, allowing for example to join data across organizations in a single query. Secrets can also be persisted, so that they do not need to be specified every time DuckDB is launched.

> **Note.** Persistent secrets are stored in unencrypted binary format in the `~/.duckdb/stored_secrets` directory. They are protected by file system permissions (mode `600`), similar to how SSH private keys are stored: only the user running the DuckDB process can read and write them.

Types of Secrets

Secrets are typed, their type identifies which service they are for. Most secrets are not included in DuckDB by default; instead, they are registered by extensions. Currently, the following secret types are available:

Secret type	Service / protocol	Extension
`azure`	Azure Blob Storage	`azure`
`ducklake`	DuckLake	`ducklake`
`gcs`	Google Cloud Storage	`httpfs`
`http`	HTTP and HTTPS	`httpfs`
`huggingface`	Hugging Face	`httpfs`
`iceberg`	Iceberg REST Catalog	`httpfs`, `iceberg`
`mysql`	MySQL	`mysql`
`postgres`	PostgreSQL	`postgres`
`quack`	Quack	`quack`
`r2`	Cloudflare R2	`httpfs`
`s3`	AWS S3	`httpfs`
`aws`	AWS	`aws`

For each type, there are one or more "secret providers" that specify how the secret is created. Secrets can also have an optional scope, which is a file path prefix that the secret applies to. When fetching a secret for a path, the secret scopes are compared to the path, returning the matching secret for the path. In the case of multiple matching secrets, the longest prefix is chosen.

Creating a Secret

Secrets can be created using the `CREATE SECRET` SQL statement. Secrets can be **temporary** or **persistent**. Temporary secrets are used by default – and are stored in-memory for the life span of the DuckDB instance similar to how settings worked previously. Persistent secrets are stored in **unencrypted binary format** in the `~/.duckdb/stored_secrets` directory. On startup of DuckDB, persistent secrets are read from this directory and automatically loaded.

Secret Providers

To create a secret, a **Secret Provider** needs to be used. A Secret Provider is a mechanism through which a secret is generated. To illustrate this, for the S3, GCS, R2, and AZURE secret types, DuckDB currently supports two providers: `CONFIG` and `credential_chain`. The `CONFIG` provider requires the user to pass all configuration information into the `CREATE SECRET`, whereas the `credential_chain` provider will automatically try to fetch credentials. When no Secret Provider is specified, the `CONFIG` provider is used. For more details on how to create secrets using different providers check out the respective pages on httpfs and azure.

Temporary Secrets

To create a temporary unscoped secret to access S3, we can now use the following:

```
CREATE SECRET my_secret (
    TYPE s3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value',
    REGION 'my_region'
);
```

Note that we implicitly use the default `CONFIG` secret provider here.

Persistent Secrets

In order to persist secrets between DuckDB database instances, we can now use the `CREATE PERSISTENT SECRET` command, e.g.:

```
CREATE PERSISTENT SECRET my_persistent_secret (
    TYPE s3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value'
);
```

By default, this will write the secret (unencrypted) to the `~/.duckdb/stored_secrets` directory. To change the secrets directory, issue:

```
SET secret_directory = 'path/to/my_secrets_dir';
```

The `secret_directory` setting applies to the current DuckDB instance and is not persisted across DuckDB instances. If you use a custom secrets directory, set `secret_directory` again when starting a new DuckDB instance before creating or using persistent secrets from that directory.

Note that setting the value of the `home_directory` configuration option has no effect on the location of the secrets.

Connection-Scoped Secrets

> New As of DuckDB 2.0, a secret can be stored in connection-scoped storage so that it is only visible to queries running on the same connection. Use the `IN connection` clause:

```
CREATE SECRET my_secret IN connection (
    TYPE s3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value'
);
```

Connection-scoped secrets are always temporary. This lets different connections hold secrets of the same type and scope without clashing.

Deleting Secrets

Secrets can be deleted using the DROP SECRET statement, e.g.:

```
DROP PERSISTENT SECRET my_persistent_secret;
```

Creating Multiple Secrets for the Same Service Type

If two secrets exist for a service type, the scope can be used to decide which one should be used. For example:

```
CREATE SECRET secret1 (
    TYPE s3,
    KEY_ID 'my_secret_key1',
    SECRET 'my_secret_value1',
    SCOPE 's3://<my-bucket>'
);
```

```
CREATE SECRET secret2 (
    TYPE s3,
    KEY_ID 'my_secret_key2',
    SECRET 'my_secret_value2',
    SCOPE 's3://<my-other-bucket>'
);
```

Now, if the user queries something from `s3://<my-other-bucket>/something`, secret `secret2` will be chosen automatically for that request. To see which secret is being used, the `which_secret` scalar function can be used, which takes a path and a secret type as parameters:

```
FROM which_secret('s3://<my-other-bucket>/file.parquet', 's3');
```

Listing Secrets

Secrets can be listed using the built-in table-producing function, e.g., by using the `duckdb_secrets()` table function:

```
FROM duckdb_secrets();
```

Sensitive information will be redacted.

Extensions

Extensions

DuckDB has a flexible extension mechanism that allows for dynamically loading extensions. Extensions can enhance DuckDB's functionality by providing support for additional file formats, introducing new types, and domain-specific functionality.

> Extensions are loadable on all clients (e.g., Python and R). Extensions distributed via the Core and Community repositories are built and tested on macOS, Windows and Linux. All operating systems are supported for both the AMD64 and the ARM64 architectures, with the exception of macOS on Intel (`osx_amd64`), which is being phased out: in DuckDB 2.0, core extensions are still built for `osx_amd64` but community extensions are no longer built for it, and DuckDB 2.1 will drop support for the platform.

Listing Extensions

To get a list of extensions, use the `duckdb_extensions` function:

```
SELECT extension_name, installed, description
FROM duckdb_extensions();
```

extension_name	installed	description
autocomplete	true	Adds support for autocomplete in the shell
avro	false	Adds support for reading Avro files
aws	false	Provides features that depend on the AWS SDK
azure	false	Adds a filesystem abstraction for Azure blob storage to DuckDB
...	...	...

This list will show which extensions are available, which extensions are installed, at which version, where it is installed, and more. The list includes most, but not all, available core extensions. For the full list, we maintain a list of core extensions.

> **Tip.** We provide an endpoint that serves weekly extension download statistics as JSON files: core extensions and community extensions.

Built-In Extensions

DuckDB's binary distribution comes standard with a few built-in extensions. They are statically linked into the binary and can be used as is. For example, to use the built-in `json` extension to read a JSON file:

```
SELECT *
FROM 'test.json';
```

To make the DuckDB distribution lightweight, only a few essential extensions are built-in, varying slightly per distribution. To check which extensions are built into your DuckDB distribution, query the `install_mode` column of the `duckdb_extensions` function:

```
SELECT extension_name
FROM duckdb_extensions()
WHERE install_mode = 'STATICALLY_LINKED';
```

> Most of DuckDB's scalar and aggregate functions are provided by the `core_functions` extension, which is built in and loaded by default in the standard distributions (such as the CLI). Minimal embeddings that do not load any extensions by default may need to run `LOAD core_functions;` explicitly. Some extensions depend on it as well, for example DuckLake, which otherwise fails with an error such as `Binder Error: Failed to load DuckLake table data. Referenced table "system" not found!`.

Installing More Extensions

To make an extension that is not built-in available in DuckDB, two steps need to happen:

1. **Extension installation** is the process of downloading the extension binary and verifying its metadata. During installation, DuckDB stores the downloaded extension and some metadata in a local directory. From this directory DuckDB can then load the Extension whenever it needs to. This means that installation needs to happen only once.
2. **Extension loading** is the process of dynamically loading the binary into a DuckDB instance. DuckDB will search the local extension directory for the installed extension, then load it to make its features available. This means that every time DuckDB is restarted, all extensions that are used need to be (re)loaded

> Extension installation and loading are subject to a few limitations.

There are two main methods of making DuckDB perform the **installation** and **loading** steps for an installable extension: **explicitly** and through **autoloading**.

Explicit `INSTALL` and `LOAD`

In DuckDB extensions can also be explicitly installed and loaded. Both non-autoloadable and autoloadable extensions can be installed this way. To explicitly install and load an extension, DuckDB has the dedicated SQL statements `INSTALL` and `LOAD`, which can be combined into a single `INSTALL ... AND ... LOAD` statement. For example, to install and load the `spatial` extension, run:

```
INSTALL AND LOAD spatial;
```

With this statement, DuckDB will ensure the `spatial` extension is installed (skipping the installation if it is already installed), then proceed to load the spatial extension (again skipping this step if it is already loaded).

Extension Repository Optionally a repository can be provided where the extension should be installed from, by appending `FROM <repository>`{:.language-sql .highlight} to the `INSTALL` / `FORCE INSTALL` command. This repository can either be an alias, such as `community`, the name of a custom extension repository, or a direct URL, provided as a single-quoted string. Custom extension repositories, introduced in DuckDB 2.0, allow organizations to host extensions signed with their own keys, which DuckDB pins when the repository is created.

After installing/loading an extension, the `duckdb_extensions` function can be used to get more information.

Autoloading Extensions

For many of DuckDB's core extensions, explicitly loading and installing extensions is not necessary. DuckDB contains an autoloading mechanism which can install and load the core extensions as soon as they are used in a query. For example, when running:

```
SELECT *
FROM 'https://raw.githubusercontent.com/duckdb/duckdb-web/main/data/weather.csv';
```

DuckDB will automatically install and load the `httpfs` extension. No explicit `INSTALL` or `LOAD` statements are required.

Not all extensions can be autoloaded. This can have various reasons: some extensions make several changes to the running DuckDB instance, making autoloading technically not (yet) possible. For others, it is preferred to have users opt-in to the extension explicitly before use due to the way they modify behavior in DuckDB.

Autoloading only applies to core extensions. If a query requires an extension that cannot be autoloaded, DuckDB typically raises an error that names the extension, which you can then install and load explicitly.

Community Extensions

DuckDB supports installing third-party community extensions. For example, you can install the `tarfs` community extension via:

```
INSTALL tarfs FROM community;
```

Community extensions are contributed by community members but they are built, signed, and distributed in a centralized repository.

Updating Extensions

While built-in extensions are tied to a DuckDB release due to their nature of being built into the DuckDB binary, installable extensions can and do receive updates. To ensure all currently installed extensions are on the most recent version, call:

```
UPDATE EXTENSIONS;
```

For more details on extension versions, refer to the Extension Versioning page.

Developing Extensions

The same API that the core extensions use is available for developing extensions. This allows users to extend the functionality of DuckDB such that it suits their domain the best. A template for creating extensions is available in the `extension-template` repository. This template also holds some documentation on how to get started building your own extension.

Working with Extensions

See the installation instructions and the advanced installation methods page.

Installing Extensions

To install core DuckDB extensions, use the `INSTALL` command. For example:

```
INSTALL httpfs;
```

This installs the extension from the default repository (`core`).

Extension Repositories

By default, DuckDB extensions are installed from a single repository containing extensions built and signed by the core DuckDB team. This ensures the stability and security of the core set of extensions. These extensions live in the default `core` repository, which points to `http://extensions.duckdb.org`.

Besides the core repository, DuckDB also supports installing extensions from other repositories. For example, the `core_nightly` repository contains nightly builds for core extensions that are built for the latest stable release of DuckDB. This allows users to try out new features in extensions before they are officially published.

Installing Extensions from Different Repositories

To install extensions from the default repository (`core`), run:

```
INSTALL httpfs;
```

To explicitly install an extension from the core repository, run:

```
INSTALL httpfs FROM core;
-- or
INSTALL httpfs FROM 'http://extensions.duckdb.org';
```

To install an extension from the core nightly repository:

```
INSTALL spatial FROM core_nightly;
-- or
INSTALL spatial FROM 'http://nightly-extensions.duckdb.org';
```

To install an extension from a custom repository:

```
INSTALL <custom_extension> FROM 'https://my-custom-extension-repository';
```

Alternatively, the `custom_extension_repository` setting can be used to change the default repository used by DuckDB:

```
SET custom_extension_repository = 'http://nightly-extensions.duckdb.org';
```

DuckDB contains the following predefined repositories:

Alias	URL	Description
`core`	`http://extensions.duckdb.org`	DuckDB core extensions
`core_nightly`	`http://nightly-extensions.duckdb.org`	Nightly builds for `core`

Alias	URL	Description
`community`	`http://community-extensions.duckdb.org`	DuckDB community extensions
`local_build_debug`	`./build/debug/repository`	Repository created when building DuckDB from source in debug mode (for development)
`local_build_release`	`./build/release/repository`	Repository created when building DuckDB from source in release mode (for development)

Backup Repositories

The `core` and `community` repositories are mirrored at `http://extensions.duckdb-backup.org` and `http://community-extensions.duckdb-backup.org`. If the primary server is unavailable, DuckDB automatically falls back to the backup server. This does not apply to repositories set via `custom_extension_repository` or `INSTALL ... FROM '<url>'`.

To use the backup server manually, run:

```
SET custom_extension_repository = 'http://extensions.duckdb-backup.org';
```

For details, see Backup Extension Repositories.

Working with Multiple Repositories

When working with extensions from different repositories, especially mixing `core` and `core_nightly`, it is important to know the origins and version of the different extensions. For this reason, DuckDB keeps track of this in the extension installation metadata. For example:

```
INSTALL httpfs FROM core;
INSTALL aws FROM core_nightly;
SELECT extension_name, extension_version, installed_from, install_mode
FROM duckdb_extensions();
```

This outputs:

extensions_name	extensions_version	installed_from	install_mode
httpfs	62d61a417f	core	REPOSITORY
aws	42c78d3	core_nightly	REPOSITORY
...	...	...	...

Force Installing to Upgrade Extensions

When DuckDB installs an extension, it is copied to a local directory to be cached and avoid future network traffic. Any subsequent calls to `INSTALL <extension_name>{:.language-sql .highlight}` will use the local version instead of downloading the extension again. To force re-downloading the extension, run:

```
FORCE INSTALL extension_name;
```

Force installing can also be used to overwrite an extension with an extension of the same name from another repository,

For example, first, `spatial` is installed from the core repository:

```
INSTALL spatial;
```

Then, to overwrite this installation with the `spatial` extension from the `core_nightly` repository:

```
FORCE INSTALL spatial FROM core_nightly;
```

Switching between Repositories

To switch repositories for an extension, use the `FORCE INSTALL` command. For example, if you have installed `httpfs` from the `core_nightly` repository but would like to switch back to using `core`, run:

```
FORCE INSTALL httpfs FROM core;
```

Installing Extensions through Client APIs

For many clients, using SQL to load and install extensions is the preferred method. However, some clients have a dedicated API to install and load extensions. For example, the Python client, has dedicated `install_extension(extension: str)` and `load_extension(extension: str)` methods. For more details on a specific client API, refer to the Client API documentation

Installation Location

By default, extensions are installed under the user's home directory:

```
~/.duckdb/extensions/<duckdb_version>/<platform_name>/
```

For stable DuckDB releases, the <duckdb_version>{:.language-sql .highlight} will be equal to the version tag of that release. For nightly DuckDB builds, it will be equal to the first 10 characters of the build's commit hash. So for example, the extensions for DuckDB version v0.10.3 on macOS ARM64 (Apple Silicon) are installed to `~/.duckdb/extensions/v0.10.3/osx_arm64/`. An example installation path for a nightly DuckDB build could be `~/.duckdb/extensions/fc2e4b26a6/linux_amd64`.

To change the location where DuckDB stores and loads its extensions, use the `extension_directories` configuration option. It takes a list of directories: extensions are installed into the first directory and loaded from any of them, which is useful, e.g., for package managers or air-gapped environments:

```
SET extension_directories = ['/path/to/your/extension/directory'];
SET extension_directories = ['/usr/lib/duckdb/extensions', '/opt/duckdb/extensions'];
```

> **Deprecated.** The `extension_directory` setting (singular) is deprecated in DuckDB 2.0 and emits a warning when set. It will be removed in a future release. Use `extension_directories` instead.

Note that setting the value of the `home_directory` configuration option has no effect on the location of the extensions.

Uninstalling Extensions

Currently, DuckDB does not provide a command to uninstall extensions. To uninstall an extension, navigate to the extension's Installation Location and remove its `.duckdb_extension` binary file: For example:

```
rm ~/.duckdb/extensions/v1.2.1/osx_arm64/excel.duckdb_extension
```

Sharing Extensions between Clients

The shared installation location allows extensions to be shared between the client APIs *of the same DuckDB version*, as long as they share the same `platform` or ABI. For example, if an extension is installed with version 1.2.1 of the CLI client on macOS, it is available from the Python, R, etc. client libraries provided that they have access to the user's home directory and use DuckDB version 1.2.1.

Limitations

DuckDB's extension mechanism has the following limitations:

- Extensions cannot be unloaded.
- Extensions cannot be reloaded. If you update extensions, restart the DuckDB process to use newer extensions.

Advanced Installation Methods

Downloading Extensions Directly from S3

Downloading an extension directly can be helpful when building a Lambda service or container that uses DuckDB. DuckDB extensions are stored in public S3 buckets, but the directory structure of those buckets is not searchable. As a result, a direct URL to the file must be used. To download an extension file directly, use the following format:

```
http://extensions.duckdb.org/v<duckdb_version>/<platform_name>/<extension_name>.duckdb_extension.gz
```

For example:

```
http://extensions.duckdb.org/v2.0.0/windows_amd64/json.duckdb_extension.gz
```

Installing an Extension from an Explicit Path

The `INSTALL` command can be used with the path to a `.duckdb_extension` file:

```
INSTALL 'path/to/httpfs.duckdb_extension';
```

Note that compressed `.duckdb_extension.gz` files need to be decompressed beforehand. It is also possible to specify remote paths.

Loading an Extension from an Explicit Path

`LOAD` can be used with the path to a `.duckdb_extension`. For example, if the file was available at the (relative) path `path/to/httpfs.duckdb_extension`, you can load it as follows:

```
LOAD 'path/to/httpfs.duckdb_extension';
```

This will skip any currently installed extensions and load the specified extension directly.

Note that using remote paths for compressed files is currently not possible.

Building and Installing Extensions from Source

For building and installing extensions from source, see the Building DuckDB guide.

Statically Linking Extensions

To statically link extensions, follow the developer documentation's "Using extension config files" section.

Extension Distribution

Platforms

Extension binaries are distributed for several platforms (see below). For platforms where packages for certain extensions are not available, users can build them from source and install the resulting binaries manually.

Most official extensions are distributed for the following platforms. Some extensions are only available on a subset of these platforms; see the individual extension pages for details.

Platform name	Operating system	Architecture	CPU types
`linux_amd64`	Linux	x86_64 (AMD64)	
`linux_arm64`	Linux	AArch64 (ARM64)	AWS Graviton, Snapdragon, etc.
`osx_amd64`	macOS	x86_64 (AMD64)	Intel
`osx_arm64`	macOS	AArch64 (ARM64)	Apple Silicon M1, M2, etc.
`windows_amd64`	Windows	x86_64 (AMD64)	Intel, AMD, etc.
`windows_arm64`	Windows	AArch64 (ARM64)	Copilot+ PC with Qualcomm CPU

> **Deprecated.** The `osx_amd64` platform (macOS on Intel) is being phased out. In DuckDB 2.0, DuckDB and its core extensions are still built for `osx_amd64`, but community extensions are no longer built for this platform. DuckDB 2.1 will stop supporting `osx_amd64`.

Some extensions are distributed for the following platforms:

- `linux_amd64_musl` and `linux_arm64_musl`
- `windows_amd64_mingw`
- `wasm_eh` and `wasm_mvp` (see DuckDB-Wasm's extensions)

For platforms outside the ones listed above, we do not officially distribute extensions (e.g., `linux_arm64_android`, `openharmony_arm64`).

Extensions Signing

Signed Extensions

Extensions can be signed with a cryptographic key. By default, DuckDB uses its built-in public keys to verify the integrity of extensions before loading them. All core and community extensions are signed by the DuckDB team. In addition, custom extension repositories can serve extensions signed with their own keys: DuckDB pins the repository's public keys when the repository is created and verifies the extensions installed from it against these keys.

Signing the extension simplifies their distribution, this is why they can be distributed over HTTP without the need for HTTPS, which itself is supported through an extension (`httpfs`).

Unsigned Extensions

> **Warning.** Only load unsigned extensions from sources you trust. Avoid loading unsigned extensions over HTTP. Consult the Securing DuckDB page for guidelines on how to set up DuckDB in a secure manner.

If you wish to load your own extensions or extensions from third-parties you will need to enable the `allow_unsigned_extensions` flag. To load unsigned extensions using the CLI client, pass the `-unsigned` flag to it on startup:

```
duckdb -unsigned
```

Now any extension can be loaded, signed or not:

```
LOAD './some/local/ext.duckdb_extension';
```

For client APIs, the `allow_unsigned_extensions` database configuration options needs to be set, see the respective Client API docs. For example, for the Python client, see the Loading and Installing Extensions section in the Python API documentation.

Binary Compatibility

To avoid binary compatibility issues, the binary extensions distributed by DuckDB are tied both to a specific DuckDB version and a platform. This means that DuckDB can automatically detect binary compatibility between it and a loadable extension. When trying to load an extension that was compiled for a different version or platform, DuckDB will throw an error and refuse to load the extension.

DuckDB stores this compatibility information in a metadata footer appended to the end of the extension binary.

> **Warning.** Do not run `strip` on an extension binary. Stripping removes or corrupts the appended metadata footer, after which DuckDB reports the metadata as invalid and refuses to load the extension. On some platforms, `strip` fails outright because the footer extends beyond the binary's segments.

Skipping the Compatibility Check

For debugging purposes, extension developers can skip this check using the `allow_extensions_metadata_mismatch` setting. This lets `INSTALL` and `UPDATE EXTENSIONS` accept extensions with mismatching or invalid metadata. For `LOAD`, the setting only takes effect if `allow_unsigned_extensions` is also enabled:

```
SET allow_unsigned_extensions = true;
SET allow_extensions_metadata_mismatch = true;
LOAD './some/local/ext.duckdb_extension';
```

> **Warning.** Loading an extension built for a different DuckDB version or platform may crash DuckDB or cause undefined behavior.

Creating a Custom Repository

You can create a custom DuckDB extension repository. To have DuckDB trust signed extensions from such a repository without enabling `allow_unsigned_extensions`, register it with its public keys using `CREATE EXTENSION REPOSITORY`. A DuckDB repository is an HTTP, HTTPS, S3, or local file based directory that serves the extensions files in a specific structure. This structure is described in the "Downloading Extensions Directly from S3" section, and is the same for local paths and remote servers, for example:

```
base_repository_path_or_url
└── v1.0.0
    └── osx_arm64
        ├── autocomplete.duckdb_extension
        ├── httpfs.duckdb_extension
        ├── icu.duckdb_extension
        ├── inet.duckdb_extension
        ├── json.duckdb_extension
        ├── parquet.duckdb_extension
        ├── tpcds.duckdb_extension
        └── tpch.duckdb_extension
```

See the `extension-template` repository for all necessary code and scripts to set up a repository.

When installing an extension from a custom repository, DuckDB will search for both a gzipped and non-gzipped version. For example:

```
INSTALL icu FROM '<custom_repository>';
```

The execution of this statement will first look for `icu.duckdb_extension.gz`, then `icu.duckdb_extension` in the repository's directory structure.

If the custom repository is served over HTTPS or S3, the `httpfs` extension is required. DuckDB will attempt to autoload the `httpfs` extension when an installation over HTTPS or S3 is attempted.

Extension Repositories

> New Support for custom extension repositories was introduced in DuckDB 2.0. This feature is currently work-in-progress and the syntax may still change.

By default, DuckDB installs extensions from its built-in repositories (`core`, `core_nightly`, `community`, …). In addition, you can register your own trusted repositories, so that an organization can host and sign its own extensions and have them install and load just like the built-in ones.

A repository is a name, a URL prefix, and one or more RSA public keys that are trusted to sign the extensions served from it.

Enabling Custom Repositories

Adding custom repositories is gated by the `allow_extension_repositories` setting. It has three values:

- `undecided` (the default): adding new repositories is blocked, but existing repositories remain trusted.
- `allowed`: custom repositories can be added.
- `forbidden`: adding new repositories is blocked and existing custom repositories are distrusted.

While the database is running, the setting can only move from `undecided` to `allowed` or `forbidden`, or from `allowed` to `forbidden`. To opt in:

```
SET allow_extension_repositories = 'allowed';
```

Creating a Repository

The prefix can point at anything DuckDB can read: a local path, `https`, `s3`, and so on.

```
CREATE EXTENSION REPOSITORY my_repo
    WITH PREFIX 'https://extensions.example.org';
```

At CREATE time, DuckDB fetches the repository's public keys and pins them into the repository definition, printing each key's SHA-256 fingerprint so you can compare it against one published out of band.

If you would rather not trust the network, you can pass the key directly:

```
CREATE EXTENSION REPOSITORY my_repo
    WITH PREFIX 'https://extensions.example.org'
    USING PUBLIC KEY '-----BEGIN PUBLIC KEY----- ...';
```

Extensions are signed using 2048-bit RSA keys. The key can be provided either in PEM format or as the base64-encoded body of the PEM format.

Using a Repository

Once a repository is registered, install and load its extensions by referring to the repository by name:

```
INSTALL AND LOAD my_ext FROM my_repo;
```

Inspecting Repositories

Pinned repositories survive restarts, support key rotation by trusting multiple keys, and can be audited at any time through the `duckdb_extension_repositories()` table function:

```
SELECT repository_name, prefix, key_fingerprints
FROM duckdb_extension_repositories();
```

Dropping a Repository

```
DROP EXTENSION REPOSITORY my_repo;
```

Community Extensions

Community-contributed extensions can be installed from the Community Extensions repository since summer 2024. Please visit the "Community Extensions" section of the documentation for more details.

> **Warning.** Community extensions are contributed by third parties. They are not written, audited, or maintained by DuckLabs, and they run with the same privileges as DuckDB itself. Signing proves only that an extension was built by the Community Extension CI, not that its code is safe. Install and load community extensions only if you trust their author and source. See the security considerations for details.

> **Deprecated.** Starting with DuckDB 2.0, community extensions are no longer built for the `osx_amd64` platform (macOS on Intel). DuckDB and its core extensions are still available for `osx_amd64` in DuckDB 2.0, but DuckDB 2.1 will stop supporting this platform. See the list of supported platforms.

Versioning of Extensions

Extension Versioning

Most software has some sort of version number. Version numbers serve a few important goals:

- Tie a binary to a specific state of the source code
- Allow determining the expected feature set
- Allow determining the state of the APIs
- Allow efficient processing of bug reports (e.g., bug `#1337` was introduced in version `v3.4.5`)
- Allow determining chronological order of releases (e.g., version `v1.2.3` is older than `v1.2.4`)
- Give an indication of expected stability (e.g., `v0.0.1` is likely not very stable, whereas `v13.11.0` probably is stable)

Just like DuckDB itself, DuckDB extensions have their own version number. To ensure consistent semantics of these version numbers across the various extensions, DuckDB's Core Extensions use a versioning scheme that prescribes how extensions should be versioned. The versioning scheme for Core Extensions is made up of 3 different stability levels: **unstable**, **pre-release**, and **stable**. Let's go over each of the 3 levels and describe their format:

Unstable Extensions

Unstable extensions are extensions that can't (or don't want to) give any guarantees regarding their current stability, or their goals of becoming stable. Unstable extensions are tagged with the **short git hash** of the extension.

For example, an unstable extension may have a version such as `690bfc5`.

What to expect from an extension that has a version number in the **unstable** format?

- The state of the source code of the extension can be found by looking up the hash in the extension repository
- Functionality may change or be removed completely with every release
- This extension's API could change with every release
- This extension may not follow a structured release cycle, new (breaking) versions can be pushed at any time

Pre-Release Extensions

Pre-release extensions are the next step up from Unstable extensions. They are tagged with version in the **SemVer** format, more specifically, those in the `v0.y.z` format. In semantic versioning, versions starting with `v0` have a special meaning: they indicate that the more strict semantics of regular (`>v1.0.0`) versions do not yet apply. It basically means that an extension is working towards becoming a stable extension, but is not quite there yet.

For example, a pre-release extension may have a version such as `v0.1.0`.

What to expect from an extension that has a version number in the **pre-release** format?

- The extension is compiled from the source code corresponding to the tag.
- Semantic Versioning semantics apply. See the Semantic Versioning specification for details.
- The extension follows a release cycle where new features are tested in nightly builds before being grouped into a release and pushed to the `core` repository.
- Release notes describing what has been added each release should be available to make it easy to understand the difference between versions.

Stable Extensions

Stable extensions are the final step of extension stability. This is denoted by using a **stable SemVer** of format `vx.y.z` where `x>0`.

For example, a stable extension may have a version such as `v1.0.0`.

To check the versions of the extensions in your DuckDB installation, run:

```
SELECT extension_name, extension_version
FROM duckdb_extensions();
```

What to expect from an extension that has a version number in the **stable** format? Essentially the same as pre-release extensions, but now the more strict SemVer semantics apply: the API of the extension should now be stable and will only change in backwards incompatible ways when the major version is bumped. See the SemVer specification for details

Release Cycle of Pre-Release and Stable Core Extensions

In general for extensions the release cycle depends on their stability level. **unstable** extensions are often in sync with DuckDB's release cycle, but may also be quietly updated between DuckDB releases. **pre-release** and **stable** extensions follow their own release cycle. These may or may not coincide with DuckDB releases. To find out more about the release cycle of a specific extension, refer to the documentation or GitHub page of the respective extension. Generally, **pre-release** and **stable** extensions will document their releases as GitHub releases, an example of which you can see in the `delta` extension.

Finally, there is a small exception: All in-tree extensions simply follow DuckDB's release cycle.

Installing a Specific Extension Version

To install a specific version of an extension, add a `VERSION` clause to the `INSTALL` statement:

```
INSTALL <extension_name> VERSION '<version>';
```

The `VERSION` clause can be combined with a `FROM` clause that selects a repository, and with `FORCE INSTALL` to replace an already installed version:

```
FORCE INSTALL <extension_name> FROM core VERSION '<version>';
```

Pinning a version is mainly useful for extensions that publish their own versioned releases, i.e., pre-release and stable extensions. The requested version must be available for your DuckDB build and platform, otherwise the download fails with an HTTP error.

Nightly Builds

Just like DuckDB itself, DuckDB's core extensions have nightly or dev builds that can be used to try out features before they are officially released. This can be useful when your workflow depends on a new feature, or when you need to confirm that your stack is compatible with the upcoming version.

Nightly builds for extensions are slightly complicated due to the fact that currently DuckDB extensions binaries are tightly bound to a single DuckDB version. Because of this tight connection, there is a potential risk for a combinatorial explosion. Therefore, not all combinations of nightly extension build and nightly DuckDB build are available.

In general, there are 2 ways of using nightly builds: using a nightly DuckDB build and using a stable DuckDB build. Let's go over the differences between the two:

From Stable DuckDB

In most cases, users will be interested in a nightly build of a specific extension, but don't necessarily want to switch to using the nightly build of DuckDB itself. This allows using a specific bleeding-edge feature while limiting the exposure to unstable code.

To achieve this, Core Extensions tend to regularly push builds to the `core_nightly` repository. Let's look at an example:

First we install a **stable DuckDB build**.

Then we can install and load a **nightly** extension like this:

```
INSTALL AND LOAD aws FROM core_nightly;
```

In this example we are using the latest **nightly** build of the aws extension with the latest **stable** version of DuckDB.

From Nightly DuckDB

When DuckDB CI produces a nightly binary of DuckDB itself, the binaries are distributed with a set of extensions that are pinned at a specific version. This extension version will be tested for that specific build of DuckDB, but might not be the latest dev build. Let's look at an example:

First, we install a **nightly DuckDB build**. Then, we can install and load the aws extension as expected:

```
INSTALL AND LOAD aws;
```

Updating Extensions

DuckDB has a dedicated statement that will automatically update all extensions to their latest version. The output will give the user information on which extensions were updated to/from which version. For example:

```
UPDATE EXTENSIONS;
```

extension_name	repository	update_result	previous_version	current_version
httpfs	core	NO_UPDATE_AVAILABLE	70fd6a8a24	70fd6a8a24
delta	core	UPDATED	d9e5cc1	04c61e4
azure	core	NO_UPDATE_AVAILABLE	49b63dc	49b63dc
aws	core_nightly	NO_UPDATE_AVAILABLE	42c78d3	42c78d3

Note that DuckDB will look for updates in the source repository for each extension. So if an extension was installed from `core_nightly`, it will be updated with the latest nightly build.

The update statement can also be provided with a list of specific extensions to update:

```
UPDATE EXTENSIONS (httpfs, azure);
```

extension_name	repository	update_result	previous_version	current_version
httpfs	core	NO_UPDATE_AVAILABLE	70fd6a8a24	70fd6a8a24
azure	core	NO_UPDATE_AVAILABLE	49b63dc	49b63dc

Target DuckDB Version

Currently, when extensions are compiled, they are tied to a specific version of DuckDB. What this means is that, for example, an extension binary compiled for version 0.10.3 does not work for version 1.0.0. In most cases, this will not cause any issues and is fully transparent; DuckDB will automatically ensure it installs the correct binary for its version. For extension developers, this means that they must ensure that new binaries are created whenever a new version of DuckDB is released. However, note that DuckDB provides an extension template that makes this fairly simple.

In-Tree vs. Out-of-Tree

Originally, DuckDB extensions lived exclusively in the DuckDB main repository, `github.com/duckdb/duckdb`. These extensions are called in-tree. Later, the concept of out-of-tree extensions was added, where extensions were separated into their own repository, which we call out-of-tree.

While from a user's perspective, there are generally no noticeable differences, there are some minor differences related to versioning:

- in-tree extensions use the version of DuckDB instead of having their own version
- in-tree extensions do not have dedicated release notes, their changes are reflected in the regular DuckDB release notes
- core out-of tree extensions tend to live in repositories named `github.com/duckdb/duckdb-<extension_name>`{:.language-sql .highlight} but the name may vary. See the full list of core extensions for details.

Troubleshooting of Extensions

You might be visiting this page directed via a DuckDB error message, similar to:

```
INSTALL non_existing;
```

```
HTTP Error: Failed to download extension "non_existing" at URL "http://extensions.duckdb.org/v2.0.0/osx_
arm64/non_existing.duckdb_extension.gz" (HTTP 404)

Candidate extensions: "inet", "encodings", "core_functions", "vortex", "odbc_scanner"
For more info, visit
https://duckdb.org/docs/current/extensions/troubleshooting?version=v2.0.0&platform=osx_
arm64&extension=non_existing
```

There are multiple scenarios for which an extension might not be available in a given extension repository at a given time:

- the extension has not been uploaded yet, here some delay after a given release date might be expected. Consider checking the issues at duckdb/duckdb or duckdb/community-extensions, or creating one yourself.
- the extension is available, but in a different repository, try for example `INSTALL <name> FROM core;`{:.language-sql .highlight} or `INSTALL <name> FROM community;`{:.language-sql .highlight} or `INSTALL <name> FROM core_nightly;`{:.language-sql .highlight} (see the Installing Extensions page).
- networking issues, so extension exists at the endpoint but it's not reachable from your local DuckDB. Here you can try visiting the given URL via a browser directly pasting the link from the error message in the search bar.

If you are on a development version of DuckDB, that is any version for which `PRAGMA version` returns a `library_version` not starting with a `v`, then extensions might not be available anymore on the default extension repository.

When in doubt, consider raising an issue in duckdb/duckdb.

HTTP 451 Errors

An `HTTP 451` ("Unavailable For Legal Reasons") error when installing an extension means that the extension repository is not served to your location:

```
HTTP Error:
Failed to download extension "..." at URL "https://extensions.duckdb.org/..." (HTTP 451)
```

The default extension repository is distributed through a content delivery network that restricts access from certain regions for legal reasons, so the same `INSTALL` command can succeed from one network and fail from another. To work around this, you can:

- Connect from a network in a non-restricted region (for example, through a VPN).
- Download the extension manually from an allowed location and install it from the local file.
- Install from a different extension repository, such as a self-hosted mirror.

Manual Process to Download Extensions via the Browser

To check if an extension is available, consider trying to download the relevant extension resource, for example via your browser visiting https://extensions.duckdb.org/v2.0.0/osx_arm64/spatial.duckdb_extension.gz or any other link that has been provided. The extension server can be reached over both `http://` and `https://`. Note that DuckDB's built-in `core` and `community` repositories use `http://` by default: extension binaries are signed, so their integrity is verified on load regardless of the transport protocol.

If successful, this will download and unpack the extension to the default `Downloads` folder, so that from SQL you can run:

```
INSTALL '~/Downloads/spatial.duckdb_extension';
-- or
FORCE INSTALL '~/Downloads/spatial.duckdb_extension';
```

and after this command the extension will be regularly installed.

Core Extensions

Core Extensions

List of Core Extensions

Name	Description	Maintainer	Support tier	Aliases
autocomplete	Autocompletion for CLI client	DuckDB team	Secondary	
avro	Avro files reading	DuckDB team	Secondary	
aws	AWS S3 functionality	DuckDB team	Secondary	
azure	Filesystem abstraction for Azure Blob Storage	DuckDB team	Secondary	
delta	Delta Lake format	DuckDB team	Secondary	
ducklake	DuckLake format	DuckDB team	Secondary	
encodings	Read CSVs in 1000+ encodings	DuckDB team	Secondary	
excel	Excel (`.xlsx`) read/write	DuckDB team	Secondary	
fts	Full-text search	DuckDB team	Secondary	
httpfs	HTTP(S) and S3 - file read/write operations	DuckDB team	Primary	http, https, s3
iceberg	Apache Iceberg table format	DuckDB team	Secondary	
icu	Time zones, calendars and collations	DuckDB team	Primary	
inet	IP-related data types and functions	DuckDB team	Secondary	
json	JSON operations	DuckDB team	Primary	
lance	Lance tables read/write	Third party		
legacy_parser	Legacy PostgreSQL-derived SQL parser	DuckDB team	Secondary	
motherduck	MotherDuck connectivity	Third party		md
mysql	MySQL database read/write operations	DuckDB team	Secondary	mysql_scanner
odbc	ODBC connectivity	DuckDB team	Secondary	odbc_scanner
parquet	Parquet files read/write	DuckDB team	Primary	

Name	Description	Maintainer	Support tier	Aliases
postgres	PostgreSQL database read/write operations	DuckDB team	Secondary	postgres_scanner
quack	DuckDB-Quack protocol for remote access	DuckDB team	Secondary	
sirius	GPU-accelerated query execution (coming soon)	Third party		
spatial	Geospatial data and functions	DuckDB team	Secondary	
sqlite	SQLite database read/write operations	DuckDB team	Secondary	sqlite_scanner, sqlite3
tpcds	TPC-DS data generation and query	DuckDB team	Secondary	
tpch	TPC-H data generation and query	DuckDB team	Secondary	
unity_catalog	Unity Catalog connectivity	DuckDB team	Secondary	uc_catalog
ui	Local UI for DuckDB	Third party		
vortex	Vortex files read/write	Third party		
vss	Vector similarity search queries	DuckDB team	Secondary	

The **Maintainer** column denotes whether the extension is maintained by the DuckDB team or by a third party. For the extensions maintained by the DuckDB team, the **Support tier** column denotes the extension's support status. *Primary extensions* are covered by community support. *Secondary extensions* are supported on a best-effort basis. That said, they still receive frequent bugfixes/updates and are shipped with new DuckDB releases.

Backup Repository

The `core` repository is mirrored at `http://extensions.duckdb-backup.org`. If `extensions.duckdb.org` is unavailable, DuckDB automatically falls back to the backup server. To use the backup server manually, run:

```
SET custom_extension_repository = 'http://extensions.duckdb-backup.org';
```

For details, see Backup Extension Repositories.

AutoComplete Extension

The `autocomplete` extension adds support for autocomplete in the CLI client. The extension is shipped by default with the CLI client.

Behavior

For the behavior of the `autocomplete` extension, see the documentation of the CLI client.

Functions

Function	Description
`sql_auto_complete(query_string)`	Attempts autocompletion on the given `query_string`.

Example

```
SELECT *
FROM sql_auto_complete('SEL');
```

Returns:

suggestion	suggestion_start
SELECT	0
DELETE	0
INSERT	0
CALL	0
LOAD	0
CALL	0
ALTER	0
BEGIN	0
EXPORT	0
CREATE	0
PREPARE	0
EXECUTE	0
EXPLAIN	0
ROLLBACK	0
DESCRIBE	0
SUMMARIZE	0
CHECKPOINT	0

suggestion	suggestion_start
DEALLOCATE	0
UPDATE	0
DROP	0

Avro Extension

The `avro` extension enables DuckDB to read Apache Avro files.

> The `avro` extension was released as a community extension in late 2024 and became a core extension in early 2025.

The `read_avro` Function

The extension adds a single DuckDB function, `read_avro`. This function can be used like so:

```
FROM read_avro('<some_file>.avro');
```

This function will expose the contents of the Avro file as a DuckDB table. You can then use any arbitrary SQL constructs to further transform this table.

File IO

The `read_avro` function is integrated into DuckDB's file system abstraction, meaning you can read Avro files directly from, e.g., HTTP or S3 sources. For example:

```
FROM read_avro('https://blobs.duckdb.org/data/userdata1.avro');
FROM read_avro('s3://<your-bucket>/<some_file>.avro');
```

should "just" work.

You can also *glob* multiple files in a single read call or pass a list of files to the functions:

```
FROM read_avro('some_file_*.avro');
FROM read_avro(['some_file_1.avro', 'some_file_2.avro']);
```

If the filenames somehow contain valuable information (as is unfortunately all-too-common), you can pass the `filename` argument to `read_avro`:

```
FROM read_avro('some_file_*.avro', filename=true);
```

This will result in an additional column in the result set that contains the actual filename of the Avro file.

Schema Conversion

This extension automatically translates the Avro Schema to the DuckDB schema. *All* Avro types can be translated, except for *recursive type definitions*, which DuckDB does not support.

The type mapping is very straightforward except for Avro's "unique" way of handling `NULL`. Unlike other systems, Avro does not treat `NULL` as a possible value in a range of e.g. `INTEGER` but instead represents `NULL` as a union of the actual type with a special `NULL` type. This is different to DuckDB, where any value can be `NULL`. Of course DuckDB also supports `UNION` types, but this would be quite cumbersome to work with.

This extension *simplifies* the Avro schema where possible: An Avro union of any type and the special null type is simplified to just the non-null type. For example, an Avro record of the union type `["int","null"]` becomes a DuckDB `INTEGER`, which just happens to be `NULL` sometimes. Similarly, an Avro union that contains only a single type is converted to the type it contains. For example, an Avro record of the union type `["int"]` also becomes a DuckDB `INTEGER`.

The extension also "flattens" the Avro schema. Avro defines tables as root-level "record" fields, which are the same as DuckDB `STRUCT` fields. For more convenient handling, this extension turns the entries of a single top-level record into top-level columns.

Implementation

Internally, this extension uses the ”official” Apache Avro C API, albeit with some minor patching to allow reading of Avro files from memory.

Limitations and Future Plans

- This extension currently does not make use of **parallelism** when reading either a single (large) Avro file or when reading a list of files. Adding support for parallelism in the latter case is on the roadmap.
- There is currently no support for either projection or filter **pushdown**, but this is also planned at a later stage.
- There is currently no support for the Wasm or the Windows-MinGW builds of DuckDB due to issues with the Avro library dependency (sigh again). We plan to fix this eventually.
- As mentioned above, DuckDB cannot express recursive type definitions that Avro has, this is unlikely to ever change.
- There is no support to allow users to provide a separate Avro schema file. This is unlikely to change, all Avro files we have seen so far had their schema embedded.
- There is currently no support for the `union_by_name` flag that other readers in DuckDB support. This is planned for the future.

AWS Extension

The aws extension adds functionality, e.g., authentication, on top of the `httpfs` extension's S3 capabilities, using the AWS SDK.

Installing and Loading

The aws extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD aws;
```

> In most cases, the aws extension works in conjunction with the `httpfs` extension.

Configuration and Authentication

> **Tip.** For an overview of the available methods and guidance on which one to use, see the S3 and AWS Authentication page.

The preferred way to configure and authenticate to AWS S3 endpoints is to use secrets. There are two S3 secret providers:

- The `config` provider, where you supply the access key and secret manually. It is part of the `httpfs` extension and is documented in the S3 API page. Use it when you already have static credentials.
- The `credential_chain` provider, described below, which fetches credentials automatically using the AWS SDK. It is provided by the aws extension and supports profiles, SSO, assumed roles, web identities (IRSA), and instance metadata.

The full list of S3 secret parameters that apply to *both* providers (`ENDPOINT`, `REGION`, `URL_STYLE`, `USE_SSL`, `KMS_KEY_ID`, `REQUESTER_PAYS`, …) is documented in the S3 API page.

`credential_chain` Provider

The `credential_chain` provider allows automatically fetching credentials using mechanisms provided by the AWS SDK. For example, to use the AWS SDK default provider:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain
);
```

To query a file using the above secret, simply query any `s3://` prefixed file.

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'env;config'
);
```

The possible values for `CHAIN` are the following:

- `config`
- `sts`

- `sso`
- `env`
- `instance`
- `process`
- `web_identity` (for IAM Roles for Service Accounts (IRSA); see Web Identity (IRSA))

The `credential_chain` provider also allows overriding the automatically fetched config. For example, to automatically load credentials, and then override the region, run:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    REGION '<eu-west-1>'
);
```

Selecting a Profile

To load credentials based on a named profile which is not the default (from the `AWS_PROFILE` environment variable or the default profile based on AWS SDK precedence), use the `PROFILE` parameter:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    PROFILE '<my_profile>'
);
```

This approach is equivalent to the deprecated S3 API's method `load_aws_credentials('<my_profile>')`.

Assuming a Role (STS)

To assume an IAM role, pass its ARN via `ASSUME_ROLE_ARN`. An `EXTERNAL_ID` can be supplied for the role's trust policy:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'sts',
    ASSUME_ROLE_ARN 'arn:aws:iam::<account_id>:role/<role>',
    EXTERNAL_ID '<external_id>',
    REGION '<us-east-1>'
);
```

> The `sts` chain value requires an `ASSUME_ROLE_ARN` value. If the selected profile itself uses STS, use `CHAIN 'config'` instead.

Web Identity (IRSA)

For IAM Roles for Service Accounts (IRSA) — commonly used on Amazon EKS — use the `web_identity` chain together with a role ARN and a token file. A `SESSION_NAME` can optionally be set:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'web_identity',
    ASSUME_ROLE_ARN 'arn:aws:iam::<account_id>:role/<role>',
    WEB_IDENTITY_TOKEN_FILE '</var/run/secrets/eks.amazonaws.com/serviceaccount/token>'
);
```

Single Sign-On (SSO)

DuckDB can use credentials obtained through AWS IAM Identity Center (formerly AWS SSO). DuckDB does not perform the sign-in itself: it reads the short-lived token that the AWS CLI caches after you sign in. The setup therefore has three steps.

First, configure an SSO-enabled profile in `~/.aws/config`, either by running `aws configure sso` or by adding an `sso_session` and `sso_account_id` to the profile manually.

Second, sign in from the command line, which opens a browser to complete authentication and caches a temporary token locally:

```
aws sso login --profile <my-sso-profile>
```

Third, create a secret that uses the `sso` chain and points at that profile. DuckDB then picks up the cached token automatically:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'sso',
    PROFILE '<my-sso-profile>'
);
```

The token expires after the SSO session's configured lifetime. When it does, run `aws sso login` again. You do not need to recreate the secret.

HTTP Proxy

When credentials must be fetched through an HTTP proxy, configure it on the secret:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    HTTP_PROXY '<proxy.example.com:8080>',
    HTTP_PROXY_USERNAME '<username>',
    HTTP_PROXY_PASSWORD '<password>'
);
```

Region Resolution

If no region is provided explicitly, DuckDB resolves it from the following sources, in order:

1. The `REGION` secret parameter.
2. The `s3_region` setting (`SET s3_region = '<region>'`{:.language-sql .highlight}).
3. The `AWS_REGION` environment variable.
4. The `AWS_DEFAULT_REGION` environment variable.
5. The `region` of the profile in `~/.aws/config`.

If none of these resolve, `CREATE SECRET` still succeeds, but DuckDB logs a warning:

```
Set region explicitly using REGION 'us-east-1' in your CREATE SECRET statement, adding a region to your
profile in ~/.aws/config or configure the AWS_REGION or AWS_DEFAULT_REGION environment variables.
```

Validation

The AWS `credential_chain` provider will look for any required credentials during `CREATE SECRET` time, failing if absent/unavailable.

This behavior may be configured via the `VALIDATION` option as follows:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    VALIDATION 'exists'
);
```

Two validation modes are supported:

- `exists` (default) requires present credentials.
- none allows `CREATE SECRET` to succeed for `credential_chains` with no available credentials.

> `VALIDATION 'exists'` validates only the **presence** of a credential, **not its operational readiness**. Thus, no attempt is made to convert into an access token, or perform a read, write, etc.

Auto-Refresh

Some AWS endpoints require periodic refreshing of the credentials. This can be specified with the `REFRESH auto` option:

```
CREATE SECRET env_test (
    TYPE s3,
    PROVIDER credential_chain,
    REFRESH auto
);
```

> When `CHAIN` is `sts` or `web_identity`, `REFRESH auto` is enabled automatically, since these credentials are short-lived.

Amazon RDS (IAM Authentication)

The `aws` extension can generate short-lived IAM authentication tokens for connecting to Amazon RDS and Aurora databases, exposed through a secret of type `rds`. It accepts the same `credential_chain` options as an `s3` secret, plus the required `RDS_USER`, `RDS_HOST`, `RDS_PORT`, and `REGION` parameters. Unlike an `s3` secret, an `rds` secret also requires an explicit `CHAIN`:

```
CREATE SECRET aws_rds_secret (
    TYPE rds,
    PROVIDER credential_chain,
    CHAIN 'env;sso',
    REGION '<eu-west-1>',
    RDS_USER '<db_user>',
    RDS_HOST '<instance>.<identifier>.<region>.rds.amazonaws.com',
    RDS_PORT '5432'
);
```

> The `rds` secret *type* is registered by the `postgres` extension, not by `aws`, so the `postgres` extension must be installed and loaded before the statement above will run. For the complete end-to-end setup — passing the secret to a connection via `AWS_RDS_SECRET`, attaching, and querying — see the Amazon RDS with IAM authentication guide or the Postgres secrets documentation.

Legacy Features

> **Deprecated.** The `load_aws_credentials` function is deprecated and is removed in later releases. Use a secret with the `credential_chain` provider instead.

Prior to version 0.10.0, DuckDB did not have a Secrets manager, to load the credentials automatically, the AWS extension provided a special function to load the AWS credentials in the legacy authentication method.

Function	Type	Description
`load_aws_credentials`	PRAGMA function	Loads the AWS credentials through the AWS Default Credentials Provider Chain

Load AWS Credentials (Legacy)

To load the AWS credentials, run:

```
CALL load_aws_credentials();
```

loaded_access_key_id	loaded_secret_access_key	loaded_session_token	loaded_region
AKIAIOSFODNN7EXAMPLE	`<redacted>`	NULL	us-east-2

The function takes a string parameter to specify a specific profile:

```
CALL load_aws_credentials('minio-testing-2');
```

loaded_access_key_id	loaded_secret_access_key	loaded_session_token	loaded_region
minio_duckdb_user_2	`<redacted>`	NULL	NULL

There are several parameters to tweak the behavior of the call:

```
CALL load_aws_credentials('minio-testing-2', set_region = false, redact_secret = false);
```

loaded_access_key_id	loaded_secret_access_key	loaded_session_token	loaded_region
minio_duckdb_user_2	minio_duckdb_user_password_2	NULL	NULL

Azure Extension

The `azure` extension lets DuckDB read and write files in Azure Blob Storage and Azure Data Lake Storage (ADLS) Gen2.

Installing and Loading

The `azure` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD azure;
```

Usage

Queries need a secret; for a public container, a secret with the storage account name is enough (see the `CONFIG` provider). Then you can query Azure storage as follows:

Azure Blob Storage

Allowed URI schemes: `az` or `azure`

```
SELECT count(*)
FROM 'az://<my_container>/<path>/<my_file>.<parquet_or_csv>';
```

Globs are also supported:

```
SELECT *
FROM 'az://<my_container>/<path>/*.csv';
```

```
SELECT *
FROM 'az://<my_container>/<path>/**';
```

Or with a fully qualified path syntax:

```
SELECT count(*)
FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/<my_file>.<parquet_or_csv>';
```

```
SELECT *
FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/*.csv';
```

Azure Data Lake Storage (ADLS)

Allowed URI schemes: `abfss` or `abfs` (both connect over HTTPS)

```
SELECT count(*)
FROM 'abfss://<my_filesystem>/<path>/<my_file>.<parquet_or_csv>';
```

Globs are also supported:

```
SELECT *
FROM 'abfss://<my_filesystem>/<path>/*.csv';
```

```
SELECT *
FROM 'abfss://<my_filesystem>/<path>/**';
```

Or with a fully qualified path syntax:

```
SELECT count(*)
FROM 'abfss://<my_storage_account>.dfs.core.windows.net/<my_filesystem>/<path>/<my_file>.<parquet_or_
csv>';

SELECT *
FROM 'abfss://<my_storage_account>.dfs.core.windows.net/<my_filesystem>/<path>/*.csv';
```

The fully qualified path can also use the <my_filesystem>@<my_storage_account>{:.language-sql .highlight} form:

```
SELECT count(*)
FROM 'abfss://<my_filesystem>@<my_storage_account>.dfs.core.windows.net/<path>/<my_file>.<parquet_or_
csv>';
```

Writing to Azure Blob Storage

You can write data directly to Azure Blob or ADLSv2 Storage using the COPY statement.

```
-- Write query results to a Parquet file on Blob Storage
COPY (SELECT * FROM my_table)
TO 'az://<my_container>/<path>/output.parquet';

-- Write a table to a CSV file on ADLSv2 Storage
COPY my_table
TO 'abfss://<my_filesystem>/<path>/output.csv';
```

You can also use fully qualified paths:

```
COPY my_table
TO 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/output.parquet';
```

Since v1.5.4, the extension buffers written data locally and uploads it in blocks of `azure_write_block_size` bytes (default: 8 MiB). By default, the uploaded blocks are committed when the file is closed, so other readers see the new content only after the write completes. To commit earlier, set `azure_write_staged_blocks_per_commit` to the number of blocks after which an intermediate commit happens.

> **Warning.** An intermediate commit makes an incomplete file visible to readers. A Parquet file cannot be read until its footer is written at the end, so keep the default `0` for Parquet files and for Delta, Iceberg or DuckLake tables.

Writing Large Files

Azure limits a blob to 50,000 blocks. With the default block size of 8 MiB, the largest file you can write is about 390 GiB. To write larger files, increase `azure_write_block_size` (maximum: 4000 MiB, about 3.9 GiB). For example, to write files up to about 3 TiB:

```
SET azure_write_block_size = 64 * 1024 * 1024;
```

Configuration

Use the following configuration options to control how the extension reads and writes remote files:

Name	Description	Type	Default
`azure_http_stats`	Include HTTP info from Azure Storage in the `EXPLAIN ANALYZE` statement.	BOOLEAN	`false`

Name	Description	Type	Default
azure_http_logging	Write Azure HTTP requests to the DuckDB HTTP log when it is enabled. See Logging.	BOOLEAN	true
azure_http_logging_redact_query_params	Semicolon-separated list of URL query parameter names (case-sensitive) whose values are redacted in the HTTP log.	VARCHAR	sig
azure_http_logging_redact_headers	Semicolon-separated list of header names (case-insensitive) whose values are redacted in the HTTP log, for both requests and responses.	VARCHAR	Authorization
azure_read_transfer_concurrency	Maximum number of threads the Azure client can use for a single parallel read. Values above 1 let the client fill one buffer with parallel requests, when the chunk size is smaller than the buffer.	INTEGER	5
azure_read_transfer_chunk_size	Maximum size in bytes that the Azure client will read in a single request. Should be a divisor of azure_read_buffer_size.	BIGINT	8*1024*1024
azure_read_buffer_size	Size of the read buffer. Should be a multiple of azure_read_transfer_chunk_size.	UBIGINT	8*1024*1024
azure_write_block_size	Size in bytes of each block for Blob and ADLS writes. Setting it to 0 restores the default. The maximum is 4000 MiB (about 3.9 GiB). See Writing Large Files.	UBIGINT	8*1024*1024
azure_write_staged_blocks_per_commit	Number of uploaded blocks after which an intermediate commit happens. With 0, all blocks are committed when the file is closed. Non-zero values make partial writes visible sooner, at the cost of extra requests. This option does not change the 50,000-block limit.	UBIGINT	0
azure_transport_option_type	Underlying adapter to use in the Azure SDK. Valid values are: default or curl.	VARCHAR	default

Name	Description	Type	Default
`azure_context_caching`	Enable/disable the caching of the underlying Azure SDK HTTP connection in the DuckDB connection context when performing queries. If you suspect that this is causing some side effect, you can try to disable it by setting it to false (not recommended).	`BOOLEAN`	`true`

Since v1.5.4, `azure_read_transfer_chunk_size` and `azure_read_buffer_size` default to 8 MiB (previously 1 MiB).

> **Tip.** Setting `azure_transport_option_type` to `curl` has the following effects:
>
> - On Linux, it may fix certificate errors such as `Problem with the SSL CA cert (path? access rights?)`: the extension then looks for the CA bundle in the usual paths, which *curl* does not do by default and which static linking may get wrong.
> - On Windows, it replaces the default adapter (*WinHTTP*), so you can use all *curl* features, e.g., SOCKS proxies.
> - On all operating systems, it honors these environment variables:
>   - `CURL_CA_INFO`: path to a PEM encoded file with the certificate authorities for libcurl. It is known to work only on Linux and may fail on other platforms.
>   - `CURL_CA_PATH`: path to a directory of PEM encoded files with the certificate authorities for libcurl.

Example:

```
SET azure_http_stats = false;
SET azure_read_transfer_concurrency = 5;
SET azure_read_transfer_chunk_size = 8_388_608;
SET azure_read_buffer_size = 8_388_608;
```

Authentication

Authenticate with a secret. The SET variables in Authentication with Variables still work but are deprecated.

Authentication with Secret

Several Secret Providers are available, described below. These rules apply to every provider:

- If you need to define different secrets for different storage accounts, use the SCOPE configuration. Note that the SCOPE requires a trailing slash (`SCOPE 'az://<my_storage_account>.blob.core.windows.net/'{:.language-sql .highlight}`).
- With fully qualified paths, `ACCOUNT_NAME` is optional.
- For paths that are not fully qualified, the `ENDPOINT` attribute overrides the default endpoint (`blob.core.windows.net` for `az://` and `azure://`, `dfs.core.windows.net` for `abfss://` and `abfs://`).

`CONFIG` Provider The default provider, `CONFIG` (i.e., user-configured), allows access to the storage account using a connection string or anonymously. For example:

```
CREATE SECRET secret1 (
    TYPE azure,
    CONNECTION_STRING '<connection_string>'
);
```

For anonymous access, give the storage account name:

```
CREATE SECRET secret2 (
    TYPE azure,
    PROVIDER config,
    ACCOUNT_NAME '<storage_account_name>'
);
```

`credential_chain` Provider The `credential_chain` provider allows connecting using credentials automatically fetched by the Azure SDK via the Azure credential chain. By default, the provider uses the Azure SDK's `DefaultAzureCredential` chain. The token is fetched at query time, not when the secret is created. For example:

```
CREATE SECRET secret3 (
    TYPE azure,
    PROVIDER credential_chain,
    ACCOUNT_NAME '<storage_account_name>'
);
```

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE SECRET secret4 (
    TYPE azure,
    PROVIDER credential_chain,
    CHAIN 'cli;env',
    ACCOUNT_NAME '<storage_account_name>'
);
```

The possible values are:

- `cli`: the Azure CLI login
- `managed_identity`
- `workload_identity`, e.g., on AKS
- `env`: environment variables
- `default`: the full default chain, used when no `CHAIN` is given

`MANAGED_IDENTITY` Provider If the machine has one managed identity, the `credential_chain` provider uses it with no extra options. If it has several, use the `MANAGED_IDENTITY` provider and pick one with `CLIENT_ID`, `OBJECT_ID` or `RESOURCE_ID`:

```
CREATE SECRET azure_mi (
    TYPE azure,
    PROVIDER managed_identity,
    ACCOUNT_NAME '<storage_account_name>',
    CLIENT_ID '<client_id>'
);
```

Without an ID, the provider works like `credential_chain`. With several identities and no ID, the Azure SDK decides which one is used, so set an ID.

`SERVICE_PRINCIPAL` Provider The `SERVICE_PRINCIPAL` provider allows connecting using an Azure Service Principal (SPN).

With a client secret:

```
CREATE SECRET azure_spn (
    TYPE azure,
    PROVIDER service_principal,
    TENANT_ID '<tenant_id>',
    CLIENT_ID '<client_id>',
    CLIENT_SECRET '<client_secret>',
    ACCOUNT_NAME '<storage_account_name>'
);
```

Or with a certificate:

```
CREATE SECRET azure_spn_cert (
    TYPE azure,
    PROVIDER service_principal,
    TENANT_ID '<tenant_id>',
    CLIENT_ID '<client_id>',
    CLIENT_CERTIFICATE_PATH '<client_cert_path>',
    ACCOUNT_NAME '<storage_account_name>'
);
```

ACCESS_TOKEN Provider The ACCESS_TOKEN provider allows connecting with an OAuth access token (bearer token) that you obtained outside of DuckDB:

```
CREATE SECRET azure_token (
    TYPE azure,
    PROVIDER access_token,
    ACCESS_TOKEN '<access_token>',
    ACCOUNT_NAME '<storage_account_name>'
);
```

> **Warning.** The extension does not refresh the token. When it expires, run `CREATE OR REPLACE SECRET` with a new token; it takes effect on the next query.

Configuring a Proxy To configure proxy information when using secrets, you can add HTTP_PROXY, PROXY_USER_NAME and PROXY_PASSWORD in the secret definition. For example:

```
CREATE SECRET secret5 (
    TYPE azure,
    CONNECTION_STRING '<connection_string>',
    HTTP_PROXY 'http://localhost:3128',
    PROXY_USER_NAME '<proxy_user_name>',
    PROXY_PASSWORD '<proxy_password>'
);
```

> **Note.** When you use secrets:
>
> - The HTTP_PROXY environment variable is still honored, unless the secret sets HTTP_PROXY.
> - The SET variables of Authentication with Variables are ignored.

Authentication with Variables

> **Deprecated.** Use secrets instead.

```
SET <variable_name> = <value>;
```

Where `variable_name` can be one of the following:

Name	Description	Type	Default
`azure_storage_connection_string`	Azure connection string, used for authenticating and configuring Azure requests.	VARCHAR	-
`azure_account_name`	Azure account name, when set, the extension will attempt to automatically detect credentials (not used if you pass the connection string).	VARCHAR	-

Name	Description	Type	Default
`azure_endpoint`	Override the Azure endpoint for when the Azure credential providers are used.	VARCHAR	`blob.core.windows.net`
`azure_credential_chain`	Ordered list of Azure credential providers, in string format separated by `;`. For example: `'cli;managed_identity;env'`. See the list of possible values in the `credential_chain` provider section. Not used if you pass the connection string.	VARCHAR	-
`azure_http_proxy`	Proxy for login and for requests to Azure.	VARCHAR	HTTP_PROXY environment variable (if set).
`azure_proxy_user_name`	HTTP proxy username if needed.	VARCHAR	-
`azure_proxy_password`	HTTP proxy password if needed.	VARCHAR	-

Additional Information

Logging

Since v1.5.2, the Azure extension writes its HTTP requests to the DuckDB HTTP log. To inspect them:

```
CALL enable_logging('HTTP');
SELECT * FROM 'az://<my_container>/<path>/<my_file>.parquet';
SELECT request.type, request.url, response.status FROM duckdb_logs_parsed('HTTP');
```

Logged requests show the DuckDB `User-Agent` header. The values of the `sig` query parameter and the `Authorization` header are replaced by REDACTED. To change what is redacted, set `azure_http_logging_redact_query_params` and `azure_http_logging_redact_headers`. A new value replaces the default list, so include `sig` or `Authorization` if you still want them redacted:

```
SET azure_http_logging_redact_headers = 'Authorization;x-ms-date';
```

> **Warning.** The Azure extension applies only these settings, not DuckDB's `redact_http_logs` setting. To keep Azure requests out of the HTTP log, run `SET azure_http_logging = false;`.

The Azure extension relies on the Azure SDK to connect to Azure Blob storage and supports printing the SDK logs to the console. To control the log level, set the `AZURE_LOG_LEVEL` environment variable.

For instance, to start the DuckDB CLI with verbose logs:

```
AZURE_LOG_LEVEL=verbose duckdb
```

```
CREATE SECRET myaccount (
    TYPE azure,
    PROVIDER credential_chain,
    SCOPE 'az://<my_storage_account>.blob.core.windows.net/'
);
SELECT count(*) FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/<my_
file>.parquet';
```

Difference between ADLS and Blob Storage

For glob patterns over many directories, prefer `abfss://`: the ADLS endpoint lists only the directories that match, while the Blob endpoint lists every file under the prefix and filters afterwards. The following example shows how a glob is performed through each endpoint.

Using the following filesystem:

```
root
├── l_receiptmonth=1997-10
│   ├── l_shipmode=AIR
│   │   └── data_0.csv
│   ├── l_shipmode=SHIP
│   │   └── data_0.csv
│   └── l_shipmode=TRUCK
│       └── data_0.csv
├── l_receiptmonth=1997-11
│   ├── l_shipmode=AIR
│   │   └── data_0.csv
│   ├── l_shipmode=SHIP
│   │   └── data_0.csv
│   └── l_shipmode=TRUCK
│       └── data_0.csv
└── l_receiptmonth=1997-12
    ├── l_shipmode=AIR
    │   └── data_0.csv
    ├── l_shipmode=SHIP
    │   └── data_0.csv
    └── l_shipmode=TRUCK
        └── data_0.csv
```

The following query is performed through the Blob endpoint:

```
SELECT count(*)
FROM 'az://root/l_receiptmonth=1997-*/l_shipmode=SHIP/*.csv';
```

It will perform the following steps:

- List all the files with the prefix `root/l_receiptmonth=1997-`

  - `root/l_receiptmonth=1997-10/l_shipmode=SHIP/data_0.csv`
  - `root/l_receiptmonth=1997-10/l_shipmode=AIR/data_0.csv`
  - `root/l_receiptmonth=1997-10/l_shipmode=TRUCK/data_0.csv`
  - `root/l_receiptmonth=1997-11/l_shipmode=SHIP/data_0.csv`
  - `root/l_receiptmonth=1997-11/l_shipmode=AIR/data_0.csv`
  - `root/l_receiptmonth=1997-11/l_shipmode=TRUCK/data_0.csv`
  - `root/l_receiptmonth=1997-12/l_shipmode=SHIP/data_0.csv`
  - `root/l_receiptmonth=1997-12/l_shipmode=AIR/data_0.csv`
  - `root/l_receiptmonth=1997-12/l_shipmode=TRUCK/data_0.csv`

- Filter the result with the requested pattern `root/l_receiptmonth=1997-*/l_shipmode=SHIP/*.csv`

  - `root/l_receiptmonth=1997-10/l_shipmode=SHIP/data_0.csv`
  - `root/l_receiptmonth=1997-11/l_shipmode=SHIP/data_0.csv`
  - `root/l_receiptmonth=1997-12/l_shipmode=SHIP/data_0.csv`

Meanwhile, the same query can be performed through the datalake endpoint as follows:

```
SELECT count(*)
FROM 'abfss://root/l_receiptmonth=1997-*/l_shipmode=SHIP/*.csv';
```

This will perform the following steps:

- List all directories in `root/`

```
- root/l_receiptmonth=1997-10
- root/l_receiptmonth=1997-11
- root/l_receiptmonth=1997-12
```

- Filter and list subdirectories: `root/l_receiptmonth=1997-10`, `root/l_receiptmonth=1997-11`, `root/l_receiptmonth=1997-12`

```
- root/l_receiptmonth=1997-10/l_shipmode=SHIP
- root/l_receiptmonth=1997-10/l_shipmode=AIR
- root/l_receiptmonth=1997-10/l_shipmode=TRUCK
- root/l_receiptmonth=1997-11/l_shipmode=SHIP
- root/l_receiptmonth=1997-11/l_shipmode=AIR
- root/l_receiptmonth=1997-11/l_shipmode=TRUCK
- root/l_receiptmonth=1997-12/l_shipmode=SHIP
- root/l_receiptmonth=1997-12/l_shipmode=AIR
- root/l_receiptmonth=1997-12/l_shipmode=TRUCK
```

- Filter and list subdirectories: `root/l_receiptmonth=1997-10/l_shipmode=SHIP`, `root/l_receiptmonth=1997-11/l_shipmode=SHIP`, `root/l_receiptmonth=1997-12/l_shipmode=SHIP`

```
- root/l_receiptmonth=1997-10/l_shipmode=SHIP/data_0.csv
- root/l_receiptmonth=1997-11/l_shipmode=SHIP/data_0.csv
- root/l_receiptmonth=1997-12/l_shipmode=SHIP/data_0.csv
```

Because the Blob endpoint does not support the notion of directories, the filter can only be performed after the listing, whereas the ADLS endpoint will list files recursively. Especially with higher partition/directory counts, the performance difference can be very significant.

Delta Extension

The `delta` extension adds support for the Delta Lake open-source storage format. It is built using the Delta Kernel. The extension reads Delta tables, local and remote, and writes them: it can create tables and append to them.

For implementation details, see the "Native Delta Lake Support in DuckDB" blog post.

> To connect to Unity Catalog, DuckDB has the `unity_catalog` extension.

Installing and Loading

The `delta` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD delta;
```

Usage

To scan a local Delta table, run:

```
SELECT *
FROM delta_scan('file:///<path>');
```

Attaching Tables

A Delta table can also be attached as a database. The database holds exactly one table, with the same name as the database, so `my_table` and `my_table.my_table` refer to the same table:

```
ATTACH 'file:///<path>' AS my_table (TYPE delta);
SELECT * FROM my_table;
```

Reading from an S3 Bucket

To scan a Delta table in an S3 bucket, run:

```
SELECT *
FROM delta_scan('s3://<my_bucket>/<path>');
```

To authenticate, create a secret:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
SELECT *
FROM delta_scan('s3://<my_bucket>/<path>');
```

For a public bucket, you may need to set its region in a secret:

```
CREATE SECRET (
    TYPE s3,
    REGION '<my_region>'
);
SELECT *
FROM delta_scan('s3://<my_bucket>/<path>');
```

Reading from Azure Blob Storage

To scan a Delta table in an Azure Blob Storage container, run:

```
SELECT *
FROM delta_scan('az://<my_container>/<path>');
```

To authenticate, create a secret:

```
CREATE SECRET (
    TYPE azure,
    PROVIDER credential_chain,
    ACCOUNT_NAME '<storage_account_name>'
);
SELECT *
FROM delta_scan('az://<my_container>/<path>');
```

Reading from Google Cloud Storage

To scan a Delta table in a GCS bucket through its S3 API compatibility, use HMAC keys and create a secret:

```
CREATE SECRET (
    TYPE gcs,
    KEY_ID '<hmac_key_id>',
    SECRET '<hmac_secret>'
);
SELECT *
FROM delta_scan('gs://<my_bucket>/<path>');
```

Creating Tables

> New This feature was introduced in DuckDB 2.0.

To create a Delta table, attach the location for it and create a table named after the attached database:

```
ATTACH 's3://<my_bucket>/<path>' AS events (TYPE delta);
CREATE TABLE events.events (id BIGINT, region VARCHAR)
PARTITIONED BY (region);
INSERT INTO events VALUES (1, 'eu');
```

Only `CREATE TABLE` needs the full `events.events` name: without the database name, it creates the table in the default database.

The table can be read and written without attaching it again.

> **Warning.** ATTACH does not check that the path holds a Delta table. A mistyped path attaches as an empty database and only fails on the first read.

Table Properties Options in a `WITH` clause are stored as Delta table properties, except `location`, which must match the attached path:

```
ATTACH 's3://<my_bucket>/<other_path>' AS audit (TYPE delta);
CREATE TABLE audit.audit (id BIGINT, name VARCHAR)
WITH ('delta.enableDeletionVectors' = 'true', 'delta.appendOnly' = 'true');
```

Properties that enable a Delta feature, such as deletion vectors, also upgrade the table's protocol as needed. Values can be any constant expression except `NULL`. Some properties cannot be set at creation, such as `'delta.enableVariantShredding'`.

To create a table with column mapping, set `'delta.columnMapping.mode'` to `'name'` or `'id'`.

Appending Data

To append rows to a Delta table, attach it and use `INSERT INTO`:

```
ATTACH 's3://<my_bucket>/<path>' AS my_table (TYPE delta);
INSERT INTO my_table SELECT * FROM other_table;
```

Since v1.5.6, if the table schema declares `CHAR(n)` or `VARCHAR(n)` (Spark writes these), a write with a value longer than n characters fails as a whole.

Time Travel

> New Time travel by timestamp was introduced in DuckDB 2.0.

To read a table as it was at an earlier version or point in time, use the `AT` clause, an `ATTACH` option or a `delta_scan` parameter:

```
-- Query an attached table
SELECT * FROM my_table AT (VERSION => 5);
SELECT * FROM my_table AT (TIMESTAMP => TIMESTAMPTZ '2026-09-01 12:00:00+00');

-- Attach the table at a version or point in time
ATTACH 's3://<my_bucket>/<path>' AS my_table_v5 (TYPE delta, VERSION 5);
ATTACH 's3://<my_bucket>/<path>' AS my_table_then (TYPE delta, TIMESTAMP TIMESTAMPTZ '2026-09-01
12:00:00+00');

-- Scan without attaching
SELECT * FROM delta_scan('s3://<my_bucket>/<path>', version => 5); -- since v1.5.4
SELECT * FROM delta_scan('s3://<my_bucket>/<path>', timestamp => TIMESTAMPTZ '2026-09-01 12:00:00+00');
```

- A table attached at a version or timestamp is read-only.
- A version or timestamp given to `ATTACH` is the default for queries on that table; an `AT` clause overrides it.
- A version and a timestamp cannot be given together.
- A timestamp reads the latest version committed at or before it. One later than `now()` or earlier than the table's first commit fails.
- A timestamp without a time zone uses the session's `TimeZone`.
- Within a transaction, a timestamp always reads the same version.

Checkpointing

Since v1.5.1, `CHECKPOINT` compacts the Delta log of an attached table into a checkpoint file, which speeds up reads of tables with long logs:

```
ATTACH '<path>' AS my_table (TYPE delta);
CHECKPOINT my_table;
```

Idempotent Appends

To retry a batch without writing it twice, tag the append with an application ID and a version. DuckDB commits the append only if the table's last recorded version for that ID is the one you expect. This uses Delta's per-application transaction versions.

Within a transaction, tag the append with `delta_set_transaction_version(<table>, <app_id>, <new_version>, <expected_previous_version>)`{:.language-sql .highlight}:

```
ATTACH '<path>' AS my_table (TYPE delta);

BEGIN TRANSACTION;
CALL delta_set_transaction_version('my_table', 'my_app_id', 1::UBIGINT, NULL::UBIGINT);
INSERT INTO my_table VALUES (1);
COMMIT;
```

If another process committed a different version for `my_app_id` first, `COMMIT` fails. A rolled-back transaction leaves the version unchanged. Read the current version with `delta_get_transaction_version(<table>, <app_id>)`{:.language-sql .highlight}, which returns `NULL` if no version has been recorded yet.

Attach Options

When attaching a Delta table you can pass the following options to `ATTACH`:

Option	Type	Default	Description
`VERSION`	`UBIGINT`	latest	Attach the table at a specific table version.
`TIMESTAMP`	`TIMESTAMP WITH TIME ZONE`	latest	Attach the table at the version current at a point in time.
`PIN_SNAPSHOT`	`BOOLEAN`	`false`	Resolve the latest table snapshot once, on first use, and reuse it for every later query. Without it, each transaction reads the latest version; since v1.5.3, it loads only the log entries committed since the previous read.
`PUSHDOWN_PARTITION_INFO`	`BOOLEAN`	`true`	Push down partition information so that whole files can be skipped based on partition values.
`PUSHDOWN_FILTERS`	`VARCHAR`	`all`	Filter pushdown mode for file skipping. One of `none`, `all`, `constant_only`, `dynamic_only`.

```
ATTACH 's3://<my_bucket>/<path>' AS my_table (
    TYPE delta,
    PIN_SNAPSHOT true,
    PUSHDOWN_FILTERS 'constant_only'
);
```

Inspecting Scanned Files

`delta_list_files` returns the data files a scan would read for a table, together with their cardinality, partition values, and whether they carry deletion vectors. This is useful for understanding the effect of data skipping:

```
SELECT * FROM delta_list_files('file:///<path>');
```

Column	Type	Description
`data_file`	`VARCHAR`	Path to the Parquet data file.
`cardinality`	`UBIGINT`	Number of rows in the file.
`partitions`	`MAP(VARCHAR, VARCHAR)`	Partition column values for the file.

Column	Type	Description
have_deletes	BOOLEAN	Whether the file has an associated deletion vector.

To list the files of an earlier table version, pass `version => <version>`{:.language-sql .highlight}. To add a `delete_count` column with the number of rows each file's deletion vector removes, pass `delete_count => true`:

```
SELECT data_file, cardinality, delete_count
FROM delta_list_files('file:///<path>', version => 3, delete_count => true);
```

Reading Domain Metadata

Since v1.5.1, `delta_domain_metadata` returns one row per domain of a table, with the VARCHAR columns `domain` and `configuration`. Pass `version => <version>`{:.language-sql .highlight} for an earlier version:

```
SELECT domain, configuration
FROM delta_domain_metadata('file:///<path>');
```

For an attached table, `duckdb_tables()` shows the same pairs in its `tags` column.

Column Mapping

Delta tables with column mapping enabled are read as the Delta protocol specifies for each mode: in `id` mode, columns are matched by Parquet field ID; in `name` mode, by physical column name.

The protocol requires every data file of an `id`-mode table to carry Parquet field IDs, but some writers omit them, including DuckDB 1.5.5 and earlier. Since v1.5.6, the `delta_column_mapping_policy` setting decides how such files are read:

- `strict`: fail with an error that names the file.
- `lenient`: match the file's columns by physical name, or failing that by logical name, and log a warning under the `delta.ColumnMapping` log type. The names must cover every column in the file, otherwise the read fails.

> **Warning.** The default changed from `lenient` in DuckDB 1.5 to `strict` in v2.0. To read such files as v1.5 did, run `SET delta_column_mapping_policy = 'lenient';`{:.language-sql .highlight}.

VARIANT Columns

Delta `variant` columns are read as DuckDB `VARIANT`, whether their files are shredded or not, and top-level `VARIANT` columns can be appended to.

> New The following were introduced in DuckDB 2.0.

- A query that reads one field of a `VARIANT` or `STRUCT` column, such as `SELECT v.a`, reads only that field from the Parquet files, so on a shredded table it skips the columns of the other fields. This does not apply to a column with a `TIMESTAMP WITH TIME ZONE` field at any depth: DuckDB reads all of it.
- `CREATE TABLE` accepts `VARIANT` columns.
- When writing, DuckDB shreds `VARIANT` values only if the table sets `'delta.enableVariantShredding'` to `'true'`, as the Delta protocol requires; otherwise it writes them unshredded. Because `CREATE TABLE` cannot set this property, a table created by DuckDB holds unshredded files.

Catalog-Managed Tables

A Delta table with the `catalogManaged` feature, such as a Unity Catalog managed table, has commits that only its catalog tracks. Reading or writing such a table straight from storage, with `delta_scan` or `ATTACH ... (TYPE delta)`, fails with an error. Attach its catalog instead, for example with the `unity_catalog` extension, and query the table through it.

Credential Chains in Delta

For some network operations, the `delta` extension uses `delta-kernel-rs` and `object_store`, which try credential sources in a different order than DuckDB and include different ones by default. If more than one source is available, e.g., a service principal in the environment and the Azure CLI, credential loading may be inconsistent. Configure exactly one credential type in the chain of your secret.

Since v1.5.6, the `delta` extension supports Azure workload identity (for example on AKS): name `workload_identity` in the `CHAIN` of an Azure `credential_chain` secret, and set the `AZURE_FEDERATED_TOKEN_FILE`, `AZURE_CLIENT_ID` and `AZURE_TENANT_ID` environment variables.

Settings

The `delta` extension adds the following settings:

Setting	Type	Default	Description
`delta_column_mapping_policy`	`VARCHAR`	`strict`	How to read an `id`-mode data file without Parquet field IDs: `strict` raises an error, `lenient` matches its columns by name. See Column Mapping.
`delta_kernel_logging`	`BOOLEAN`	`false`	Forward the internal logging of the Delta Kernel to the DuckDB logger. May impact performance even when DuckDB logging is disabled.
`delta_scan_explain_files_filtered`	`BOOLEAN`	`true`	Add the filtered files to the `EXPLAIN` output. May impact the performance of `delta_scan` during `EXPLAIN ANALYZE` queries.

Features

The `delta` extension supports:

- multithreaded scans and Parquet metadata reading
- data skipping/filter pushdown
  - skipping row groups in file (based on Parquet metadata)
  - skipping complete files (based on Delta partition information and file statistics)
- projection pushdown, down to single fields of struct and `VARIANT` columns
- scanning tables with deletion vectors
- column mapping in `name` and `id` mode
- time travel by version or timestamp

- all primitive types except intervals
- nested types (structs, lists and maps)
- creating tables (`CREATE TABLE`)
- blind appends (`INSERT INTO`), with file statistics for data skipping
- idempotent appends
- checkpoints (`CHECKPOINT`)
- cloud storage (AWS S3, Azure, GCS) with secrets

Limitations

- `CREATE OR REPLACE TABLE` and `CREATE TABLE ... AS SELECT` are not supported. Create the table, then `INSERT` into it.
- `UPDATE`, `DELETE`, `MERGE INTO`, `ALTER TABLE` and `DROP TABLE` are not supported, nor are `RETURNING` and `ON CONFLICT` clauses on `INSERT`.
- `CREATE TABLE` does not support:
  - constraints, including `NOT NULL`
  - `DEFAULT` values and generated columns
  - `SORTED BY`
  - column types without a Delta equivalent, such as `TIME` or unsigned integers
  - a table whose columns are all partition columns
- A table created with column mapping cannot have nested or partition columns, because DuckDB cannot write to such tables yet.
- Tables with an interval (`interval year to month` or `interval day to second`), `geometry` or `geography` column cannot be read.
- Writing to a table with the `allowColumnDefaults` writer feature fails, even if no column has a default yet: DuckDB cannot fill in Delta defaults and does not write `NULL` instead. Reading works.
- Writing to a table that declares a `CHAR(n)` or `VARCHAR(n)` width on a nested field, such as `ARRAY<VARCHAR(5)>`, is not supported.
- Writing to a table with a `VARIANT` inside a struct, list or map is not supported.

Supported Platforms

The `delta` extension currently only supports the following platforms:

- Linux (x86_64 and ARM64): `linux_amd64` and `linux_arm64`
- macOS Intel and Apple Silicon: `osx_amd64` and `osx_arm64`
- Windows AMD64: `windows_amd64`

Support for the other DuckDB platforms is work-in-progress.

Using delta-rs with DuckDB

This example writes a Delta table with the `delta-rs` Python package and reads it with DuckDB. It then reads the table's change data feed through the Arrow zero-copy integration. For larger data, use Arrow Datasets to read it lazily.

The example needs `deltalake`, the Python package of `delta-rs`, with its `pyarrow` extra:

```
pip install duckdb "deltalake[pyarrow]"
```

DuckLake

> DuckLake 1.1 was released together with DuckDB 2.0. Read the announcement blog post.

The `ducklake` extension adds support for attaching to databases stored in the DuckLake format. The complete documentation of this extension is available at the DuckLake website.

Installing and Loading

To install `ducklake`, run:

```
INSTALL ducklake;
```

The `ducklake` extension will be transparently autoloaded on first use in an `ATTACH` clause. If you would like to load it manually, run:

```
LOAD ducklake;
```

Usage

```
ATTACH 'ducklake:metadata.ducklake' AS my_ducklake (DATA_PATH 'data_files');
USE my_ducklake;
```

Tables

In DuckDB, the `ducklake` extension stores the catalog tables for a DuckLake named `my_ducklake`{:.language-sql .highlight} in the `__ducklake_metadata_<my_ducklake>`{:.language-sql .highlight} catalog.

Functions

Note that DuckLake registers several functions. These should be called with the catalog name as the first argument, e.g.:

```
FROM ducklake_snapshots('my_ducklake');
```

snapshot_id	snapshot_time	schema_version	changes	author	commit_message	commit_extra_info
int64	timestamp with time zone	int64	map(varchar, varchar[])	varchar	varchar	varchar
0	2026-10-20 23:56:36.256+02	0	{schemas_created=[main]}	NULL	NULL	NULL

`ducklake_snapshots`

Returns the snapshots stored in the DuckLake catalog name `catalog`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	

The information is encoded into a table with the following schema:

Column name	Column type
`snapshot_id`	`BIGINT`
`snapshot_time`	`TIMESTAMP WITH TIME ZONE`
`schema_version`	`BIGINT`
`changes`	`MAP(VARCHAR, VARCHAR[])`
`author`	`VARCHAR`
`commit_message`	`VARCHAR`
`commit_extra_info`	`VARCHAR`

`ducklake_table_info`

The `ducklake_table_info` function returns information on the tables stored in the DuckLake catalog named `catalog`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	

The information is encoded into a table with the following schema:

Column name	Column type
`table_name`	`VARCHAR`
`schema_id`	`BIGINT`
`table_id`	`BIGINT`
`table_uuid`	`UUID`
`file_count`	`BIGINT`
`file_size_bytes`	`BIGINT`
`delete_file_count`	`BIGINT`
`delete_file_size_bytes`	`BIGINT`

`ducklake_table_insertions`

The `ducklake_table_insertions` function returns the rows inserted in a given table between snapshots of given versions or timestamps. The function has two variants, depending on whether `start_snapshot` and `end_snapshot` have types `BIGINT` or `TIMESTAMP WITH TIME ZONE`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`schema_name`	`VARCHAR`	no	
`table_name`	`VARCHAR`	no	
`start_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	
`end_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	

The schema of the table returned by the function is equivalent to that of the table `table_name`.

`ducklake_table_deletions`

The `ducklake_table_deletions` function returns the rows deleted from a given table between snapshots of given versions or timestamps. The function has two variants, depending on whether `start_snapshot` and `end_snapshot` have types `BIGINT` or `TIMESTAMP WITH TIME ZONE`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`schema_name`	`VARCHAR`	no	
`table_name`	`VARCHAR`	no	
`start_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	
`end_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	

The schema of the table returned by the function is equivalent to that of the table `table_name`.

`ducklake_table_changes`

The `ducklake_table_changes` function returns the rows changed in a given table between snapshots of given versions or timestamps. The function has two variants, depending on whether `start_snapshot` and `end_snapshot` have types `BIGINT` or `TIMESTAMP WITH TIME ZONE`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`schema_name`	`VARCHAR`	no	
`table_name`	`VARCHAR`	no	
`start_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	
`end_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	

The schema of the table returned by the function contains the following three columns plus the schema of the table `table_name`.

Column name	Column type	Description
`snapshot_id`	`BIGINT`	

Column name	Column type	Description
`rowid`	`BIGINT`	
`change_type`	`VARCHAR`	The type of change: `insert` or `delete`

Commands

`ducklake_cleanup_old_files`

The `ducklake_cleanup_old_files` function cleans up old files in the DuckLake denoted by `catalog`. It returns a table with a single column (`path`) listing the files that were deleted (or, with `dry_run`, would be deleted).

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`cleanup_all`	`BOOLEAN`	yes	
`dry_run`	`BOOLEAN`	yes	
`older_than`	`TIMESTAMP WITH TIME ZONE`	yes	

`ducklake_expire_snapshots`

The `ducklake_expire_snapshots` function expires snapshots with the versions specified by the `versions` parameter or the ones older than the `older_than` parameter. It returns the expired snapshots, using the same columns as `ducklake_snapshots`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`versions`	`BIGINT[]`	yes	
`older_than`	`TIMESTAMP WITH TIME ZONE`	yes	

`ducklake_merge_adjacent_files`

The `ducklake_merge_adjacent_files` function merges adjacent files in the storage. It returns one row per table with the columns `schema_name`, `table_name`, `files_processed` and `files_created`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	

Compatibility Matrix

The DuckLake specification and the `ducklake` DuckDB extension are currently released together. See the Compatibility Matrix.

Encodings Extension

The `encodings` extension adds support for reading CSVs using more than 1,000 character encodings.

For a complete list of supported encodings, see All Supported Encodings. For detailed information on character encoding, see the ICU data repository.

Installing and Loading

```
INSTALL AND LOAD encodings;
```

Usage

Refer to the `encoding` while reading from files.

To read a `.csv` file with `shift_jis` encoding:

```
FROM read_csv('my_shift_jis.csv', encoding = 'shift_jis');
```

To read a `.csv` file with `windows-1251-2000` encoding:

```
FROM read_csv('my_windows_1251_2000.csv', encoding = 'windows-1251-2000');
```

To read a `.csv` file with `windows-1252-2000` encoding:

```
FROM read_csv('my_windows_1252_2000.csv', encoding = 'windows-1252-2000');
```

To read a `.csv` file with EUC_CN encoding:

```
FROM read_csv('my_euc_cn.csv', encoding = 'EUC_CN');
```

All Supported Encodings

The following table is an alphabetized list of all `encoding` values supported by DuckDB using the `encodings` core extension:

Id	Encoding
1	5601
2	8859_1
3	8859_10
4	8859_15
5	8859_2
6	8859_3
7	8859_4
8	8859_5
9	8859_6
10	8859_7

Id	Encoding
11	8859_8
12	8859_9
13	aix-IBM_udcJP-4.3.6
14	ANSI_X3.110
15	ascii
16	ASMO_449
17	BALTIC
18	big5
19	CNS11643.1986_1
20	CNS11643.1986_2
21	CNS-11643-1992
22	cp037
23	cp1026
24	CP1250
25	CP1251
26	CP1252
27	CP1253
28	CP1254
29	CP1255
30	CP1256
31	CP1257
32	CP1258
33	cp273
34	cp424
35	cp437
36	cp500
37	CP737
38	CP775
39	cp850
40	cp852
41	cp855
42	cp857
43	cp860
44	cp861
45	cp862
46	cp863
47	cp864
48	cp865
49	cp866

Id	Encoding
50	cp869
51	cp949
52	CSN_369103
53	CWI
54	DEC_MCS
55	EBCDIC_AT_DE
56	EBCDIC_AT_DE_A
57	EBCDIC_CA_FR
58	EBCDIC_DK_NO
59	EBCDIC_DK_NO_A
60	EBCDIC_ES
61	EBCDIC_ES_A
62	EBCDIC_ES_S
63	EBCDIC_FI_SE
64	EBCDIC_FI_SE_A
65	EBCDIC_FR
66	EBCDIC_IS_FRISS
67	EBCDIC_IT
68	EBCDIC_PT
69	EBCDIC_UK
70	EBCDIC_US
71	EUC_CN
72	EUC_JP
73	EUC_KR
74	EUC_TW
75	euc-jp-2007
76	eucTH
77	euc-tw-2014
78	gb18030
79	glibc-ANSI_X3.110-2.3.3
80	glibc-ARMSCII_8-2.3.3
81	glibc-BIG5-2.3.3
82	glibc-BIG5HKSCS-2.3.3
83	glibc-BS_4730-2.3.3
84	glibc-CP10007-2.3.3
85	glibc-CP1125-2.3.3
86	glibc-CP932-2.3.3
87	glibc-CSA_Z243.4_1985_1-2.3.3
88	glibc-CSA_Z243.4_1985_2-2.3.3

Id	Encoding
89	glibc-DIN_66003-2.3.3
90	glibc-DS_2089-2.3.3
91	glibc-ECMA_CYRILLIC-2.3.3
92	glibc-ES-2.3.3
93	glibc-ES2-2.3.3
94	glibc-EUC_CN-2.3.3
95	glibc-EUC_JP_MS-2.3.3
96	glibc-EUC_JP-2.3.3
97	glibc-EUC_KR-2.3.3
98	glibc-GB_1988_80-2.3.3
99	glibc-GBK-2.3.3
100	glibc-GEORGIAN_ACADEMY-2.3.3
101	glibc-GEORGIAN_PS-2.3.3
102	glibc-IBM1046-2.3.3
103	glibc-IBM1124-2.3.3
104	glibc-IBM1129-2.3.3
105	glibc-IBM1132-2.3.3
106	glibc-IBM1133-2.3.3
107	glibc-IBM1160-2.3.3
108	glibc-IBM1161-2.3.3
109	glibc-IBM1162-2.3.3
110	glibc-IBM1163-2.3.3
111	glibc-IBM1164-2.3.3
112	glibc-IBM856-2.3.3
113	glibc-IBM864-2.3.3
114	glibc-IBM866NAV-2.3.3
115	glibc-IBM870-2.3.3
116	glibc-IBM874-2.3.3
117	glibc-IBM922-2.3.3
118	glibc-IBM943-2.3.3
119	glibc-ISIRI_3342-2.3.3
120	glibc-ISO_5428-2.3.3
121	glibc-ISO_6937-2.3.3
122	glibc-ISO_8859_13-2.3.3
123	glibc-ISO_8859_16-2.3.3
124	glibc-ISO_8859_7-2.3.3
125	glibc-ISO_8859_8-2.3.3
126	glibc-ISO_IR_209-2.3.3
127	glibc-IT-2.3.3

Id	Encoding
128	glibc-JIS_C6220_1969_RO-2.3.3
129	glibc-JIS_C6229_1984_B-2.3.3
130	glibc-JOHAB-2.3.3
131	glibc-JUS_I.B1.002-2.3.3
132	glibc-KOI8_R-2.3.3
133	glibc-KOI8_T-2.3.3
134	glibc-KOI8_U-2.3.3
135	glibc-KSC5636-2.3.3
136	glibc-MAC_SAMI-2.3.3
137	glibc-MACINTOSH-2.3.3
138	glibc-MSZ_7795.3-2.3.3
139	glibc-NC_NC00_10-2.3.3
140	glibc-NF_Z_62_010_1973-2.3.3
141	glibc-NF_Z_62_010-2.3.3
142	glibc-NS_4551_1-2.3.3
143	glibc-NS_4551_2-2.3.3
144	glibc-PT154-2.3.3
145	glibc-PT-2.3.3
146	glibc-PT2-2.3.3
147	glibc-RK1048-2.3.3
148	glibc-SEN_850200_B-2.3.3
149	glibc-SEN_850200_C-2.3.3
150	glibc-SJIS-2.3.3
151	glibc-T.61_8BIT-2.3.3
152	glibc-UHC-2.3.3
153	glibc-VISCII-2.3.3
154	glibc-WIN_SAMI_2-2.3.3
155	GOST_19768_74
156	GREEK_CCITT
157	GREEK7
158	GREEK7_OLD
159	HP_ROMAN8
160	hpux-big5-11.11
161	hpux-cp1140-11.11
162	hpux-cp1141-11.11
163	hpux-cp1142-11.11
164	hpux-cp1143-11.11
165	hpux-cp1144-11.11
166	hpux-cp1145-11.11

Id	Encoding
167	hpux-cp1146-11.11
168	hpux-cp1147-11.11
169	hpux-cp1148-11.11
170	hpux-cp1149-11.11
171	hpux-cp1250-11.11
172	hpux-cp1251-11.11
173	hpux-cp1252-11.11
174	hpux-cp1253-11.11
175	hpux-cp1254-11.11
176	hpux-cp1255-11.11
177	hpux-cp1256-11.11
178	hpux-cp1257-11.11
179	hpux-cp1258-11.11
180	hpux-cp437-11.11
181	hpux-cp737-11.11
182	hpux-cp775-11.11
183	hpux-cp850-11.11
184	hpux-cp852-11.11
185	hpux-cp855-11.11
186	hpux-cp857-11.11
187	hpux-cp860-11.11
188	hpux-cp861-11.11
189	hpux-cp862-11.11
190	hpux-cp863-11.11
191	hpux-cp864-11.11
192	hpux-cp865-11.11
193	hpux-cp866-11.11
194	hpux-cp869-11.11
195	hpux-cp874-11.11
196	hpux-eucJP0201-11.11
197	hpux-eucJP-11.11
198	hpux-eucJPMS-11.11
199	hpux-eucKR-11.11
200	hpux-eucTW-11.11
201	hpux-greee-11.11
202	hpux-hkbig5-11.11
203	hpux-hp15CN-11.11
204	hpux-iso87-11.11
205	hpux-roc15-11.11

Id	Encoding
206	hpux-sjis0201-11.11
207	hpux-sjis-11.11
208	hpux-sjisMS-11.11
209	IBM_1046
210	IBM_1124
211	IBM_1129
212	IBM_1252
213	IBM_850
214	IBM_856
215	IBM_858
216	IBM_932
217	IBM_943
218	IBM_eucJP
219	IBM_eucKR
220	IBM_eucTW
221	IBM_udcJP_GR
222	IBM038
223	IBM1004
224	ibm-1004_P100-1995
225	ibm-1006_P100-1995
226	ibm-1006_X100-1995
227	ibm-1008_P100-1995
228	ibm-1008_X100-1995
229	ibm-1009_P100-1995
230	ibm-1010_P100-1995
231	ibm-1011_P100-1995
232	ibm-1012_P100-1995
233	ibm-1013_P100-1995
234	ibm-1014_P100-1995
235	ibm-1015_P100-1995
236	ibm-1016_P100-1995
237	ibm-1017_P100-1995
238	ibm-1018_P100-1995
239	ibm-1019_P100-1995
240	ibm-1020_P100-2003
241	ibm-1021_P100-2003
242	ibm-1023_P100-2003
243	ibm-1025_P100-1995
244	ibm-1026_P100-1995

Id	Encoding
245	ibm-1027_P100-1995
246	ibm-1040_P100-1995
247	ibm-1041_P100-1995
248	ibm-1042_P100-1995
249	ibm-1043_P100-1995
250	ibm-1046_X110-1999
251	IBM1047
252	ibm-1047_P100-1995
253	ibm-1051_P100-1999
254	ibm-1088_P100-1995
255	ibm-1089_P100-1995
256	ibm-1097_P100-1995
257	ibm-1097_X100-1995
258	ibm-1098_P100-1995
259	ibm-1098_X100-1995
260	ibm-1100_P100-2003
261	ibm-1101_P100-2003
262	ibm-1102_P100-2003
263	ibm-1103_P100-2003
264	ibm-1104_P100-2003
265	ibm-1105_P100-2003
266	ibm-1106_P100-2003
267	ibm-1107_P100-2003
268	ibm-1112_P100-1995
269	ibm-1114_P100-1995
270	ibm-1114_P100-2001
271	ibm-1115_P100-1995
272	ibm-1122_P100-1999
273	ibm-1123_P100-1995
274	ibm-1124_X100-1996
275	ibm-1125_P100-1997
276	ibm-1126_P100_P100-1997_U3
277	ibm-1126_P100-1997
278	ibm-1127_P100-2004
279	ibm-1129_P100-1997
280	ibm-1130_P100-1997
281	ibm-1131_P100-1997
282	ibm-1132_P100-1997
283	ibm-1132_P100-1998

Id	Encoding
284	ibm-1133_P100-1997
285	ibm-1137_P100-1999
286	ibm-1137_PMOD-1999
287	ibm-1140_P100-1997
288	ibm-1141_P100-1997
289	ibm-1142_P100-1997
290	ibm-1143_P100-1997
291	ibm-1144_P100-1997
292	ibm-1145_P100-1997
293	ibm-1146_P100-1997
294	ibm-1147_P100-1997
295	ibm-1148_P100-1997
296	ibm-1149_P100-1997
297	ibm-1153_P100-1999
298	ibm-1154_P100-1999
299	ibm-1155_P100-1999
300	ibm-1156_P100-1999
301	ibm-1157_P100-1999
302	ibm-1158_P100-1999
303	ibm-1159_P100-1999
304	ibm-1160_P100-1999
305	ibm-1161_P100-1999
306	ibm-1162_P100-1999
307	ibm-1163_P100-1999
308	ibm-1164_P100-1999
309	ibm-1165_P101-2000
310	ibm-1166_P100-2002
311	ibm-1167_P100-2002
312	ibm-1168_P100-2002
313	ibm-1174_X100-2007
314	ibm-1250_P100-1999
315	ibm-1251_P100-1995
316	ibm-1252_P100-2000
317	ibm-1253_P100-1995
318	ibm-1254_P100-1995
319	ibm-1255_P100-1995
320	ibm-1256_P110-1997
321	ibm-1257_P100-1995
322	ibm-1258_P100-1997

Id	Encoding
323	ibm-12712_P100-1998
324	ibm-1275_P100-1995
325	ibm-1275_X100-1995
326	ibm-1276_P100-1995
327	ibm-1277_P100-1995
328	ibm-1280_P100-1996
329	ibm-1281_P100-1996
330	ibm-1282_P100-1996
331	ibm-1283_P100-1996
332	ibm-1284_P100-1996
333	ibm-1285_P100-1996
334	ibm-13121_P100-1995
335	ibm-13124_P100-1995
336	ibm-13124_P10A-1995
337	ibm-13125_P100-1997
338	ibm-13140_P101-2000
339	ibm-13143_P101-2000
340	ibm-13145_P101-2000
341	ibm-13156_P101-2000
342	ibm-13157_P101-2000
343	ibm-13162_P101-2000
344	ibm-13218_P100-1996
345	ibm-1350_P110-1997
346	ibm-1351_P110-1997
347	ibm-1362_P100-1997
348	ibm-1362_P110-1999
349	ibm-1363_P100-1997
350	ibm-1363_P10A-1997
351	ibm-1363_P10B-1998
352	ibm-1363_P110-1999
353	ibm-1363_P11A-1999
354	ibm-1363_P11B-1999
355	ibm-1363_P11C-2006
356	ibm-1364_P100-2007
357	ibm-1364_P110-2007
358	ibm-13676_P102-2001
359	ibm-1370_P100-1999
360	ibm-1370_X100-1999
361	ibm-1371_P100-1999

Id	Encoding
362	ibm-1371_X100-1999
363	ibm-1373_P100-2002
364	ibm-1374_P100_P100-2005_MS
365	ibm-1374_P100-2005
366	ibm-1375_P100-2004
367	ibm-1375_P100-2006
368	ibm-1375_P100-2007
369	ibm-1375_P100-2008
370	ibm-1375_X100-2004
371	ibm-1377_P100_P100-2006_U3
372	ibm-1377_P100-2006
373	ibm-1377_P100-2008
374	ibm-1380_P100-1995
375	ibm-1380_X100-1995
376	ibm-1381_P110-1999
377	ibm-1381_X110-1999
378	ibm-1382_P100-1995
379	ibm-1382_X100-1995
380	ibm-1383_P110-1999
381	ibm-1383_X110-1999
382	ibm-1385_P100-1997
383	ibm-1385_P100-2005
384	ibm-1386_P100-2001
385	ibm-1386_P110-1997
386	ibm-1388_P100-2024
387	ibm-1388_P103-2001
388	ibm-1388_P110-2000
389	ibm-1390_P100-1999
390	ibm-1390_P110-2003
391	ibm-1399_P100-1999
392	ibm-1399_P110-2003
393	ibm-16684_P100-1999
394	ibm-16684_P110-2003
395	ibm-16804_X110-1999
396	ibm-17221_P100-2001
397	ibm-17240_P101-2000
398	ibm-17248_X110-1999
399	ibm-20780_P100-1999
400	ibm-21344_P101-2000

Id	Encoding
401	ibm-21427_P100-1999
402	ibm-21427_X100-1999
403	ibm-25546_P100-1997
404	IBM256
405	ibm-256_P100-1995
406	ibm-259_P100-1995
407	ibm-259_X100-1995
408	ibm-273_P100-1999
409	IBM274
410	ibm-274_P100-2000
411	IBM275
412	ibm-275_P100-1995
413	IBM277
414	ibm-277_P100-1999
415	IBM278
416	ibm-278_P100-1999
417	IBM280
418	ibm-280_P100-1999
419	IBM281
420	ibm-282_P100-1995
421	IBM284
422	ibm-284_P100-1999
423	IBM285
424	ibm-285_P100-1999
425	ibm-286_P100-2003
426	ibm-28709_P100-1995
427	IBM290
428	ibm-290_P100-1995
429	ibm-293_P100-1995
430	ibm-293_X100-1995
431	IBM297
432	ibm-297_P100-1999
433	ibm-300_P110-1997
434	ibm-300_P120-2006
435	ibm-300_X110-1997
436	ibm-301_P110-1997
437	ibm-301_X110-1997
438	ibm-33058_P100-2000
439	ibm-33722_P120-1999

Id	Encoding
440	ibm-33722_P12A_P12A-2004_U2
441	ibm-33722_P12A_P12A-2009_U2
442	ibm-33722_P12A-1999
443	ibm-367_P100-1995
444	ibm-37_P100-1999
445	IBM420
446	ibm-420_X110-1999
447	ibm-420_X120-1999
448	IBM423
449	ibm-423_P100-1995
450	ibm-424_P100-1995
451	ibm-425_P101-2000
452	ibm-437_P100-1995
453	ibm-4517_P100-2005
454	ibm-4899_P100-1998
455	ibm-4904_P101-2000
456	ibm-4909_P100-1999
457	ibm-4930_P100-1997
458	ibm-4930_P110-1999
459	ibm-4933_P100-1996
460	ibm-4933_P100-2002
461	ibm-4944_P101-2000
462	ibm-4945_P101-2000
463	ibm-4948_P100-1995
464	ibm-4951_P100-1995
465	ibm-4952_P100-1995
466	ibm-4954_P101-2000
467	ibm-4955_P101-2000
468	ibm-4956_P101-2000
469	ibm-4957_P101-2000
470	ibm-4958_P101-2000
471	ibm-4959_P101-2000
472	ibm-4960_P100-1995
473	ibm-4960_X100-1995
474	ibm-4961_P101-2000
475	ibm-4962_P101-2000
476	ibm-4963_P101-2000
477	ibm-4971_P100-1999
478	ibm-500_P100-1999

Id	Encoding
479	ibm-5012_P100-1999
480	ibm-5026_P120-1999
481	ibm-5026_X120-1999
482	ibm-5035_P120_P12A-2005_U2
483	ibm-5035_P120-1999
484	ibm-5035_X120-1999
485	ibm-5039_P110-1996
486	ibm-5039_P11A-1998
487	ibm-5048_P100-1995
488	ibm-5049_P100-1995
489	ibm-5050_P120-1999
490	ibm-5050_P12A-1999
491	ibm-5067_P100-1995
492	ibm-5104_X110-1999
493	ibm-5123_P100-1999
494	ibm-5142_P100-1995
495	ibm-5210_P100-1999
496	ibm-5233_P100-2011
497	ibm-5346_P100-1998
498	ibm-5347_P100-1998
499	ibm-5348_P100-1997
500	ibm-5349_P100-1998
501	ibm-5350_P100-1998
502	ibm-5351_P100-1998
503	ibm-5352_P100-1998
504	ibm-5353_P100-1998
505	ibm-5354_P100-1998
506	ibm-53685_P101-2000
507	ibm-54191_P100-2006
508	ibm-5470_P100_P100-2005_MS
509	ibm-5470_P100-2005
510	ibm-5471_P100-2006
511	ibm-5471_P100-2007
512	ibm-5473_P100-2006
513	ibm-5478_P100-1995
514	ibm-5486_P100-1999
515	ibm-5487_P100-2001
516	ibm-5488_P100-2001
517	ibm-5495_P100-1999

Id	Encoding
518	ibm-62383_P100-2007
519	ibm-720_P100-1997
520	ibm-737_P100-1997
521	ibm-775_P100-1996
522	ibm-803_P100-1999
523	ibm-806_P100-1998
524	ibm-808_P100-1999
525	ibm-813_P100-1995
526	ibm-819_P100-1999
527	ibm-833_P100-1995
528	ibm-834_P100-1995
529	ibm-834_X100-1995
530	ibm-835_P100-1995
531	ibm-835_X100-1995
532	ibm-836_P100-1995
533	ibm-837_P100-1995
534	ibm-837_P100-2011
535	ibm-837_X100-1995
536	ibm-838_P100-1995
537	ibm-848_P100-1999
538	ibm-8482_P100-1999
539	ibm-849_P100-1999
540	ibm-850_P100-1999
541	IBM851
542	ibm-851_P100-1995
543	ibm-852_P100-1999
544	ibm-855_P100-1995
545	ibm-856_P100-1995
546	ibm-857_P100-1995
547	ibm-858_P100-1997
548	ibm-859_P100-1999
549	ibm-860_P100-1995
550	ibm-861_P100-1995
551	ibm-8612_P100-1995
552	ibm-8612_X110-1995
553	ibm-862_P100-1995
554	ibm-863_P100-1995
555	ibm-864_X110-1999
556	ibm-864_X120-2012

Id	Encoding
557	ibm-865_P100-1995
558	ibm-866_P100-1995
559	ibm-867_P100-1998
560	IBM868
561	ibm-868_P100-1995
562	ibm-868_X100-1995
563	ibm-869_P100-1995
564	IBM870
565	ibm-870_P100-1999
566	IBM871
567	ibm-871_P100-1999
568	ibm-872_P100-1999
569	IBM874
570	ibm-874_P100-1995
571	IBM875
572	ibm-875_P100-1995
573	ibm-878_P100-1996
574	IBM880
575	ibm-880_P100-1995
576	IBM891
577	ibm-891_P100-1995
578	ibm-895_P100-1995
579	ibm-896_P100-1995
580	ibm-897_P100-1995
581	ibm-9005_X100-2005
582	ibm-9005_X110-2007
583	ibm-901_P100-1999
584	ibm-902_P100-1999
585	ibm-9027_P100-1999
586	ibm-9027_X100-1999
587	IBM903
588	ibm-903_P100-1995
589	ibm-9030_P100-1995
590	IBM904
591	ibm-904_P100-1995
592	ibm-9042_P101-2000
593	ibm-9044_P100-1999
594	ibm-9048_P100-1998
595	ibm-9049_P100-1999

Id	Encoding
596	IBM905
597	ibm-905_P100-1995
598	ibm-9056_P100-1995
599	ibm-9061_P100-1999
600	ibm-9064_P101-2000
601	ibm-9066_P100-1995
602	ibm-9067_X100-2005
603	ibm-912_P100-1999
604	ibm-913_P100-2000
605	ibm-914_P100-1995
606	ibm-9145_P110-1997
607	ibm-9145_X110-1997
608	ibm-915_P100-1995
609	ibm-916_P100-1995
610	IBM918
611	ibm-918_P100-1995
612	ibm-918_X100-1995
613	ibm-920_P100-1995
614	ibm-921_P100-1995
615	ibm-922_P100-1999
616	ibm-923_P100-1998
617	ibm-9238_X110-1999
618	ibm-924_P100-1998
619	ibm-926_P100-2000
620	ibm-927_P100-1995
621	ibm-927_X100-1995
622	ibm-928_P100-1995
623	ibm-930_P120_P12A-2006_U2
624	ibm-930_P120-1999
625	ibm-930_X120-1999
626	ibm-9306_P101-2000
627	ibm-931_P120-1999
628	ibm-931_X120-1999
629	ibm-932_P120-1999
630	ibm-932_P12A_P12A-2000_U2
631	ibm-932_P12A-1999
632	ibm-933_P110-1999
633	ibm-933_X110-1999
634	ibm-935_P110-1999

Id	Encoding
635	ibm-935_X110-1999
636	ibm-937_P110-1999
637	ibm-937_X110-1999
638	ibm-939_P120_P12A-2005_U2
639	ibm-939_P120-1999
640	ibm-939_X120-1999
641	ibm-941_P120-1996
642	ibm-941_P12A-1996
643	ibm-941_P130-2001
644	ibm-941_P13A-2001
645	ibm-941_X110-1996
646	ibm-941_X11A-1996
647	ibm-942_P120-1999
648	ibm-942_P12A_P12A-2000_U2
649	ibm-942_P12A-1999
650	ibm-943_P130-1999
651	ibm-943_P14A-1999
652	ibm-943_P15A-2003
653	ibm-944_P100-1995
654	ibm-944_X100-1995
655	ibm-9444_P100_P100-2005_MS
656	ibm-9444_P100-2001
657	ibm-9444_P100-2005
658	ibm-9447_P100-2002
659	ibm-9448_X100-2005
660	ibm-9449_P100-2002
661	ibm-946_P100-1995
662	ibm-947_P100-1995
663	ibm-947_X100-1995
664	ibm-948_P110-1999
665	ibm-948_X110-1999
666	ibm-949_P110-1999
667	ibm-949_P11A-1999
668	ibm-949_X110-1999
669	ibm-950_P110-1999
670	ibm-950_X110-1999
671	ibm-951_P100-1995
672	ibm-951_X100-1995
673	ibm-952_P110-1997

Id	Encoding
674	ibm-953_P100-2000
675	ibm-954_P101-2007
676	ibm-955_P110-1997
677	ibm-9577_P100-2001
678	ibm-9580_P110-1999
679	ibm-960_P100-2000
680	ibm-963_P100-1995
681	ibm-964_P110-1999
682	ibm-964_X110-1999
683	ibm-970_P110_P110-2006_U2
684	ibm-970_P110-1999
685	ibm-971_P100-1995
686	IEC_P27_1
687	INIS
688	INIS_8
689	INIS_CYRILLIC
690	ISO_10367_BOX
691	ISO_5427
692	ISO_5427_EXT
693	ISO_5428
694	ISO_8859_1
695	ISO_8859_10
696	ISO_8859_11
697	ISO_8859_13
698	ISO_8859_14
699	ISO_8859_15
700	ISO_8859_2
701	ISO_8859_3
702	ISO_8859_4
703	ISO_8859_5
704	ISO_8859_6
705	ISO_8859_7
706	ISO_8859_8
707	ISO_8859_9
708	ISO_IR_197
709	ISO646_US
710	iso81
711	iso815
712	iso82

Id	Encoding
713	iso85
714	iso86
715	iso87
716	iso88
717	ISO8859_1
718	iso-8859_10-1998
719	iso-8859_11-2001
720	iso-8859_1-1998
721	iso-8859_13-1998
722	iso-8859_14-1998
723	ISO8859_15
724	iso-8859_15-1999
725	iso-8859_16-2001
726	ISO8859_2
727	iso-8859_2-1999
728	ISO8859_3
729	iso-8859_3-1999
730	ISO8859_4
731	iso-8859_4-1998
732	ISO8859_5
733	iso-8859_5-1999
734	ISO8859_6
735	iso-8859_6-1999
736	ISO8859_7
737	iso-8859_7-1987
738	iso-8859_7-2003
739	ISO8859_8
740	iso-8859_8-1999
741	ISO8859_9
742	iso-8859_9-1999
743	iso89
744	java-ASCII-1.3_P
745	java-Big5-1.3_P
746	java-Cp037-1.3_P
747	java-Cp1006-1.3_P
748	java-Cp1025-1.3_P
749	java-Cp1026-1.3_P
750	java-Cp1097-1.3_P
751	java-Cp1098-1.3_P

Id	Encoding
752	java-Cp1112-1.3_P
753	java-Cp1122-1.3_P
754	java-Cp1123-1.3_P
755	java-Cp1124-1.3_P
756	java-Cp1250-1.3_P
757	java-Cp1251-1.3_P
758	java-Cp1252-1.3_P
759	java-Cp1253-1.3_P
760	java-Cp1254-1.3_P
761	java-Cp1255-1.3_P
762	java-Cp1256-1.3_P
763	java-Cp1257-1.3_P
764	java-Cp1258-1.3_P
765	java-Cp1381-1.3_P
766	java-Cp1383-1.3_P
767	java-Cp273-1.3_P
768	java-Cp277-1.3_P
769	java-Cp278-1.3_P
770	java-Cp280-1.3_P
771	java-Cp284-1.3_P
772	java-Cp285-1.3_P
773	java-Cp297-1.3_P
774	java-Cp33722-1.3_P
775	java-Cp420-1.3_P
776	java-Cp424-1.3_P
777	java-Cp437-1.3_P
778	java-Cp500-1.3_P
779	java-Cp737-1.3_P
780	java-Cp775-1.3_P
781	java-Cp838-1.3_P
782	java-Cp850-1.3_P
783	java-Cp852-1.3_P
784	java-Cp855-1.3_P
785	java-Cp856-1.3_P
786	java-Cp857-1.3_P
787	java-Cp860-1.3_P
788	java-Cp861-1.3_P
789	java-Cp862-1.3_P
790	java-Cp863-1.3_P

Id	Encoding
791	java-Cp864-1.3_P
792	java-Cp865-1.3_P
793	java-Cp866-1.3_P
794	java-Cp868-1.3_P
795	java-Cp869-1.3_P
796	java-Cp870-1.3_P
797	java-Cp871-1.3_P
798	java-Cp874-1.3_P
799	java-Cp875-1.3_P
800	java-Cp918-1.3_P
801	java-Cp921-1.3_P
802	java-Cp922-1.3_P
803	java-Cp930-1.3_P
804	java-Cp933-1.3_P
805	java-Cp935-1.3_P
806	java-Cp937-1.3_P
807	java-Cp939-1.3_P
808	java-Cp942-1.3_P
809	java-Cp942C-1.3_P
810	java-Cp943-1.2.2
811	java-Cp943C-1.3_P
812	java-Cp948-1.3_P
813	java-Cp949-1.3_P
814	java-Cp949C-1.3_P
815	java-Cp950-1.3_P
816	java-Cp964-1.3_P
817	java-Cp970-1.3_P
818	java-EUC_CN-1.3_P
819	java-EUC_JP-1.3_P
820	java-EUC_KR-1.3_P
821	java-EUC_TW-1.3_P
822	java-ISO2022JP-1.3_P
823	java-ISO2022KR-1.3_P
824	java-ISO8859_1-1.3_P
825	java-ISO8859_13-1.3_P
826	java-ISO8859_2-1.3_P
827	java-ISO8859_3-1.3_P
828	java-ISO8859_4-1.3_P
829	java-ISO8859_5-1.3_P

Id	Encoding
830	java-ISO8859_6-1.3_P
831	java-ISO8859_7-1.3_P
832	java-ISO8859_8-1.3_P
833	java-ISO8859_9-1.3_P
834	java-Johab-1.3_P
835	java-KOI8_R-1.3_P
836	java-MS874-1.3_P
837	java-MS932-1.3_P
838	java-MS949-1.3_P
839	java-SJIS-1.3_P
840	java-TIS620-1.3_P
841	JISX0201.1976_0
842	JISX0201.1976_GR
843	JISX0208.1983_0
844	JISX0208.1983_GR
845	KOI_8
846	KOI8_R
847	KOI8_U
848	KSC5601.1987_0
849	LATIN_GREEK
850	LATIN_GREEK_1
851	latin-1
852	MAC_IS
853	mac_roman
854	MAC_UK
855	macos-0_1-10.2
856	macos-0_2-10.2
857	macos-1024-10.2
858	macos-1040-10.2
859	macos-1049-10.2
860	macos-1057-10.2
861	macos-1059-10.2
862	macos-1280-10.2
863	macos-1281-10.2
864	macos-1282-10.2
865	macos-1283-10.2
866	macos-1284-10.2
867	macos-1285-10.2
868	macos-1286-10.2

Id	Encoding
869	macos-1287-10.2
870	macos-1288-10.2
871	macos-1536-10.2
872	macos-21-10.5
873	macos-2562-10.2
874	macos-2563-10.2
875	macos-2566-10.2
876	macos-2817-10.2
877	macos-29-10.2
878	macos-3074-10.2
879	macos-33-10.5
880	macos-34-10.2
881	macos-35-10.2
882	macos-36_1-10.2
883	macos-36_2-10.2
884	macos-37_2-10.2
885	macos-37_3-10.2
886	macos-37_4-10.2
887	macos-37_5-10.2
888	macos-38_1-10.2
889	macos-38_2-10.2
890	macos-513-10.2
891	macos-514-10.2
892	macos-515-10.2
893	macos-516-10.2
894	macos-517-10.2
895	macos-518-10.2
896	macos-519-10.2
897	macos-520-10.2
898	macos-521-10.2
899	macos-527-10.2
900	macos-6_2-10.4
901	macos-6-10.2
902	macos-7_1-10.2
903	macos-7_2-10.2
904	macos-7_3-10.2
905	NATS_DANO
906	NATS_SEFI
907	osd-EBCDIC-DF03-IRV

Id	Encoding
908	osd-EBCDIC-DF04-1
909	osd-EBCDIC-DF04-15
910	PCK
911	roma8
912	shift_jis
913	solaris-zh_HK.hkscs-5.9
914	solaris-zh_TW_big5-2.7
915	thai8
916	TIS_620
917	utf-16
918	utf-8
919	windows-10000-2000
920	windows-10001-2000
921	windows-10002-2000
922	windows-10003-2000
923	windows-10004-2000
924	windows-10005-2000
925	windows-10006-2000
926	windows-10007-2000
927	windows-10008-2000
928	windows-10010-2000
929	windows-10017-2000
930	windows-10021-2000
931	windows-10029-2000
932	windows-10079-2000
933	windows-10081-2000
934	windows-10082-2000
935	windows-1026-2000
936	windows-1047-2000
937	windows-1140-2000
938	windows-1141-2000
939	windows-1142-2000
940	windows-1143-2000
941	windows-1144-2000
942	windows-1145-2000
943	windows-1146-2000
944	windows-1147-2000
945	windows-1148-2000
946	windows-1149-2000

Id	Encoding
947	windows-1250-2000
948	windows-1251-2000
949	windows-1252-2000
950	windows-1253-2000
951	windows-1254-2000
952	windows-1255-2000
953	windows-1256-2000
954	windows-1257-2000
955	windows-1258_db-2013
956	windows-1258-2000
957	windows-1361-2000
958	windows-20000-2000
959	windows-20001-2000
960	windows-20002-2000
961	windows-20003-2000
962	windows-20004-2000
963	windows-20005-2000
964	windows-20105-2000
965	windows-20106-2000
966	windows-20107-2000
967	windows-20108-2000
968	windows-20127-2000
969	windows-20261-2000
970	windows-20269-2000
971	windows-20273-2000
972	windows-20277-2000
973	windows-20278-2000
974	windows-20280-2000
975	windows-20284-2000
976	windows-20285-2000
977	windows-20290-2000
978	windows-20297-2000
979	windows-20420-2000
980	windows-20423-2000
981	windows-20424-2000
982	windows-20833-2000
983	windows-20838-2000
984	windows-20866-2000
985	windows-20871-2000

Id	Encoding
986	windows-20880-2000
987	windows-20905-2000
988	windows-20924-2000
989	windows-20932-2000
990	windows-20936-2000
991	windows-20949-2000
992	windows-21025-2000
993	windows-21027-2000
994	windows-21866-2000
995	windows-28591-2000
996	windows-28592-2000
997	windows-28593-2000
998	windows-28594-2000
999	windows-28595-2000
1000	windows-28596-2000
1001	windows-28597-2000
1002	windows-28598-2000
1003	windows-28599-2000
1004	windows-28603-vista
1005	windows-28605-2000
1006	windows-37-2000
1007	windows-38598-2000
1008	windows-437-2000
1009	windows-500-2000
1010	windows-51932-2006
1011	windows-51936-2000
1012	windows-51949-2000
1013	windows-708-2000
1014	windows-720-2000
1015	windows-737-2000
1016	windows-775-2000
1017	windows-850-2000
1018	windows-852-2000
1019	windows-855-2000
1020	windows-857-2000
1021	windows-858-2000
1022	windows-860-2000
1023	windows-861-2000
1024	windows-862-2000

Id	Encoding
1025	windows-863-2000
1026	windows-864-2000
1027	windows-865-2000
1028	windows-866-2000
1029	windows-869-2000
1030	windows-870-2000
1031	windows-874-2000
1032	windows-875-2000
1033	windows-932-2000
1034	windows-936-2000
1035	windows-949-2000
1036	windows-950_hkscs-2001
1037	windows-950-2000
1038	zh_CN.euc
1039	zh_CN.gbk
1040	zh_CN_cp935
1041	zh_TW_cp937
1042	zh_TW_euc

Excel Extension

The `excel` extension provides functions to format numbers per Excel's formatting rules by wrapping the i18npool library and to read/write Excel (`.xlsx`) files. However, please note that `.xls` files are not supported.

Installing and Loading

The `excel` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD excel;
```

Excel Scalar Functions

Function	Description
`excel_text(number, format_string)`	Format the given `number` per the rules given in the `format_string`
`text(number, format_string)`	Alias for `excel_text`

Examples

```
SELECT excel_text(1_234_567.897, 'h:mm AM/PM') AS timestamp;
```

timestamp
9:31 PM

```
SELECT excel_text(1_234_567.897, 'h AM/PM') AS timestamp;
```

timestamp
9 PM

Reading XLSX Files

Reading a `.xlsx` file is as simple as just SELECTing from it immediately, e.g.:

```
SELECT *
FROM 'test.xlsx';
```

a	b
1.0	2.0
3.0	4.0

However, if you want to set additional options to control the import process, you can use the `read_xlsx` function instead. The following named parameters are supported.

Option	Type	Default	Description
`header`	BOOLEAN	*Automatically inferred*	Whether to treat the first row as containing the names of the resulting columns.
`sheet`	VARCHAR	*Automatically inferred*	The name of the sheet in the xlsx file to read. Default is the first sheet.
`all_varchar`	BOOLEAN	`false`	Whether to read all cells as containing VARCHARs.
`ignore_errors`	BOOLEAN	`false`	Whether to ignore errors and silently replace cells that can't be cast to the corresponding inferred column type with `NULL`'s.
`range`	VARCHAR	*Automatically inferred*	The range of cells to read, in spreadsheet notation. For example, `A1:B2` reads the cells from A1 to B2. If not specified the resulting range will be inferred as rectangular region of cells between the first row of consecutive non-empty cells and the first empty row spanning the same columns.
`stop_at_empty`	BOOLEAN	*Automatically inferred*	Whether to stop reading the file when an empty row is encountered. If an explicit `range` option is provided, this is `false` by default, otherwise `true`.
`empty_as_varchar`	BOOLEAN	`false`	Whether to treat empty cells as VARCHAR instead of DOUBLE when trying to automatically infer column types.

```
SELECT *
FROM read_xlsx('test.xlsx', header = true);
```

a	b
1.0	2.0
3.0	4.0

Alternatively, the COPY statement with the `XLSX` format option can be used to import an Excel file into an existing table, in which case the types of the columns in the target table will be used to coerce the types of the cells in the Excel file.

```
CREATE TABLE test (a DOUBLE, b DOUBLE);
COPY test FROM 'test.xlsx' WITH (FORMAT xlsx, HEADER);
SELECT * FROM test;
```

Type and Range Inference

Because Excel itself only really stores numbers or strings in cells, and does not enforce that all cells in a column are of the same type, the `excel` extension has to do some guesswork to ”infer” and decide the types of the columns when importing an Excel sheet. While almost all columns are inferred as either DOUBLE or VARCHAR, there are some caveats:

- `TIMESTAMP`, `TIME`, `DATE` and `BOOLEAN` types are inferred when possible based on the *format* applied to the cell.
- Text cells containing `TRUE` and `FALSE` are inferred as `BOOLEAN`.
- Empty cells are considered to be `DOUBLE` by default, unless the `empty_as_varchar` option is set to `true`, in which case they are typed as `VARCHAR`.

If the `all_varchar` option is set to `true`, none of the above applies and all cells are read as `VARCHAR`.

When no types are specified explicitly, (e.g., when using the `read_xlsx` function instead of `COPY TO ... FROM '<file>.xlsx'`{:.language-sql .highlight}) the types of the resulting columns are inferred based on the first "data" row in the sheet, that is:

- If no explicit range is given
  - The first row after the header if a header is found or forced by the `header` option
  - The first non-empty row in the sheet if no header is found or forced
- If an explicit range is given
  - The second row of the range if a header is found in the first row or forced by the `header` option
  - The first row of the range if no header is found or forced

This can sometimes lead to issues if the first "data row" is not representative of the rest of the sheet (e.g., it contains empty cells) in which case the `ignore_errors` or `empty_as_varchar` options can be used to work around this.

However, when the `COPY TO ... FROM '<file>.xlsx'`{:.language-sql .highlight} syntax is used, no type inference is done and the types of the resulting columns are determined by the types of the columns in the table being copied to. All cells will simply be converted by casting from `DOUBLE` or `VARCHAR` to the target column type.

Writing XLSX Files

Writing `.xlsx` files is supported using the `COPY` statement with `XLSX` given as the format. The following additional parameters are supported.

Option	Type	Default	Description
`header`	`BOOLEAN`	`false`	Whether to write the column names as the first row in the sheet
`sheet`	`VARCHAR`	`Sheet1`	The name of the sheet in the xlsx file to write.
`sheet_row_limit`	`INTEGER`	`1048576`	The maximum number of rows in a sheet. An error is thrown if this limit is exceeded.

> **Warning.** Many tools only support a maximum of 1,048,576 rows in a sheet, so increasing the `sheet_row_limit` may render the resulting file unreadable by other software.

These are passed as options to the `COPY` statement after the `FORMAT`, e.g.:

```
CREATE TABLE test AS
    SELECT *
    FROM (VALUES (1, 2), (3, 4)) AS t(a, b);
COPY test TO 'test.xlsx' WITH (FORMAT xlsx, HEADER true);
```

Type Conversions

Because XLSX files only really support storing numbers or strings – the equivalent of `VARCHAR` and `DOUBLE`, the following type conversions are applied when writing XLSX files.

- Numeric types are cast to `DOUBLE` when writing to an XLSX file.

- Temporal types (`TIMESTAMP`, `DATE`, `TIME`, etc.) are converted to excel "serial" numbers, that is the number of days since 1900-01-01 for dates and the fraction of a day for times. These are then styled with a "number format" so that they appear as dates or times when opened in Excel.
- `TIMESTAMP_TZ` and `TIME_TZ` are cast to UTC `TIMESTAMP` and `TIME` respectively, with the timezone information being lost.
- `BOOLEAN`s are converted to `1` and `0`, with a "number format" applied to make them appear as `TRUE` and `FALSE` in Excel.
- All other types are cast to `VARCHAR` and then written as text cells.

Full-Text Search Extension

Full-Text Search is an extension to DuckDB that allows for search through strings, similar to SQLite's FTS5 extension.

Installing and Loading

The `fts` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD fts;
```

Usage

The extension adds two PRAGMA statements to DuckDB: one to create, and one to drop an index. Additionally, a scalar macro `stem` is added, which is used internally by the extension.

PRAGMA create_fts_index

```
create_fts_index(input_table, input_id, *input_values, stemmer = 'porter',
                 stopwords = 'english', ignore = '(\\.|[^a-z])+',
                 strip_accents = 1, lower = 1, overwrite = 0, incremental = 0)
```

PRAGMA that creates a FTS index for the specified table.

Name	Type	Description
`input_table`	VARCHAR	Qualified name of specified table, e.g., `'table_name'` or `'main.table_name'`
`input_id`	VARCHAR	Column name of document identifier, e.g., `'document_identifier'`
`input_values...`	VARCHAR	Column names of the text fields to be indexed (vararg), e.g., `'text_field_1'`, `'text_field_2'`, ..., `'text_field_N'`, or `'\*'` for all columns in input_table of type VARCHAR
`stemmer`	VARCHAR	The type of stemmer to be used. One of `'arabic'`, `'basque'`, `'catalan'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hindi'`, `'hungarian'`, `'indonesian'`, `'irish'`, `'italian'`, `'lithuanian'`, `'nepali'`, `'norwegian'`, `'porter'`, `'portuguese'`, `'romanian'`, `'russian'`, `'serbian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`, or `'none'` if no stemming is to be used. Defaults to `'porter'`
`stopwords`	VARCHAR	Qualified name of table containing a single VARCHAR column containing the desired stopwords, or `'none'` if no stopwords are to be used. Defaults to `'english'` for a pre-defined list of 571 English stopwords
`ignore`	VARCHAR	Regular expression of patterns to be ignored. Defaults to `'(\\.|[^a-z])+'`, ignoring all escaped and non-alphabetic lowercase characters
`strip_accents`	BOOLEAN	Whether to remove accents (e.g., convert á to a). Defaults to 1
`lower`	BOOLEAN	Whether to convert all text to lowercase. Defaults to 1

Name	Type	Description
`overwrite`	`BOOLEAN`	Whether to overwrite an existing index on a table. Defaults to `0`
`incremental`	`BOOLEAN`	Whether to maintain the FTS index with triggers after inserts and deletes on the input table. Requires the document identifier column to be declared `NOT NULL` or `PRIMARY KEY`. Defaults to `0`

This PRAGMA builds the index under a newly created schema. The schema will be named after the input table: if an index is created on table `'main.table_name'`, then the schema will be named `'fts_main_table_name'`.

`PRAGMA drop_fts_index`

```
drop_fts_index(input_table)
```

Drops a FTS index for the specified table.

Name	Type	Description
`input_table`	`VARCHAR`	Qualified name of input table, e.g., `'table_name'` or `'main.table_name'`

`match_bm25` Function

```
match_bm25(input_id, query_string, fields := NULL, k := 1.2, b := 0.75, conjunctive := 0)
```

When an index is built, this retrieval macro is created that can be used to search the index.

Name	Type	Description
`input_id`	`VARCHAR`	Column name of document identifier, e.g., `'document_identifier'`
`query_string`	`VARCHAR`	The string to search the index for
`fields`	`VARCHAR`	Comma-separated list of fields to search in, e.g., `'text_field_2, text_field_N'`. Defaults to `NULL` to search all indexed fields
k	`DOUBLE`	Parameter *k1* in the Okapi BM25 retrieval model. Defaults to `1.2`
b	`DOUBLE`	Parameter *b* in the Okapi BM25 retrieval model. Defaults to `0.75`
`conjunctive`	`BOOLEAN`	Whether to make the query conjunctive i.e., all terms in the query string must be present in order for a document to be retrieved

`stem` Function

```
stem(input_string, stemmer)
```

Reduces words to their base. Used internally by the extension.

Name	Type	Description
`input_string`	`VARCHAR`	The column or constant to be stemmed.

Name	Type	Description
`stemmer`	VARCHAR	The type of stemmer to be used. One of `'arabic'`, `'basque'`, `'catalan'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hindi'`, `'hungarian'`, `'indonesian'`, `'irish'`, `'italian'`, `'lithuanian'`, `'nepali'`, `'norwegian'`, `'porter'`, `'portuguese'`, `'romanian'`, `'russian'`, `'serbian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`, or `'none'` if no stemming is to be used.

Example Usage

Create a table and fill it with text data:

```
CREATE TABLE documents (
    document_identifier VARCHAR,
    text_content VARCHAR,
    author VARCHAR,
    doc_version INTEGER
);
INSERT INTO documents
    VALUES ('doc1',
            'The mallard is a dabbling duck that breeds throughout the temperate.',
            'Hannes Mühleisen',
            3),
           ('doc2',
            'The cat is a domestic species of small carnivorous mammal.',
            'Laurens Kuiper',
            2
           );
```

Build the index, and make both the `text_content` and `author` columns searchable.

```
PRAGMA create_fts_index(
    'documents', 'document_identifier', 'text_content', 'author'
);
```

Search the `author` field index for documents that are authored by `Muhleisen`. This retrieves `doc1`:

```
SELECT document_identifier, text_content, score
FROM (
    SELECT *, fts_main_documents.match_bm25(
        document_identifier,
        'Muhleisen',
        fields := 'author'
    ) AS score
    FROM documents
) sq
WHERE score IS NOT NULL
  AND doc_version > 2
ORDER BY score DESC;
```

document_identifier	text_content	score
doc1	The mallard is a dabbling duck that breeds throughout the temperate.	0.0

Search for documents about `small cats`. This retrieves `doc2`:

```
SELECT document_identifier, text_content, score
FROM (
```

```
    SELECT *, fts_main_documents.match_bm25(
        document_identifier,
        'small cats'
    ) AS score
    FROM documents
) sq
WHERE score IS NOT NULL
ORDER BY score DESC;
```

document_identifier	text_content	score
doc2	The cat is a domestic species of small carnivorous mammal.	0.0

> **Warning.** By default, the FTS index will not update automatically when the input table changes. To keep it up to date, create the index with `incremental = 1` (this requires the document identifier column to be `NOT NULL`), or recreate the index to refresh it.

httpfs (HTTP and S3)

httpfs Extension for HTTP and S3 Support

The `httpfs` extension is an autoloadable extension implementing a file system that allows reading remote/writing remote files. For plain HTTP(S), only file reading is supported. For object storage using the S3 API, the `httpfs` extension supports reading/writing/globbing files.

Installation and Loading

The `httpfs` extension will be, by default, autoloaded on first use of any functionality exposed by this extension.

To manually install and load the `httpfs` extension, run:

```
INSTALL AND LOAD httpfs;
```

HTTP(S)

The `httpfs` extension supports connecting to HTTP(S) endpoints.

S3 API

The `httpfs` extension supports connecting to S3 API endpoints.

HTTP(S) Support

With the `httpfs` extension, it is possible to directly query files over the HTTP(S) protocol. This works for all files supported by DuckDB or its various extensions, and provides read-only access.

```
SELECT *
FROM 'https://domain.tld/file.extension';
```

Partial Reading

For CSV files, files will be downloaded entirely in most cases, due to the row-based nature of the format. For Parquet files, DuckDB supports partial reading, i.e., it can use a combination of the Parquet metadata and HTTP range requests to only download the parts of the file that are actually required by the query. For example, the following query will only read the Parquet metadata and the data for the `column_a` column:

```
SELECT column_a
FROM 'https://domain.tld/file.parquet';
```

In some cases, no actual data needs to be read at all as they only require reading the metadata:

```
SELECT count(*)
FROM 'https://domain.tld/file.parquet';
```

DuckDB's own database format supports partial reading just as well as Parquet. Because the format is block-based and stores per-column and per-row-group statistics, DuckDB uses HTTP range requests to fetch only the blocks required by a query when a remote database file is attached read-only:

```
ATTACH 'https://domain.tld/file.duckdb' AS db (READ_ONLY);
SELECT column_a
FROM db.tbl;
```

As with Parquet, queries that only need the metadata, such as `count(*)`, do not read any table data.

Scanning Multiple Files

Scanning multiple files over HTTP(S) is also supported:

```
SELECT *
FROM read_parquet([
    'https://domain.tld/file1.parquet',
    'https://domain.tld/file2.parquet'
]);
```

Authenticating

To authenticate for an HTTP(S) endpoint, create an HTTP secret using the Secrets Manager:

```
CREATE SECRET http_auth (
    TYPE http,
    BEARER_TOKEN '<token>'
);
```

Or:

```
CREATE SECRET http_auth (
    TYPE http,
    EXTRA_HTTP_HEADERS MAP {
        'Authorization': 'Bearer <token>'
    }
);
```

HTTP Proxy

DuckDB supports HTTP proxies.

You can add an HTTP proxy using the Secrets Manager:

```
CREATE SECRET http_proxy (
    TYPE http,
    HTTP_PROXY '<http_proxy_url>',
    HTTP_PROXY_USERNAME '<username>',
    HTTP_PROXY_PASSWORD '<password>'
);
```

You can also set the scope for an HTTP proxy using the SCOPE keyword.

```
CREATE SECRET http_proxy (
    TYPE HTTP,
    SCOPE ['<https://duckdb.org>', '<https://some-other-website.org>'],
    HTTP_PROXY '<http_proxy_url>',
    HTTP_PROXY_USERNAME '<username>',
    HTTP_PROXY_PASSWORD '<password>'
);
```

Alternatively, you can add it via configuration options:

```
SET http_proxy = '<http_proxy_url>';
SET http_proxy_username = '<username>';
SET http_proxy_password = '<password>';
```

Note: You cannot set a proxy scope using the configurations options.

Using a Custom Certificate File

To use the `httpfs` extension with a custom certificate file, set the following configuration options prior to loading the extension:

```
LOAD httpfs;
SET ca_cert_file = '<certificate_file>';
SET enable_server_cert_verification = true;
```

If you would like to disable SSL verification for all HTTP requests using an HTTP secret you can do so with the following statement:

```
CREATE SECRET disable_ssl (
    TYPE HTTP,
    VERIFY_SSL 0
);
```

To enable it again for one specific endpoint, you can take advantage of the scope parameter:

```
CREATE SECRET enable_ssl_for_your_website (
    TYPE HTTP,
    SCOPE 'https://<your-website.com>',
    VERIFY_SSL 1
);
```

Hugging Face Support

The `httpfs` extension introduces support for the `hf://` protocol to access datasets hosted in Hugging Face repositories. See the announcement blog post for details.

Usage

Hugging Face repositories can be queried using the following URL pattern:

```
hf://datasets/<my_username>/<my_dataset>/<path_to_file>
```

For example, to read a CSV file, you can use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Where:

- `datasets-examples` is the name of the user/organization
- `doc-formats-csv-1` is the name of the dataset repository
- `data.csv` is the file path in the repository

The result of the query is:

kind	sound
dog	woof
cat	meow
pokemon	pika

kind	sound
human	hello

To read a JSONL file, you can run:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-jsonl-1/data.jsonl';
```

Finally, for reading a Parquet file, use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-parquet-1/data/train-00000-of-00001.parquet';
```

Each of these commands reads the data from the specified file format and displays it in a structured tabular format. Choose the appropriate command based on the file format you are working with.

Creating a Local Table

To avoid accessing the remote endpoint for every query, you can save the data in a DuckDB table by running a `CREATE TABLE ... AS` command. For example:

```
CREATE TABLE data AS
    SELECT *
    FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Then, simply query the `data` table as follows:

```
SELECT *
FROM data;
```

Multiple Files

To query all files under a specific directory, you can use a glob pattern. For example:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

count
173

By using glob patterns, you can efficiently handle large datasets and perform comprehensive queries across multiple files, simplifying your data inspections and processing tasks. Here, you can see how you can look for questions that contain the word “planet” in astronomy:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%';
```

count
21

Versioning and Revisions

In Hugging Face repositories, dataset versions or revisions are different dataset updates. Each version is a snapshot at a specific time, allowing you to track changes and improvements. In git terms, it can be understood as a branch or specific commit.

You can query different dataset versions/revisions by using the following URL:

```
hf://datasets/<my_username>/<my_dataset>@<my_branch>/<path_to_file>
```

For example:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1@~parquet/**/*.parquet';
```

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

The previous query will read all Parquet files under the `~parquet` revision. This is a special branch where Hugging Face automatically generates the Parquet files of every dataset to enable efficient scanning.

Authentication

Configure your Hugging Face Token in the DuckDB Secrets Manager to access private or gated datasets. First, visit Hugging Face Settings – Tokens to obtain your access token. Second, set it in your DuckDB session using DuckDB's Secrets Manager. DuckDB supports two providers for managing secrets:

`CONFIG`

The user must pass all configuration information into the `CREATE SECRET` statement. To create a secret using the `CONFIG` provider, use the following command:

```
CREATE SECRET hf_token (
    TYPE huggingface,
    TOKEN 'your_hf_token'
);
```

`credential_chain`

Automatically tries to fetch credentials. For the Hugging Face token, it will try to get it from `~/.cache/huggingface/token`. To create a secret using the `credential_chain` provider, use the following command:

```
CREATE SECRET hf_token (
    TYPE huggingface,
    PROVIDER credential_chain
);
```

S3 API Support

The `httpfs` extension supports reading/writing/globbing files on object storage servers using the S3 API. S3 offers a standard API to read and write to remote files (while regular http servers, predating S3, do not offer a common write API). DuckDB conforms to the S3 API, that is now common among industry storage providers.

Platforms

The `httpfs` filesystem is tested with AWS S3, Minio, Google Cloud and lakeFS. Other services that implement the S3 API (such as Cloudflare R2, SeaweedFS, and Tigris) should also work, but not all features may be supported.

The following table shows which parts of the S3 API are required for each `httpfs` feature.

Feature	Required S3 API features
Public file reads	HTTP Range requests
Private file reads	Secret key or session token authentication
File glob	ListObjectsV2
File writes	Multipart upload

Configuration and Authentication

> **Tip.** For an overview of the available methods and guidance on which one to use, see the S3 and AWS Authentication page.

The preferred way to configure and authenticate to S3 endpoints is to use secrets. Multiple secret providers are available.

To migrate from the deprecated S3 API, use a defined secret with a profile. See Selecting a Profile.

`config` Provider

The default provider, `config` (i.e., user-configured), allows access to the S3 bucket by manually providing a key. For example:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER config,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your-region>.amazonaws.com'`{:.language-sql .highlight}.

Now, to query using the above secret, simply query any `s3://` prefixed file:

```
SELECT *
FROM 's3://<your-bucket>/<your_file>.parquet';
```

`credential_chain` Provider

The `credential_chain` provider automatically fetches credentials using the AWS SDK (profiles, SSO, assumed roles, web identities, instance metadata, and more). It is provided by the aws extension. For example, to use the AWS SDK default provider:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain
);
```

For the full set of `credential_chain` options — `CHAIN` values, profile selection, assuming roles, SSO, web identity (IRSA), region resolution, validation, and auto-refresh — see the AWS extension page.

Overview of S3 Secret Parameters

Below is a complete list of the supported parameters that can be used for both the `config` and `credential_chain` providers:

Name	Description	Secret	Type	Default
`ENDPOINT`	Specify a custom S3 endpoint	S3, GCS, R2	`STRING`	`s3.amazonaws.com` for S3,
`KEY_ID`	The ID of the key to use	S3, GCS, R2	`STRING`	-
`REGION`	The region for which to authenticate (should match the region of the bucket to query)	S3, GCS, R2	`STRING`	`us-east-1`
`SECRET`	The secret of the key to use	S3, GCS, R2	`STRING`	-
`SESSION_TOKEN`	Optionally, a session token can be passed to use temporary credentials	S3, GCS, R2	`STRING`	-
`URL_COMPATIBILITY_MODE`	Can help when URLs contain problematic characters	S3, GCS, R2	`BOOLEAN`	`false`
`URL_STYLE`	Either `vhost` (alias `virtual`) or `path`	S3, GCS, R2	`STRING`	`vhost` for S3, `path` for R2 and GCS
`USE_SSL`	Whether to use HTTPS or HTTP	S3, GCS, R2	`BOOLEAN`	`true`
`VERIFY_SSL`	Whether to verify the SSL certificate of the server	S3, GCS, R2	`BOOLEAN`	`true`
`ACCOUNT_ID`	The R2 account ID to use for generating the endpoint URL	R2	`STRING`	-
`KMS_KEY_ID`	AWS KMS (Key Management Service) key for Server Side Encryption S3	S3	`STRING`	-
`REQUESTER_PAYS`	Allows use of "requester pays" S3 buckets	S3	`BOOLEAN`	`false`
`REFRESH`	Set to `auto` to periodically refresh credentials (see the aws extension)	S3, GCS, R2	`STRING`	-

Automatic Credential Refresh

Independently of the `REFRESH` secret parameter, DuckDB automatically refreshes S3 credentials when a request fails with an HTTP `401` or `403` status (for example, when temporary credentials have expired), then retries the request. This behavior is controlled by the `httpfs_enable_credential_refresh` setting (`BOOLEAN`, default `true`):

```
SET httpfs_enable_credential_refresh = false;
```

Platform-Specific Secret Types

S3 Secrets The httpfs extension supports Server Side Encryption via the AWS Key Management Service (KMS) on S3 using the `KMS_KEY_ID` option:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    REGION '<eu-west-1>',
    KMS_KEY_ID 'arn:aws:kms:<region>:<account_id>:<key>/<key_id>',
    SCOPE 's3://<bucket-sub-path>'
);
```

R2 Secrets While Cloudflare R2 uses the regular S3 API, DuckDB has a special Secret type, R2, to make configuring it a bit simpler:

```
CREATE OR REPLACE SECRET secret (
    TYPE r2,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    ACCOUNT_ID '<my_account_id>'
);
```

Note the addition of the ACCOUNT_ID which is used to generate the correct endpoint URL for you. Also note that R2 Secrets can also use both the CONFIG and `credential_chain` providers. However, since DuckDB uses an AWS client internally, when using `credential_chain`, the client will search for AWS credentials in the standard AWS credential locations (environment variables, credential files, etc.). Therefore, your R2 credentials must be made available as AWS environment variables (AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY) for the credential chain to work properly. Finally, R2 secrets are only available when using URLs starting with `r2://`, for example:

```
SELECT *
FROM read_parquet('r2://<some-file-that-uses-an-r2-secret>.parquet');
```

GCS Secrets While Google Cloud Storage is accessed by DuckDB using the S3 API, DuckDB has a special Secret type, GCS, to make configuring it a bit simpler:

```
CREATE OR REPLACE SECRET secret (
    TYPE gcs,
    KEY_ID '<my_hmac_access_id>',
    SECRET '<my_hmac_secret_key>'
);
```

Important: The KEY_ID and SECRET values must be HMAC keys generated specifically for Google Cloud Storage interoperability. These are not the same as regular GCP service account keys or access tokens. You can create HMAC keys by following the Google Cloud documentation for managing HMAC keys.

Note that the above secret will automatically have the correct Google Cloud Storage endpoint configured. Also note that GCS Secrets can also use both the CONFIG and `credential_chain` providers. However, since DuckDB uses an AWS client internally, when using `credential_chain`, the client will search for AWS credentials in the standard AWS credential locations (environment variables, credential files, etc.). Therefore, your GCS HMAC keys must be made available as AWS environment variables (AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY) for the credential chain to work properly. Finally, GCS secrets are only available when using URLs starting with `gcs://` or `gs://`, for example:

```
SELECT *
FROM read_parquet('gcs://<some/file/that/uses/a/gcs/secret>.parquet');
```

Reading

Reading files from S3 is now as simple as:

```
SELECT *
FROM 's3://<your-bucket>/<filename>.<extension>';
```

Partial Reading

The `httpfs` extension supports partial reading from S3 buckets.

Pinning Object Versions

By default, a long-running query re-reads an object at whatever version is current at read time, which can change if the object is overwritten. Set `s3_version_id_pinning` (BOOLEAN, default `false`) to pin reads to the object version captured on the first HEAD request, so a query sees a consistent version even if the object is overwritten mid-query. This requires the HTTP metadata cache:

```
SET enable_http_metadata_cache = true;
SET s3_version_id_pinning = true;
```

Reading Multiple Files

Multiple files are also possible, for example:

```
SELECT *
FROM read_parquet([
    's3://<your-bucket>/<filename-1>.parquet',
    's3://<your-bucket>/<filename-2>.parquet'
]);
```

Globbing

File globbing is implemented using the ListObjectsV2 API call and allows using filesystem-like glob patterns to match multiple files, for example:

```
SELECT *
FROM read_parquet('s3://<your-bucket>/*.parquet');
```

This query matches all files in the root of the bucket with the Parquet extension.

Several features for matching are supported, such as `*` to match any number of any character, `?` for any single character or `[0-9]` for a single character in a range of characters:

```
SELECT count(*) FROM read_parquet('s3://<your-bucket>/folder*/100?/t[0-9].parquet');
```

A useful feature when using globs is the `filename` option, which adds a column named `filename` that encodes the file that a particular row originated from:

```
SELECT *
FROM read_parquet('s3://<your-bucket>/*.parquet', filename = true);
```

This could for example result in:

column_a	column_b	filename
1	examplevalue1	s3://bucket-name/file1.parquet
2	examplevalue1	s3://bucket-name/file2.parquet

Hive Partitioning

DuckDB also offers support for the Hive partitioning scheme, which is available when using HTTP(S) and S3 endpoints.

Writing

Writing to S3 uses the multipart upload API. This allows DuckDB to robustly upload files at high speed. Writing to S3 works for both CSV and Parquet:

```
COPY table_name TO 's3://<your-bucket>/<filename>.<extension>';
```

Partitioned copy to S3 also works:

```
COPY table TO 's3://<your-bucket>/partitioned' (
    FORMAT parquet,
    PARTITION_BY (<part_col_a>, <part_col_b>)
);
```

An automatic check is performed for existing files/directories, which is currently quite conservative (and on S3 will add a bit of latency). To disable this check and force writing, an `OVERWRITE_OR_IGNORE` flag is added:

```
COPY table TO 's3://<your-bucket>/partitioned' (
    FORMAT parquet,
    PARTITION_BY (<part_col_a>, <part_col_b>),
    OVERWRITE_OR_IGNORE true
);
```

The naming scheme of the written files looks like this:

```
s3://<your-bucket>/partitioned/part_col_a=<val>/part_col_b=<val>/data_<thread_number>.parquet
```

Configuration

Some additional configuration options exist for the S3 upload, though the default values should suffice for most use cases.

Name	Description	Default
`s3_uploader_max_parts_per_file`	Used for part size calculation, see AWS docs	`10000`
`s3_uploader_max_filesize`	Used for part size calculation, see AWS docs	`80GB`

Further S3-related settings are available and documented in the configuration reference, including `enable_global_s3_configuration`, `merge_http_secret_into_s3_request`, `s3_allow_recursive_globbing`, `httpfs_enable_credential_refresh`, `s3_version_id_pinning`, and `unsafe_disable_etag_checks`.

S3 and AWS Authentication

This page is the entry point for authenticating to S3 and other AWS services from DuckDB. Authentication is configured using secrets, and the detailed reference material lives on the S3 API Support and AWS Extension pages. The sections below explain which method to use and link to the relevant reference.

Choosing an Authentication Method

There are two S3 secret providers. Use the `config` provider when you have static credentials, and the `credential_chain` provider when you want DuckDB to fetch credentials automatically through the AWS SDK.

Situation	Method	Reference
You have a static access key and secret	`config` provider	`config` provider
Credentials come from `~/.aws/credentials` or `~/.aws/config` profiles	`credential_chain` with a profile	Selecting a Profile
You sign in with AWS IAM Identity Center (SSO)	`credential_chain` with the `sso` chain	Single Sign-On (SSO)
You assume an IAM role via STS	`credential_chain` with the `sts` chain	Assuming a Role (STS)
You run on Amazon EKS with IAM Roles for Service Accounts (IRSA)	`credential_chain` with the `web_identity` chain	Web Identity (IRSA)
You run on an EC2 instance and want to use its instance profile	`credential_chain` with the `instance` chain	`credential_chain` provider

Situation	Method	Reference
You connect to Amazon RDS or Aurora with IAM authentication	`rds` secret	Amazon RDS (IAM Authentication)

The `config` provider is part of the `httpfs` extension and works with any S3-compatible storage. The `credential_chain` provider is AWS-specific and is provided by the aws extension, which is autoloaded on first use.

Static Credentials: the `config` Provider

Supply the key and secret directly:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER config,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

See the `config` provider documentation for the full details, including how to set an explicit `ENDPOINT` for non-AWS storage.

Automatic Credentials: the `credential_chain` Provider

The `credential_chain` provider fetches credentials automatically using the AWS SDK, covering profiles, SSO, assumed roles, web identities (IRSA), and instance metadata. To use the AWS SDK default provider:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain
);
```

You can select a specific chain with the `CHAIN` keyword and refine the behavior for each method. The individual mechanisms are documented on the AWS Extension page:

- Selecting a Profile
- Assuming a Role (STS)
- Web Identity (IRSA)
- Single Sign-On (SSO)

For behavior common to all `credential_chain` secrets, see Region Resolution, Validation, and Auto-Refresh.

Secret Parameters

The full list of S3 secret parameters that apply to both providers (`ENDPOINT`, `REGION`, `URL_STYLE`, `USE_SSL`, `KMS_KEY_ID`, `REQUESTER_PAYS`, and more) is documented in the Overview of S3 Secret Parameters. Platform-specific secret types for Cloudflare R2 and Google Cloud Storage are also available.

Legacy Authentication

Before DuckDB had a Secrets manager, credentials were loaded through the `load_aws_credentials` function. This method is deprecated. See the Legacy Authentication Scheme for S3 API page if you are maintaining older setups.

Legacy Authentication Scheme for S3 API

Prior to version 0.10.0, DuckDB did not have a Secrets manager. Hence, the configuration of and authentication to S3 endpoints was handled via variables. This page documents the legacy authentication scheme for the S3 API.

> **Warning.** This page describes a legacy method to store secrets as DuckDB settings. This increases the risk of accidentally leaking secrets (e.g., by printing their values). Therefore, avoid using these methods for storing secrets. The recommended way to configure and authenticate of S3 endpoints is to use secrets.

Legacy Authentication Scheme

To be able to read or write from S3, the correct region should be set:

```
SET s3_region = 'us-east-1';
```

Optionally, the endpoint can be configured in case a non-AWS object storage server is used:

```
SET s3_endpoint = '<domain>.<tld>:<port>';
```

If the endpoint is not SSL-enabled then run:

```
SET s3_use_ssl = false;
```

Switching between path-style and vhost-style URLs is possible using:

```
SET s3_url_style = 'path';
```

However, note that this may also require updating the endpoint. For example for AWS S3 it is required to change the endpoint to `s3.<region>.amazonaws.com`{:.language-sql .highlight}.

After configuring the correct endpoint and region, public files can be read. To also read private files, authentication credentials can be added:

```
SET s3_access_key_id = '<aws_access_key_id>';
SET s3_secret_access_key = '<aws_secret_access_key>';
```

Alternatively, temporary S3 credentials are also supported. They require setting an additional session token:

```
SET s3_session_token = '<aws_session_token>';
```

The aws extension allows for loading AWS credentials.

Per-Request Configuration

> **Deprecated.** Per-request configuration through S3 URL query parameters is deprecated and disabled by default in DuckDB 2.0: the `?` and the text following it are treated as part of the object key. To restore the legacy behavior, run `SET s3_enable_deprecated_url_parameters = true;` (URL compatibility mode must also be disabled). Once this setting has been explicitly set to `false`, it cannot be re-enabled in the same database instance. Instead of URL parameters, use scoped secrets to use different credentials for different paths.

Aside from the global S3 configuration described above, specific configuration values can be used on a per-request basis. This allows for use of multiple sets of credentials, regions, etc. These are used by including them on the S3 URI as query parameters. All the individual configuration values listed above can be set as query parameters. For instance:

```
SELECT *
FROM 's3://bucket/file.parquet?s3_access_key_id=accessKey&s3_secret_access_key=secretKey';
```

Multiple configurations per query are also allowed:

```
SELECT *
FROM 's3://bucket/file.parquet?s3_access_key_id=accessKey1&s3_secret_access_key=secretKey1' t1
INNER JOIN 's3://bucket/file.csv?s3_access_key_id=accessKey2&s3_secret_access_key=secretKey2' t2;
```

Configuration

Some additional configuration options exist for the S3 upload, though the default values should suffice for most use cases.

Additionally, most of the configuration options can be set via environment variables:

DuckDB setting	Environment variable	Note
`s3_region`	`AWS_REGION`	Takes priority over `AWS_DEFAULT_REGION`
`s3_region`	`AWS_DEFAULT_REGION`	
`s3_access_key_id`	`AWS_ACCESS_KEY_ID`	
`s3_secret_access_key`	`AWS_SECRET_ACCESS_KEY`	
`s3_session_token`	`AWS_SESSION_TOKEN`	
`s3_endpoint`	`DUCKDB_S3_ENDPOINT`	
`s3_use_ssl`	`DUCKDB_S3_USE_SSL`	
`s3_requester_pays`	`DUCKDB_S3_REQUESTER_PAYS`	

Iceberg

Iceberg Extension

The `iceberg` extension implements support for the Apache Iceberg open table format. There are two ways to work with Iceberg in DuckDB:

- **Individual tables** are read directly from storage, by pointing at a table's metadata. This requires no catalog and is read-only.
- **Catalog-managed tables** are accessed by attaching an Iceberg REST catalog. This unlocks the full feature set, including writing.

This page covers the basics of both. See also:

- Writing to Iceberg describes the supported write operations, partitioning and schema evolution.
- Iceberg Functions documents the functions provided by the extension.
- Iceberg Options documents the scan parameters, ATTACH options, secret options and settings.
- Catalogs explains how to attach an Iceberg REST Catalog, with instructions for specific catalogs.
- Troubleshooting lists common problems and their solutions.

Installing and Loading

The `iceberg` extension is installed and loaded automatically on first use. If you would like to install and load it manually, run:

```
INSTALL AND LOAD iceberg;
```

Updating the Extension

The `iceberg` extension often receives updates between DuckDB releases. To make sure that you have the latest version, update your extensions:

```
UPDATE EXTENSIONS;
```

Reading Iceberg Tables

Individual Iceberg tables can be read directly from storage with the `iceberg_scan` function, without attaching to a catalog. To test the examples, download the `iceberg_data.zip` file and unzip it.

Querying Individual Tables

Use the `iceberg_scan` function to read an Iceberg table from a path:

```
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

count_star()
51793

> The `allow_moved_paths` option ensures that some path resolution is performed, which allows scanning Iceberg tables that are moved.

You can also directly specify the current manifest in the query, this may be resolved from the catalog prior to the query, in this example the manifest version is a UUID. To do so, navigate to the `data/iceberg` directory and run:

```
SELECT count(*)
FROM iceberg_scan('lineitem_iceberg/metadata/v1.metadata.json');
```

count_star()
60175

For the full list of parameters accepted by `iceberg_scan`, see the Iceberg Options page.

Reading from Object Stores

The `iceberg` extension works together with the `httpfs` extension or the `azure` extension to access Iceberg tables in object stores such as S3 or Azure Blob Storage.

```
SELECT count(*)
FROM iceberg_scan('s3://bucketname/lineitem_iceberg/metadata/v1.metadata.json');
```

Metadata and Snapshots

To inspect the manifest entries and snapshots of a table, use the `iceberg_metadata` and `iceberg_snapshots` functions. To read tables without a `version-hint.text` file or with non-standard metadata file names, see Selecting Metadata Versions.

Time Travel

To read a historical snapshot of a table, pass either `snapshot_from_id` or `snapshot_from_timestamp` (the two are mutually exclusive). Use `iceberg_snapshots` to list the available snapshots:

```
-- Read a specific snapshot by id
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true, snapshot_from_id =
7635660646343998149);

-- Read the snapshot that was current at a given time
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true, snapshot_from_timestamp =
TIMESTAMP '2023-02-15 15:08:00');
```

Limitations of Direct Reads

- A version hint or explicit `version` is required to read a table, unless version guessing is enabled.
- Only `gzip`-compressed metadata is supported (via `metadata_compression_codec = 'gzip'`).

Catalog Managed Tables

For full read and write access — and to use Iceberg's catalog features — attach an Iceberg REST catalog. Most catalogs authenticate with OAuth2; store the credentials in a secret and attach the catalog with `ATTACH ... (TYPE iceberg, ...)`:

```
CREATE SECRET iceberg_secret (
    TYPE iceberg,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>',
    OAUTH2_SERVER_URI '<https://catalog.example.com/v1/oauth/tokens>'
);

ATTACH '<warehouse>' AS my_catalog (
    TYPE iceberg,
    SECRET iceberg_secret,
    URI '<https://catalog.example.com>'
);
```

Once attached, the catalog behaves like any other DuckDB database: reference its tables as `<catalog>.<schema>.<table>` and query or modify them with regular SQL.

```
SHOW ALL TABLES;

SELECT count(*) FROM my_catalog.default.events;

INSERT INTO my_catalog.default.events VALUES (1, 'click', now());
```

The metadata functions also work on catalog tables when passed a fully qualified name, e.g., `iceberg_snapshots(my_catalog.default.events)`.

- For the full set of write operations — partitioning, `UPDATE`, `DELETE`, `MERGE INTO`, `ALTER TABLE`, and table properties — see Writing to Iceberg.
- For catalog-specific setup, see Catalogs. For the complete list of `ATTACH` options and secret parameters, see Iceberg Options.

Time Travel

On an attached catalog, time travel with the `AT` clause directly on the table:

```
-- Using a snapshot id
SELECT * FROM my_catalog.default.events AT (VERSION => <snapshot_id>);

-- Using a timestamp
SELECT * FROM my_catalog.default.events AT (TIMESTAMP => TIMESTAMP '2025-09-22 12:32:43.217');
```

Interoperability with DuckLake

The `iceberg_to_ducklake` function performs a metadata-only copy of an attached Iceberg catalog into a DuckLake catalog, letting you query Iceberg tables as if they were DuckLake tables.

Writing to Iceberg

The DuckDB `iceberg` extension supports writing to Iceberg tables when an Iceberg catalog is attached. Writing to a table read directly from storage with `iceberg_scan` is not supported: the catalog is required to commit the new table metadata.

Supported Operations

The `iceberg` extension supports the following operations when used with a REST Catalog attached:

- `CREATE/DROP SCHEMA`
- `CREATE/DROP TABLE`
- `INSERT INTO`
- `UPDATE`

- DELETE
- MERGE INTO
- ALTER TABLE
- SELECT

Since these operations are supported, the following will also work:

```
COPY FROM DATABASE duckdb_db TO iceberg_datalake;
-- Or
COPY FROM DATABASE iceberg_datalake TO duckdb_db;
```

This functionality enables deep copies between Iceberg and DuckDB storage.

Creating Schemas and Tables

Iceberg namespaces are exposed as schemas, which can be created and dropped with `CREATE SCHEMA` and `DROP SCHEMA`:

```
CREATE SCHEMA iceberg_catalog.sales;
USE iceberg_catalog.sales;
```

Tables are created with the standard `CREATE TABLE` syntax. Iceberg table properties can be supplied with the `WITH` clause:

```
CREATE TABLE test_create_table (a INTEGER)
WITH (
    'format-version' = '2', -- format version will be elevated to format-version when creating a table
    'location' = 's3://path/to/data', -- location will be elevated to location when creating a table
    'property1' = 'value1',
    'property2' = 'value2'
);
```

Tables can also be created from the result of a query:

```
CREATE TABLE iceberg_catalog.sales.events_copy AS
    FROM iceberg_catalog.sales.events;
```

Table properties can also be inspected and modified after creation, see Table Properties Functions.

UPDATE and DELETE

The `UPDATE` and `DELETE` statements are supported on both partitioned and unpartitioned tables:

```
UPDATE iceberg_catalog.sales.events
SET event_name = 'view'
WHERE id = 1;

DELETE FROM iceberg_catalog.sales.events
WHERE event_time < TIMESTAMP '2026-01-01';
```

They use merge-on-read semantics and write positional deletes, see Limitations for `UPDATE` and `DELETE`.

MERGE INTO

The `MERGE INTO` statement is the recommended way to express upserts against Iceberg tables, which do not have a primary key. You can apply a change set in a single statement, deciding per row whether to insert, update, or delete:

```
MERGE INTO iceberg_catalog.default.people AS target
    USING (
        FROM (VALUES
            (1, 'John', 105_000.0),
            (3, 'Sarah', 95_000.0)
        ) t(id, name, salary)
```

```
    ) AS upserts
    ON (upserts.id = target.id)
    WHEN MATCHED THEN UPDATE
    WHEN NOT MATCHED THEN INSERT;
```

You can also use `WHEN MATCHED THEN DELETE` to express a delete set in the same statement. As with `UPDATE` and `DELETE`, `MERGE INTO` uses merge-on-read semantics and writes positional deletes to the Iceberg table.

`ALTER TABLE` and Schema Evolution

The `ALTER TABLE` statement is supported against Iceberg tables, covering the most common schema-evolution operations:

```
-- Rename the table
ALTER TABLE iceberg_catalog.default.simple_table
    RENAME TO renamed_table;

-- Add a column
ALTER TABLE iceberg_catalog.default.renamed_table
    ADD COLUMN col3 DOUBLE;

-- Rename a column
ALTER TABLE iceberg_catalog.default.renamed_table
    RENAME COLUMN col2 TO name;

-- Drop a column
ALTER TABLE iceberg_catalog.default.renamed_table
    DROP COLUMN col3;

-- Change the type of a column
ALTER TABLE iceberg_catalog.default.renamed_table
    ALTER COLUMN col1 TYPE BIGINT;

-- Set and drop the default value of a column
ALTER TABLE iceberg_catalog.default.renamed_table
    ALTER COLUMN col1 SET DEFAULT 0;
ALTER TABLE iceberg_catalog.default.renamed_table
    ALTER COLUMN col1 DROP DEFAULT;

-- Set the format-version
ALTER TABLE iceberg_catalog.default.renamed_table
    SET ('format-version' = 3);
```

Each `ALTER TABLE` statement updates the `current-schema-id` of the Iceberg table, and the changes become visible to other Iceberg-aware engines the next time they query the `LoadTableInformation` endpoint. Iceberg schema evolution is metadata-only, so no data files are rewritten.

Partitioning

Tables can be partitioned with the `PARTITIONED BY` clause using the Iceberg partition transforms:

Transform	Description
`<column>`{:.language-sql .highlight}	Identity: partition by the column value directly.
`year(<column>)`{:.language-sql .highlight}, `month(<column>)`{:.language-sql .highlight}, `day(<column>)`{:.language-sql .highlight}, `hour(<column>)`{:.language-sql .highlight}	Partition by a date or timestamp component.

Transform	Description
`bucket(<n>, <column>)`{:.language-sql .highlight}	Hash the column's value into n buckets. This is useful for stable partitioning on a high-cardinality column.
`truncate(<w>, <column>)`{:.language-sql .highlight}	Group rows by the first w characters (or by the column's value rounded down to a multiple of w for numeric columns). This is useful for prefix-based partitioning.

```
CREATE TABLE iceberg_catalog.default.events (
    event_id BIGINT,
    user_id BIGINT,
    country VARCHAR,
    event_time TIMESTAMP
)
PARTITIONED BY (day(event_time), bucket(16, user_id), truncate(2, country));
```

The partition spec of an existing table can be changed with `ALTER TABLE ... SET PARTITIONED BY`:

```
ALTER TABLE iceberg_catalog.default.events
    SET PARTITIONED BY (month(event_time));
```

The `bucket` and `truncate` transforms are also available as the `iceberg_bucket` and `iceberg_truncate` functions.

> The `write.target-file-size-bytes` and `write.parquet.row-group-size-bytes` table properties are honored for both partitioned and unpartitioned tables. Files are rotated at row-group boundaries, so if the row-group size exceeds the target file size, DuckDB caps the row-group size to the target file size.

Iceberg V3 Support

DuckDB-Iceberg supports the following Iceberg v3 specification features for both reads and writes:

- `VARIANT` and `TIMESTAMP_NS` data types
- Schema-level default values for columns
- Binary deletion vectors
- Row lineage tracking

The biggest change in practice is binary deletion vectors. In v2 tables, DuckDB-Iceberg writes positional deletes as Parquet files; in v3 tables, the same information is encoded as a much more compact binary deletion vector (Puffin file). DuckDB picks the right format automatically based on the table's `format-version`. You can create a v3 table by setting the `format-version` table property at creation time:

```
CREATE TABLE iceberg_catalog.default.v3_table
WITH ('format-version' = 3) AS
    FROM (VALUES
        (1, {'kind': 'click', 'x': 10}::VARIANT, TIMESTAMP_NS '2026-05-20 12:00:00.123456789'),
        (2, {'kind': 'view'}::VARIANT, TIMESTAMP_NS '2026-05-20 12:00:00.987654321')
    ) t(id, payload, event_time);
```

> **Warning.** The GEOGRAPHY type is not yet supported in DuckDB-Iceberg.

Limitations for UPDATE and DELETE

The `UPDATE` and `DELETE` operations have the following limitations:

- DuckDB-Iceberg only writes **positional deletes** (encoded as binary deletion vectors for v3 tables). Copy-on-write functionality is not yet supported.
- DuckDB-Iceberg only supports **merge-on-read semantics**. If a table has `write.update.mode` or `write.delete.mode` properties set to something other than `merge-on-read`, the operation fails.

Unsupported Operations

The following are not yet supported by the DuckDB `iceberg` extension:

- The `GEOGRAPHY` data type
- Copy-on-write semantics for `UPDATE`, `DELETE`, and `MERGE INTO`

Iceberg Functions

The `iceberg` extension provides the following functions:

Function	Description
`iceberg_scan(source)`	Reads an Iceberg table from a path.
`iceberg_metadata(source)`	Returns the manifest entries of an Iceberg table.
`iceberg_snapshots(source)`	Returns the snapshots of an Iceberg table.
`iceberg_column_stats(source)`	Returns per-data-file, per-column statistics of an Iceberg table.
`iceberg_partition_stats(source)`	Returns per-partition-field statistics of an Iceberg table.
`iceberg_load_table_response(table)`	Returns the raw `LoadTable` response of the REST Catalog.
`iceberg_files(source)`	Returns the data and delete files of an Iceberg table snapshot.
`iceberg_manifests(source)`	Returns the manifest files of an Iceberg table snapshot.
`iceberg_history(source)`	Returns the history of the table's current snapshot.
`iceberg_refs(source)`	Returns the branches and tags of an Iceberg table.
`iceberg_partition_specs(source)`	Returns the partition specs of an Iceberg table.
`iceberg_scan_plan(table)`	Returns the scan tasks (files to read) planned for an Iceberg table.
`iceberg_view_metadata(view)`	Returns the raw `LoadView` response of the REST Catalog.
`iceberg_table_properties(table)`	Returns all properties of the specified table.
`set_iceberg_table_properties(table, properties)`	Sets properties on the specified table.
`remove_iceberg_table_properties(table, property_names)`	Removes properties from the specified table.
`iceberg_schema_properties(schema)`	Returns all properties of the specified schema.
`set_iceberg_schema_properties(schema, properties)`	Sets properties on the specified schema.
`remove_iceberg_schema_properties(schema, property_list)`	Removes properties from the specified schema.
`iceberg_rollback_to_snapshot(table, snapshot_id)`	Rolls back an Iceberg table to an earlier snapshot.
`iceberg_rewrite_data_files(table)`	Compacts the data files of an Iceberg table.
`iceberg_to_ducklake(iceberg_catalog, ducklake_catalog)`	Copies the Iceberg metadata of a catalog to a DuckLake catalog.
`iceberg_bucket(num_buckets, value)`	Computes the Iceberg `bucket` partition transform of a value.
`iceberg_truncate(width, value)`	Computes the Iceberg `truncate` partition transform of a value.

The examples on this page use the `iceberg_data.zip` file; download and unzip it to run them.

`iceberg_scan`

The `iceberg_scan` function reads an Iceberg table from a path, without requiring an attached catalog:

```
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

count_star()
51793

The path can also point directly to a metadata file, or to an object store such as S3 or Azure Blob Storage. See the Iceberg Options page for the parameters accepted by this function.

`iceberg_metadata`

To access Iceberg metadata, you can use the `iceberg_metadata` function:

> **Warning.** In DuckDB 2.0.0, `iceberg_metadata` and `iceberg_column_stats` fail on the sample `lineitem_iceberg` table with an `Invalid Configuration Error` about `manifest_entry.sequence_number`.

```
SELECT *
FROM iceberg_metadata('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

manifest_path	manifest_sequence_number	manifest_content	status	content	file_path	file_format	record_count
lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b232e5ee23d3-m1.avro	2	DATA	ADDED	EXISTING	lineitem_iceberg/data/00041-414-f3c73457-bbd6-4b92-9c15-17b241171b16-00001.parquet	PARQUET	51793
lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b232e5ee23d3-m0.avro	2	DATA	DELETED	EXISTING	lineitem_iceberg/data/00000-411-0792dcfe-4e25-4ca3-8ada-175286069a47-00001.parquet	PARQUET	60175

`iceberg_snapshots`

To visualize the snapshots in an Iceberg table, use the `iceberg_snapshots` function:

```
SELECT *
FROM iceberg_snapshots('data/iceberg/lineitem_iceberg');
```

sequence_number	snapshot_id	timestamp_ms	manifest_list
1	3776207205136740581	2023-02-15 15:07:54.504	lineitem_iceberg/metadata/snap-3776207205136740581-1-cf3d0be5-cf70-453d-ad8f-48fdc412e608.avro
2	7635660646343998149	2023-02-15 15:08:14.73	lineitem_iceberg/metadata/snap-7635660646343998149-1-10eaca8a-1e1c-421e-ad6d-b232e5ee23d3.avro

> `iceberg_snapshots` does not take `allow_moved_paths`, `snapshot_from_id` or `snapshot_from_timestamp` as parameters.

`iceberg_column_stats`

The `iceberg_column_stats` function returns the statistics stored in the manifests for every data file and column. Its columns include `file_path`, `column_name`, `column_type`, `lower_bound`, `upper_bound`, `value_count`, `null_value_count` and `nan_value_count`.

```
SELECT *
FROM iceberg_column_stats('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

`iceberg_partition_stats`

The `iceberg_partition_stats` function returns statistics per partition field. Its columns include `partition_field_name`, `partition_source_columns`, `partition_field_transform`, `lower_bound` and `upper_bound`.

```
SELECT *
FROM iceberg_partition_stats(my_datalake.default.t);
```

`iceberg_load_table_response`

The `iceberg_load_table_response` function is an advanced function that only works with an attached Iceberg REST Catalog. It returns the raw `LoadTable` response of the catalog with the columns `metadata_location`, `metadata` (`VARIANT`), `config` (`MAP`), `storage_credentials` and `request_url`.

```
SELECT *
FROM iceberg_load_table_response(my_datalake.default.t);
```

Metadata Operations on Attached Catalogs

The functions `iceberg_metadata` and `iceberg_snapshots` are also available to use with an attached Iceberg catalog using a fully qualified path, e.g.:

```
SELECT * FROM iceberg_metadata(my_datalake.default.t)

-- Or
SELECT * FROM iceberg_snapshots(my_datalake.default.t)
```

This functionality enables the user to do **time traveling**.

```
-- Using a snapshot id
SELECT * FROM my_datalake.default.t AT (VERSION => <SNAPSHOT_ID>)

-- Or using a timestamp
SELECT * FROM my_datalake.default.t AT (TIMESTAMP => TIMESTAMP '2025-09-22 12:32:43.217')
```

Table Properties Functions

DuckDB provides functions to view and modify Iceberg table properties:

Function	Description
`iceberg_table_properties(table)`	Returns all properties of the specified table.

Function	Description
`set_iceberg_table_properties(table, properties)`	Sets properties on the specified table.
`remove_iceberg_table_properties(table, property_names)`	Removes properties from the specified table.

You can also use `ALTER TABLE ... SET` and `ALTER TABLE ... RESET` to modify table properties:

```
-- View table properties
SELECT *
FROM iceberg_table_properties(iceberg_catalog.default.my_table);

-- Set table properties
ALTER TABLE iceberg_catalog.default.my_table SET (
    'write.update.mode'='merge-on-read',
    'write.delete.mode'='merge-on-read',
    'format-version'=3
);

-- Remove table properties
ALTER TABLE iceberg_catalog.default.my_table RESET (
    'write.update.mode',
    'owner'
);
```

The `set_iceberg_table_properties` and `remove_iceberg_table_properties` functions take the table name as a string and cannot change reserved Iceberg table properties. In particular, `format-version` can only be changed with `ALTER TABLE ... SET`.

```
CALL set_iceberg_table_properties('iceberg_catalog.default.my_table', {
    'write.update.mode': 'merge-on-read'
});

CALL remove_iceberg_table_properties(
    'iceberg_catalog.default.my_table',
    ['write.update.mode']
);
```

Schema Properties Functions

Iceberg catalogs allow arbitrary key-value properties to be attached at the schema (namespace) level. These properties are typically used to record ownership, descriptions, default storage locations, or any other metadata that applies to every table in a schema.

Function	Description
`iceberg_schema_properties(schema)`	Returns all properties of the specified schema.
`set_iceberg_schema_properties(schema, properties)`	Sets properties on the specified schema.
`remove_iceberg_schema_properties(schema, property_list)`	Removes properties from the specified schema.

```
-- Set schema properties
CALL set_iceberg_schema_properties(iceberg_catalog.default, {
    'owner': 'analytics-team',
    'description': 'Default analytics schema'
});
```

```
-- Read schema properties
SELECT * FROM iceberg_schema_properties(iceberg_catalog.default);

-- Remove schema properties
CALL remove_iceberg_schema_properties(
    iceberg_catalog.default,
    ['description']
);
```

Schema properties are written through the Iceberg REST Catalog, so any other Iceberg-aware engine attached to the same catalog will see the updates immediately. `remove_iceberg_schema_properties` returns the number of remaining schema properties.

Metadata Table Functions

The following functions expose the Iceberg metadata tables. Like `iceberg_metadata` and `iceberg_snapshots`, they accept either a table path (along with the scan options `metadata_compression_codec`, `version` and `version_name_format`) or a table in an attached catalog.

Function	Additional parameters	Description
`iceberg_files(source)`	`allow_moved_paths`, `snapshot_from_id`, `snapshot_from_timestamp`	Returns one row per data file and delete file of the snapshot, including the file path, format, partition, record count, file size and column-level statistics (`column_sizes`, `value_counts`, `null_value_counts`, `lower_bounds`, `upper_bounds`, `readable_metrics`, etc.).
`iceberg_manifests(source)`	`allow_moved_paths`, `snapshot_from_id`, `snapshot_from_timestamp`	Returns one row per manifest file of the snapshot, with the manifest path and length, partition spec ID, the snapshot that added it, counts of added/existing/deleted data and delete files, and partition summaries.
`iceberg_history(source)`		Returns the snapshot log of the table with the columns `made_current_at`, `snapshot_id`, `parent_id` and `is_current_ancestor`.
`iceberg_refs(source)`		Returns the branches and tags of the table with the columns `name`, `type`, `snapshot_id`, `max_reference_age_in_ms`, `min_snapshots_to_keep` and `max_snapshot_age_in_ms`.
`iceberg_partition_specs(source)`		Returns one row per partition field of each partition spec with the columns `spec_id`, `is_default`, `field_id`, `name`, `transform`, `source_id` and `source_column`.

```
SELECT file_path, record_count, file_size_in_bytes
FROM iceberg_files(my_datalake.default.t);

SELECT *
FROM iceberg_partition_specs(my_datalake.default.t);
```

`iceberg_scan_plan`

The `iceberg_scan_plan(table)` function is an advanced function that returns the scan tasks DuckDB would read for a table, i.e., one row per data file together with its partition values and the delete files that apply to it. The `table` argument is the fully qualified name of a table in an attached Iceberg REST Catalog as a string (e.g., `'my_datalake.default.t'`), or a table location. It accepts the following named parameters:

Parameter	Type	Description
`row_filter`	`VARCHAR`	A filter expression used to prune the files to scan.
`produce_sequence_number`	`BOOLEAN`	Include the data sequence number of each file in the output.
`snapshot_from_id`	`UBIGINT`	Plan the scan for the snapshot with the given ID.
`snapshot_from_timestamp`	`TIMESTAMP_MS`	Plan the scan for the snapshot that was current at the given timestamp.
`allow_moved_paths`, `metadata_compression_codec`, `version`, `version_name_format`		The scan options, for tables given by their location.

If the `iceberg_use_server_side_scan_planning` setting is enabled and the catalog supports it, the planning is performed by the REST Catalog.

`iceberg_view_metadata`

The `iceberg_view_metadata(view)` function is the counterpart of `iceberg_load_table_response` for views: it returns the raw `LoadView` response of an attached Iceberg REST Catalog with the columns `metadata_location`, `metadata` (`VARIANT`), `config` (`MAP`) and `request_url`. This is useful for inspecting views, including views defined in a SQL dialect DuckDB cannot parse.

```
SELECT *
FROM iceberg_view_metadata('my_datalake.default.my_view');
```

Table Maintenance Functions

`iceberg_rollback_to_snapshot`

The `iceberg_rollback_to_snapshot(table, snapshot_id)` function sets the current snapshot of a table in an attached Iceberg catalog to an earlier snapshot. The table name is passed as a string and the snapshot ID can be taken from `iceberg_snapshots`. The function returns the `previous_snapshot_id` and the new `current_snapshot_id`.

```
CALL iceberg_rollback_to_snapshot('my_datalake.default.t', <snapshot_id>);
```

`iceberg_rewrite_data_files`

The `iceberg_rewrite_data_files(table)` function compacts the data files of a table in an attached Iceberg catalog: it rewrites small (and overly large) data files into files close to the target file size, applying any pending position and equality deletes. It returns the columns `rewritten_data_files_count`, `added_data_files_count`, `rewritten_bytes_count`, `failed_data_files_count` and `removed_delete_files_count`.

Parameter	Type	Default	Description
`target_file_size_bytes`	VARCHAR or integer	The table's `write.target-file-size-bytes` property, or 512 MiB	The target size of the rewritten files.
`min_file_size_bytes`	VARCHAR or integer	75% of the target file size	Files smaller than this are candidates for rewriting.
`max_file_size_bytes`	VARCHAR or integer	180% of the target file size	Files larger than this are candidates for rewriting.
`min_input_files`	BIGINT	5	The minimum number of candidate files (per partition) required to perform a rewrite.
`max_files_to_rewrite`	BIGINT	NULL (no limit)	The maximum number of files to rewrite.
`rewrite_all`	BOOLEAN	`false`	Rewrite all data files regardless of their size.
`output_spec_id`	INTEGER	The table's default partition spec	The partition spec to rewrite the files into.

```
CALL iceberg_rewrite_data_files('my_datalake.default.t', min_input_files => 2);
```

`iceberg_to_ducklake`

The DuckDB `iceberg` extension exposes a function to do metadata only copies of the Iceberg metadata to DuckLake, which enables users to query Iceberg tables as if they were DuckLake tables.

```
-- Given that we have an Iceberg catalog attached aliased to iceberg_datalake
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;

CALL iceberg_to_ducklake('iceberg_datalake', 'my_ducklake');
```

It is also possible to skip a set of tables provided the `skip_tables` parameter.

```
CALL iceberg_to_ducklake('iceberg_datalake', 'my_ducklake', skip_tables := ['table_to_skip']);
```

Partition Transform Functions

The `iceberg_bucket` and `iceberg_truncate` scalar functions implement the Iceberg partition transforms of the same name. They are most often used implicitly through the `PARTITIONED BY` clause (see Partitioning), but they can also be called directly:

Function	Description
`iceberg_bucket(num_buckets, value)`	Returns the Iceberg bucket partition value (an `INTEGER`) for `value`. Supported `value` types: `INTEGER`, `BIGINT`, `DECIMAL`, `DATE`, `TIME`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP_NS`, `VARCHAR`, `BLOB`, `UUID`.
`iceberg_truncate(width, value)`	Returns the Iceberg truncate partition value for `value`, with the same type as `value`. Supported `value` types: `INTEGER`, `BIGINT`, `DECIMAL`, `VARCHAR`, `BLOB`.

```
SELECT iceberg_bucket(16, 'duckdb');
SELECT iceberg_truncate(10, 1234);
```

Iceberg Options

This page lists the options of the `iceberg` extension: the parameters accepted by the Iceberg functions, the options of the `ATTACH` and `CREATE SECRET` statements used to connect to a catalog, and the global settings.

ATTACH Options

To make an Iceberg Catalog (not just a single table) known to the system, you'll need to use the `ATTACH` statement. The options provided to the `ATTACH` are divided into categories:

Parameter	Type	Default	Description
`URI`	`VARCHAR`	`NULL`	URL of the REST Catalog endpoint. In DuckDB 2.0, `URI` is the canonical option name; `ENDPOINT` is accepted as a deprecated alias (do not supply both).
`DEFAULT_SCHEMA`	`VARCHAR`	`NULL`	The default schema (namespace) to use for the attached catalog.
`ACCESS_DELEGATION_MODE`	`VARCHAR`	`'vended_credentials'`	Access delegation mode. Allowed values are `vended_credentials` and `none`.
`SUPPORT_NESTED_NAMESPACES`	`BOOLEAN`	`false`	Set to `true` for catalogs that support nested namespaces.
`STAGE_CREATE_TABLES`	`BOOLEAN`	`true`	Controls whether DuckDB uses staged `CREATE TABLE`. Disable for catalogs that do not support staged table creation.
`DISABLE_MULTI_TABLE_COMMIT`	`BOOLEAN`	`false`	Disables the multi-table transactions/commit endpoint. Enable for catalogs that reject this endpoint.
`SKIP_CREATE_TABLE_METADATA_UPDATES`	`BOOLEAN`	`false`	Skips follow-up metadata updates after non-staged `CREATE TABLE`. Enable for catalogs that fully initialize metadata during table creation and reject subsequent updates.
`REMOVE_FILES_ON_DELETE`	`BOOLEAN`	`true`	If disabled, files (data and metadata) created during the transaction will not be cleaned up on `ROLLBACK` or retries of the `COMMIT`.
`PURGE_REQUESTED`	`BOOLEAN`	`false`	Sends the PurgeRequested parameter when dropping a table.
`ENCODE_ENTIRE_PREFIX`	`BOOLEAN`	`false`	URL-encode the entire path prefix when communicating with the catalog.
`MAX_TABLE_STALENESS`	`INTERVAL`	`NULL`	Prevents unnecessary requests to the Iceberg REST Catalog. Accepts human-readable interval strings such as `10 minutes`, `30 seconds`, or `1 year`.
`EXTRA_HTTP_HEADERS`	`MAP(VARCHAR, VARCHAR)`	`NULL`	Extra headers (key-value) sent along with the requests to the REST Catalog.

Parameter	Type	Default	Description
DEFAULT_TABLE_LOCATION_FROM_NAMESPACE	BOOLEAN	false	If enabled, a CREATE TABLE without an explicit WITH ('location' = …) clause derives the table location as <namespace location>/<table name> from the namespace's location property. An explicit WITH ('location' = …) clause still takes precedence, and if the namespace has no location property the behavior is unchanged. Useful for catalogs such as AWS Glue that do not assign a table location server-side. Note: this feature was introduced in DuckDB 2.0.
TABLE_RESOLUTION	VARCHAR	lazy	Controls when table schemas are resolved. Accepted values: lazy, eager. With lazy (the default), a table's schema is resolved only when the table is first accessed, and unresolved tables are listed with a placeholder column of type UNKNOWN. With eager, schemas are resolved while tables are enumerated, which adds metadata-loading work and can increase catalog enumeration time for large catalogs. Eager resolution is needed when a Quack client attaches a server whose catalog still has unresolved tables.
PREFIX	VARCHAR	(empty)	Path prefix inserted into REST Catalog request URLs after /v1/, for example /v1/<prefix>/namespaces/…. Catalogs such as AWS Glue use it to encode a warehouse or account/catalog identifier. It is normally discovered from the catalog's configuration response; set it explicitly to override. A prefix containing / is split into path components unless ENCODE_ENTIRE_PREFIX is set.
NAMESPACE_SEPARATOR	VARCHAR	0x1F	Separator used to join the levels of a nested namespace into a single path component in REST requests. It is URL-decoded before use and cannot be empty. Defaults to the ASCII unit separator (0x1F).
USE_METADATA_LOCATION	BOOLEAN	false	When enabled, DuckDB loads table metadata from the file at the metadata-location returned by the catalog and treats it as authoritative, rather than using the metadata embedded in the catalog's response. Useful when the embedded metadata omits properties such as schema.name-mapping.default.

Some parameters enable others, see the list of associated additional parameters below this table.

Parameter	Type	Default	Accepted options	Description
ENDPOINT_TYPE	VARCHAR	NULL	S3_TABLES, GLUE	A common identifier of an Iceberg Catalog type, sets certain default parameters. Cannot be combined with URI (or its deprecated alias ENDPOINT) or AUTHORIZATION_TYPE.
AUTHORIZATION_TYPE	VARCHAR	OAUTH2	OAUTH2, SIGV4, NONE	The Authorization layer of the Iceberg Catalog to attach to. Use NONE for catalogs that need no authentication.

Endpoint Type

For commonly used Iceberg Catalog providers, the ENDPOINT_TYPE parameter can be used to set certain parameters to default values. Acting as a shorthand for connecting to these catalogs.

S3 Tables The parameters set by using the S3_TABLES ENDPOINT_TYPE are:

Parameter	Value
AUTHORIZATION_TYPE	SIGV4

Parameter	Value
SIGV4_REGION	The REGION section of the ARN provided as the ATTACH path.
URI	s3tables.<region>.amazonaws.com/iceberg{:.language-sql .highlight}
REMOVE_FILES_ON_DELETE	false (unless explicitly set)
STAGE_CREATE_TABLES	false (unless explicitly set)
PURGE_REQUESTED	true (unless explicitly set)

Glue When using the GLUE ENDPOINT_TYPE, the SECRET parameter of type VARCHAR has to be set to the path of an existing S3/AWS SECRET. The parameters set by using the GLUE ENDPOINT_TYPE are:

Parameter	Value
AUTHORIZATION_TYPE	SIGV4
URI	glue.<region>.amazonaws.com/iceberg{:.language-sql .highlight} using the region from the found SECRET
REMOVE_FILES_ON_DELETE	false (unless explicitly set)
STAGE_CREATE_TABLES	false (unless explicitly set)
PURGE_REQUESTED	true (unless explicitly set)

AWS Glue does not assign a table location server-side, so a CREATE TABLE without an explicit WITH ('location' = …) clause is rejected. Either provide an explicit location per table, or set DEFAULT_TABLE_LOCATION_FROM_NAMESPACE to true to derive the location from the namespace's location property.

Authorization

To configure the Authorization layer for connecting to an Iceberg REST Catalog, you'll need to pass the AUTHORIZATION_TYPE parameter. These are the supported options:

OAUTH2 Authorization Options The additional parameters that can be provided when the AUTHORIZATION_TYPE is set to OAUTH2 are the following.

Parameter	Type	Default	Description
SECRET	VARCHAR	NULL	The path to a SECRET of type ICEBERG to get the CLIENT_ID and CLIENT_SECRET from.
CLIENT_ID	VARCHAR	NULL	CLIENT_ID used in the OAuth2 authorization request.
CLIENT_SECRET	VARCHAR	NULL	CLIENT_SECRET used in the OAuth2 authorization request.

Parameter	Type	Default	Description
OAUTH2_SERVER_URI	VARCHAR	NULL	The endpoint of the OAuth2 server to contact.
OAUTH2_GRANT_TYPE	VARCHAR	NULL	The grant_type to use.
OAUTH2_SCOPE	VARCHAR	NULL	The scope to use.
DEFAULT_REGION	VARCHAR	NULL	The region to add to vended credentials if none is provided by the Catalog.
TOKEN	VARCHAR	NULL	The Bearer Token to use instead of making a request to the server. (disables refreshing)

SIGV4 Authorization Options The additional parameters that can be provided when the AUTHORIZATION_TYPE is set to SIGV4 are the following.

Parameter	Type	Default	Description
SECRET	VARCHAR	NULL	The S3 or AWS SECRET to use for signing.
SIGV4_SERVICE	VARCHAR	NULL	Override the SERVICE for signing requests, otherwise inferred from the URI parameter.
SIGV4_REGION	VARCHAR	NULL	Override the REGION for signing requests, otherwise inferred from the URI parameter.

ICEBERG SECRET Options Every option mentioned in the ATTACH Options section and its sub-sections can be used in the creation of an ICEBERG SECRET. When such an ICEBERG SECRET exists it can be inferred by subsequent ATTACH statements, or provided explicitly through the SECRET ATTACH option.

Settings

Setting	Type	Default	Description
unsafe_enable_version_guessing	BOOLEAN	false	Allows the extension to guess the latest metadata version when no version or hint file is given.
iceberg_default_format_version	UBIGINT	2	The Iceberg format version used when creating a new table without an explicit format-version property. Valid values are 2 and 3.
iceberg_unsafe_skip_puffin_verification	BOOLEAN	false	When reading V3 Deletion Vectors, skip the Puffin file verification (for compatibility with files written by older versions).
iceberg_use_metadata_log	BOOLEAN	true	Use a table's optional metadata-log to preserve atomicity guarantees, at the cost of an additional metadata GET request in rare cases.
iceberg_equality_delete_fast_filter	BOOLEAN	true	Apply equality deletes on byte-comparable types through a shared columnar hash filter instead of a bound expression per delete row. Unsupported types fall back to the expression-based path.

Setting	Type	Default	Description
`iceberg_metadata_log_clock_skew_ms`	`BIGINT`	`0`	Clock-skew allowance in milliseconds when `iceberg_use_metadata_log` is enabled. Table metadata written up to this long after the transaction start is considered visible. Set to `0` for strict timestamp comparisons.
`iceberg_use_server_side_scan_planning`	`BOOLEAN`	`false`	Use the REST Catalog's server-side scan planning endpoint (if the catalog provides it) instead of planning the scan in DuckDB.
`iceberg_logging_post_body_truncate_limit`	`UBIGINT`	`10000`	Maximum number of characters of a REST Catalog POST request body to include in Iceberg log messages. Longer bodies are truncated with a trailing `...(truncated)` marker. Set to `0` to omit the body.

Scan Options

The following parameters can be passed to `iceberg_scan`, `iceberg_column_stats`, `iceberg_metadata`, `iceberg_partition_stats`, `iceberg_snapshots`:

Parameter	Type	Default	Description
`allow_moved_paths`	`BOOLEAN`	`false`	Allows scanning Iceberg tables that are moved
`metadata_compression_codec`	`VARCHAR`	`''`	Treats metadata files as gzip-compressed when set to `'gzip'`
`snapshot_from_id`	`UBIGINT`	`NULL`	Access snapshot with a specific `id`
`snapshot_from_timestamp`	`TIMESTAMP`	`NULL`	Access snapshot with a specific `timestamp`
`version`	`VARCHAR`	`NULL` (use `version-hint.text`)	Provides an explicit version string, a version hint file, or `'?'` to guess the latest version (see Selecting Metadata Versions)
`version_name_format`	`VARCHAR`	`'v%s%s.metadata.json,%s%s.metadata.json'`	Controls how versions are converted to metadata file names

> `iceberg_snapshots` does not take `allow_moved_paths`, `snapshot_from_id` or `snapshot_from_timestamp` as parameters.

Selecting Metadata Versions

By default, the `iceberg` extension will look for a `version-hint.text` file to identify the proper metadata version to use. This can be overridden by explicitly supplying a version number via the `version` parameter to the functions of the `iceberg` extension:

```
SELECT *
FROM iceberg_snapshots(
    'data/iceberg/lineitem_iceberg',
    version = '1'
);
```

By default, `iceberg` functions will look for both `v{version}.metadata.json` and `{version}.metadata.json` files, or `v{version}.gz.metadata.json` and `{version}.gz.metadata.json` when `metadata_compression_codec = 'gzip'` is specified. Other compression codecs are not supported.

If any text file is provided through the `version` parameter, it is opened and treated as a version hint file:

```
SELECT *
FROM iceberg_snapshots(
    'data/iceberg/lineitem_iceberg',
    version = 'version-hint.txt'
);
```

The `iceberg` extension will open this file and use the **entire content** of the file as a provided version number. Note that the entire content of the `version-hint.txt` file will be treated as a literal version name, with no encoding, escaping or trimming. This includes any whitespace, or unsafe characters which will be explicitly passed formatted into filenames in the logic described below.

Working with Alternative Metadata Naming Conventions

The `iceberg` extension can handle different metadata naming conventions by specifying them as a comma-delimited list of format strings via the `version_name_format` parameter. Each format string must contain two `%s` parameters. The first is the location of the version number in the metadata filename and the second is the location of the filename extension specified by the `metadata_compression_codec`. The behavior described above is provided by the default value `v%s%s.metadata.json,%s%s.metadata.json`. If you had an alternatively named metadata file, e.g., `rev-2.metadata.json.gz`, the table can be read via the follow statement:

```
SELECT *
FROM iceberg_snapshots(
    'data/iceberg/alternative_metadata_gz_naming',
    version = '2',
    version_name_format = 'rev-%s.metadata.json%s',
    metadata_compression_codec = 'gzip'
);
```

"Guessing" Metadata Versions

By default, either a table version number or a `version-hint.text` **must** be provided for the `iceberg` extension to read a table. This is typically provided by an external data catalog. In the event neither is present, the `iceberg` extension can attempt to guess the latest version by passing `?` as the `version` parameter:

```
SELECT count(*)
FROM iceberg_scan(
    'data/iceberg/lineitem_iceberg_no_hint',
    version = '?',
    allow_moved_paths = true
);
```

The "latest" version is assumed to be the filename that is lexicographically largest when sorting the filenames. Collations are not considered. This behavior is not enabled by default as it may potentially violate ACID constraints. It can be enabled by setting `unsafe_enable_version_guessing` to `true`. When this is set, `iceberg` functions will attempt to guess the latest version by default before failing.

```
SET unsafe_enable_version_guessing = true;
SELECT count(*)
FROM iceberg_scan(
    'data/iceberg/lineitem_iceberg_no_hint',
    allow_moved_paths = true
);
```

Catalogs

The `iceberg` extension supports attaching Iceberg REST Catalogs. Attaching a catalog is required for writing to Iceberg. Before attaching an Iceberg REST Catalog, you must install the `iceberg` extension by following the instructions located in the overview.

The section below describes the generic way of attaching an Iceberg REST Catalog. For instructions specific to a catalog implementation, see Amazon S3 Tables, AWS Glue, Cloudflare R2 Data Catalog, Apache Polaris, Lakekeeper, SeaweedFS and Google Cloud BigLake.

Attaching an Iceberg REST Catalog

Most Iceberg REST Catalogs authenticate via OAuth2. You can use the existing DuckDB secret workflow to store login credentials for the OAuth2 service.

```
CREATE SECRET iceberg_secret (
    TYPE iceberg,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>',
    OAUTH2_SERVER_URI '<http://iceberg_rest_catalog_url.com/v1/oauth/tokens>'
);
```

If you already have a Bearer token, you can pass it directly to your `CREATE SECRET` statement

```
CREATE SECRET iceberg_secret (
    TYPE iceberg,
    TOKEN '<bearer_token>'
);
```

You can attach the Iceberg catalog with the following `ATTACH` statement.

```
LOAD httpfs;
ATTACH '<warehouse>' AS iceberg_catalog (
   TYPE iceberg,
   SECRET iceberg_secret, -- pass a specific secret name to prevent ambiguity
    URI '<https://rest_endpoint.com>'
);
```

To see the available tables run

```
SHOW ALL TABLES;
```

A REST Catalog with OAuth2 authorization can also be attached with just an `ATTACH` statement. For the complete list of `ATTACH` and `CREATE SECRET` options, see the Iceberg Options page.

Amazon S3 Tables

The `iceberg` extension supports reading Iceberg tables stored in Amazon S3 Tables.

You can let DuckDB detect your AWS credentials and configuration based on the default profile in your `~/.aws` directory by creating the following secret using the Secrets Manager:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Alternatively, you can set the values manually:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<key_id>',
    SECRET '<secret>',
    REGION '<region>'
);
```

For the full range of credential options (assumed roles, SSO, web identity, and more), see the aws extension.

Then, connect to the catalog using your S3 Tables ARN (available in the AWS Management Console) and the `ENDPOINT_TYPE s3_tables` option:

```
ATTACH '<s3_tables_arn>' AS my_s3_tables_catalog (
   TYPE iceberg,
   ENDPOINT_TYPE s3_tables
);
```

> **Warning.** `ENDPOINT_TYPE s3_tables` always builds an endpoint of the form `s3tables.<region>.amazonaws.com/iceberg`. This is incorrect for any region whose endpoint does not use the plain `amazonaws.com` suffix — most notably the AWS China regions (`cn-north-1`, `cn-northwest-1`), which use `amazonaws.com.cn`. For such regions, attach the catalog by passing an explicit `URI` (with the correct host for your region) together with `AUTHORIZATION_TYPE 'sigv4'`, instead of using `ENDPOINT_TYPE`.

To check whether the attachment worked, list all tables:

```
SHOW ALL TABLES;
```

You can query a table as follows:

```
SELECT count(*)
FROM my_s3_tables_catalog.<namespace_name>.<table_name>;
```

AWS Glue (Amazon SageMaker Lakehouse)

The `iceberg` extension supports reading Iceberg tables through the Amazon SageMaker Lakehouse (a.k.a. AWS Glue) catalog.

Create an S3 secret using the Secrets Manager:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN sts,
    ASSUME_ROLE_ARN 'arn:aws:iam::<account_id>:role/<role>',
    REGION 'us-east-2'
);
```

In this example we use an STS token, but other authentication methods are supported.

Then, connect to the catalog:

```
ATTACH '<account_id>' AS glue_catalog (
    TYPE iceberg,
    URI 'glue.<region>.amazonaws.com/iceberg',
    AUTHORIZATION_TYPE 'sigv4'
);
```

Or alternatively:

```
ATTACH '<account_id>' AS glue_catalog (
    TYPE iceberg,
    ENDPOINT_TYPE 'glue'
);
```

> **Warning.** As with Amazon S3 Tables, `ENDPOINT_TYPE glue` always builds an endpoint of the form `glue.<region>.amazonaws.com/iceberg`, which is incorrect for regions that do not use the plain `amazonaws.com` suffix (most notably the AWS China regions `cn-north-1` and `cn-northwest-1`, which use `amazonaws.com.cn`). For such regions, attach with an explicit `URI` (with the correct host) together with `AUTHORIZATION_TYPE 'sigv4'` instead of using `ENDPOINT_TYPE`.

The warehouse identifier (the first argument to `ATTACH`) accepts the following forms:

Warehouse	Meaning
`:`	The default catalog of the caller's account.
`<account_id>`	A 12-digit AWS account ID.
`<account_id>:<catalog>`	A named catalog in the given account.

Warehouse	Meaning
`<catalog>/<sub_catalog>`	A nested (federated) catalog.
`<account_id>:<catalog>/<sub_catalog>`	A nested catalog in the given account.

To check whether the attachment worked, list all tables:

```
SHOW ALL TABLES;
```

You can query a table as follows:

```
SELECT count(*)
FROM glue_catalog.<namespace_name>.<table_name>;
```

If you have an S3 Tables federated catalog, you can create a table using the standard `CREATE TABLE` syntax;

```
CREATE TABLE glue_catalog.<namespace_name>.<table_name> (a INTEGER, b VARCHAR);
```

If the catalog is not federated by S3 Tables, you may need to create pass a `location` table property. You can do so using the `WITH` clause.

```
CREATE TABLE glue_catalog.<namespace_name>.<table_name> (a INTEGER, b VARCHAR)
WITH (
    'location' = 's3://path/to/location'
);
```

You can learn more about the `WITH` clause at Creating Schemas and Tables.

Cloudflare R2 Data Catalog

To attach to an R2 Cloudflare managed catalog follow the attach steps below.

```
CREATE SECRET r2_secret (
    TYPE iceberg,
    TOKEN '<r2_token>'
);
```

You can create a token by following the create an API token steps in getting started. Then, attach the catalog with the following commands.

```
ATTACH '<warehouse>' AS my_r2_catalog (
    TYPE iceberg,
    URI '<catalog-uri>'
);
```

The variables for `warehouse` and `catalog-uri` are available under the settings of the R2 Object Storage Catalog (R2 Object Store, Catalog name, Settings).

Once you attached to the R2 Data Catalog, create a schema. You can set it as default with the `USE` command:

```
CREATE SCHEMA my_r2_catalog.my_schema;
USE my_r2_catalog.my_schema;
```

Apache Polaris

To attach to a Polaris catalog, use the following commands:

```
CREATE SECRET polaris_secret (
    TYPE iceberg,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>'
);
```

```
ATTACH 'quickstart_catalog' AS polaris_catalog (
    TYPE iceberg,
    URI '<polaris_rest_catalog_endpoint>',
    ACCESS_DELEGATION_MODE 'vended_credentials'
);
```

Lakekeeper

To attach to a Lakekeeper catalog the following commands will work.

```
CREATE SECRET lakekeeper_secret (
    TYPE iceberg,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>',
    OAUTH2_SCOPE '<scope>',
    OAUTH2_SERVER_URI '<lakekeeper_oauth_url>'
);
```

```
ATTACH '<warehouse>' AS lakekeeper_catalog (
    TYPE iceberg,
    URI '<lakekeeper_irc_url>',
    SECRET '<lakekeeper_secret>'
);
```

SeaweedFS

SeaweedFS table buckets provide both halves of an Iceberg deployment: the embedded Iceberg REST Catalog serves the table metadata, and the table bucket stores the table data as Parquet files behind the same SeaweedFS S3 gateway. Store the catalog's OAuth2 credentials in an Iceberg secret, and the S3 credentials in an S3 secret so DuckDB can read the Parquet files from the same gateway:

```
CREATE SECRET seaweedfs_secret (
    TYPE iceberg,
    CLIENT_ID '<access_key>',
    CLIENT_SECRET '<secret_key>',
    OAUTH2_SERVER_URI 'http://<seaweedfs_host>:8181/v1/oauth/tokens'
);
```

```
CREATE SECRET seaweedfs_storage (
    TYPE s3,
    KEY_ID '<access_key>',
    SECRET '<secret_key>',
    ENDPOINT '<seaweedfs_host>:8333',
    URL_STYLE 'path',
    USE_SSL false
);
```

```
ATTACH '<table_bucket_name>' AS seaweedfs_catalog (
    TYPE iceberg,
    URI 'http://<seaweedfs_host>:8181',
    SECRET seaweedfs_secret
);
```

Reads and writes work with the default ATTACH options, including CREATE SCHEMA, CREATE TABLE, INSERT, and DROP TABLE:

```
CREATE SCHEMA seaweedfs_catalog.sales;
CREATE TABLE seaweedfs_catalog.sales.orders (id BIGINT, region VARCHAR, amount DOUBLE);
INSERT INTO seaweedfs_catalog.sales.orders VALUES (1, 'NA', 12.5), (2, 'EU', 40.0), (3, 'APAC', 99.9);

SELECT region, sum(amount) AS total
FROM seaweedfs_catalog.sales.orders
GROUP BY region
ORDER BY total DESC;

DROP TABLE seaweedfs_catalog.sales.orders;
```

Google Cloud BigLake

To attach to a Google Cloud BigLake catalog, you can use extra HTTP headers to specify the GCP project for billing purposes.

First, get your Google Cloud access token:

```
gcloud auth application-default print-access-token
```

Then create a secret with the token and extra headers:

```
CREATE SECRET biglake_secret (
    TYPE iceberg,
    TOKEN '<your_access_token>',
    EXTRA_HTTP_HEADERS MAP {
        'x-goog-user-project': '<your_gcp_project_id>'
    }
);
```

Attach to the BigLake catalog:

```
ATTACH '<gs://your-biglake-bucket>' AS biglake_catalog (
    TYPE iceberg,
    URI 'https://biglake.googleapis.com/iceberg/v1/restcatalog',
    SECRET biglake_secret
);
```

Example using the BigLake public dataset:

```
CREATE SECRET biglake_public_secret (
    TYPE iceberg,
    TOKEN '<your_access_token>',
    EXTRA_HTTP_HEADERS MAP {
        'x-goog-user-project': '<your_gcp_project_id>'
    }
);

ATTACH 'gs://biglake-public-nyc-taxi-iceberg' AS biglake_public (
    TYPE iceberg,
    URI 'https://biglake.googleapis.com/iceberg/v1/restcatalog',
    SECRET biglake_public_secret
);

-- Query the data
SELECT count(*) FROM biglake_public.public_data.nyc_taxicab;
```

> Google Cloud access tokens expire after 1 hour. For long-running sessions, you'll need to refresh the token periodically.

Catalogs with Limited REST Spec Support

Some catalogs implement a subset of the Iceberg REST Catalog specification. Use the compatibility options below to adjust DuckDB's behavior for these catalogs.

Catalog behavior	Option to set
Does not support staged CREATE TABLE	`STAGE_CREATE_TABLES false`
Rejects the multi-table transactions/commit endpoint	`DISABLE_MULTI_TABLE_COMMIT true`
Fully initializes metadata on CREATE TABLE and rejects follow-up metadata updates	`SKIP_CREATE_TABLE_METADATA_UPDATES true`
Does not allow DuckDB to remove storage files on DROP TABLE	`REMOVE_FILES_ON_DELETE false`

For example, to attach a Unity Catalog Horizon endpoint that does not support staged creates, rejects the transactions/commit endpoint, and manages its own metadata and storage cleanup:

```
ATTACH '<warehouse>' AS horizon_catalog (
    TYPE iceberg,
    URI '<catalog_endpoint>',
    STAGE_CREATE_TABLES false,
    DISABLE_MULTI_TABLE_COMMIT true,
    SKIP_CREATE_TABLE_METADATA_UPDATES true,
    REMOVE_FILES_ON_DELETE false
);
```

Limitations

DuckDB supports Iceberg REST Catalogs backed by S3, S3 Tables, Google Cloud Storage (GCS) and Azure Data Lake Storage (ADLS).

Troubleshooting

Curl Request Fails

Problem

When trying to attach to an Iceberg REST Catalog endpoint, DuckDB returns the following error:

```
IO Error:
Curl Request to '/v1/oauth/tokens' failed with error: 'URL using bad/illegal format or missing URL'
```

Solution

Make sure that you have the latest Iceberg extension installed:

```
duckdb
```

```
FORCE INSTALL iceberg FROM core_nightly;
```

Exit DuckDB and start a new session:

```
duckdb
```

```
LOAD iceberg;
```

HTTP Error 403

Problem

When trying to list the tables in a remote-attached catalog, DuckDB returns the following error:

```
SHOW ALL TABLES;
```

```
Failed to query https://s3tables.us-east-2.amazonaws.com/iceberg/v1/arn:aws:s3tables:... http error 403
thrown.
Message: {"message":"The security token included in the request is invalid."}
```

Solution

Use the `duckdb_secrets()` function to check whether DuckDB loaded the required credentials:

```
.mode line
FROM duckdb_secrets();
```

If you do not see your credentials, set them manually using the following secret:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

ICU Extension

The `icu` extension provides collation, time zone, and calendar support.

> New In DuckDB 2.0, the dependency on the external ICU library was removed. The `icu` extension now implements collations, time zones, and calendars natively, with time zone data built directly from the IANA Time Zone Database. Behavior is unchanged, and the native implementation is smaller and faster. If you ship a custom build, note that ICU is no longer bundled.

Installing and Loading

The `icu` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD icu;
```

Features

The `icu` extension introduces the following features:

- Region-dependent collations
- Time zones, used for timestamp data types and timestamp functions

inet Extension

The `inet` extension defines the `INET` data type for storing IPv4 and IPv6 Internet addresses. It supports the CIDR notation for subnet masks (e.g., `198.51.100.0/22`, `2001:db8:3c4d::/48`).

Installing and Loading

The `inet` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD inet;
```

Examples

```
SELECT '127.0.0.1'::INET AS ipv4, '2001:db8:3c4d::/48'::INET AS ipv6;
```

ipv4	ipv6
127.0.0.1	2001:db8:3c4d::/48

```
CREATE TABLE tbl (id INTEGER, ip INET);
INSERT INTO tbl VALUES
    (1, '192.168.0.0/16'),
    (2, '127.0.0.1'),
    (3, '8.8.8.8'),
    (4, 'fe80::/10'),
    (5, '2001:db8:3c4d:15::1a2f:1a2b');
SELECT * FROM tbl;
```

id	ip
1	192.168.0.0/16
2	127.0.0.1
3	8.8.8.8
4	fe80::/10
5	2001:db8:3c4d:15::1a2f:1a2b

Operations on `INET` Values

`INET` values can be compared naturally, and IPv4 will sort before IPv6. Additionally, IP addresses can be modified by adding or subtracting integers.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('127.0.0.1'::INET + 10),
    ('fe80::10'::INET - 9),
    ('127.0.0.1'),
```

```
    ('2001:db8:3c4d:15::1a2f:1a2b');
SELECT cidr FROM tbl ORDER BY cidr ASC;
```

cidr
127.0.0.1
127.0.0.11
2001:db8:3c4d:15::1a2f:1a2b
fe80::7

`host` Function

The host component of an `INET` value can be extracted using the `HOST()` function.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.0.0/16'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, host(cidr) AS host FROM tbl;
```

cidr	host
192.168.0.0/16	192.168.0.0
127.0.0.1	127.0.0.1
2001:db8:3c4d:15::1a2f:1a2b/96	2001:db8:3c4d:15::1a2f:1a2b

`netmask` Function

Computes the network mask for the address's network.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.5/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, netmask(cidr) AS netmask FROM tbl;
```

cidr	netmask
192.168.1.5/24	255.255.255.0/24
127.0.0.1	255.255.255.255
2001:db8:3c4d:15::1a2f:1a2b/96	ffff:ffff:ffff:ffff:ffff:ffff::/96

`network` Function

Returns the network part of the address, zeroing out whatever is to the right of the netmask.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.5/24'),
```

```
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, network(cidr) AS network FROM tbl;
```

cidr	network
192.168.1.5/24	192.168.1.0/24
127.0.0.1	127.0.0.1
2001:db8:3c4d:15::1a2f:1a2b/96	2001:db8:3c4d:15::/96

`broadcast` Function

Computes the broadcast address for the address's network.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.5/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, broadcast(cidr) AS broadcast FROM tbl;
```

cidr	broadcast
192.168.1.5/24	192.168.1.255/24
127.0.0.1	127.0.0.1
2001:db8:3c4d:15::1a2f:1a2b/96	2001:db8:3c4d:15::ffff:ffff/96

`<<=` Predicate

Is subnet contained by or equal to subnet?

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.0/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, INET '192.168.1.5/32' <<= cidr AS subnet_contained FROM tbl;
```

cidr	subnet_contained
192.168.1.0/24	true
127.0.0.1	false
2001:db8:3c4d:15::1a2f:1a2b/96	false

`>>=` Predicate

Does subnet contain or equal subnet?

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.0/24'),
    ('127.0.0.1'),
```

```
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, INET '192.168.0.0/16' >>= cidr AS subnet_contains FROM tbl;
```

cidr	subnet_contains
192.168.1.0/24	true
127.0.0.1	false
2001:db8:3c4d:15::1a2f:1a2b/96	false

HTML Escape and Unescape Functions

```
SELECT html_escape('&');
```

```
┌──────────────────┐
│ html_escape('&') │
│     varchar      │
├──────────────────┤
│ &amp;            │
└──────────────────┘
```

```
SELECT html_unescape('&amp;');
```

```
┌──────────────────────────┐
│ html_unescape('&amp;') │
│         varchar          │
├──────────────────────────┤
│ &                        │
└──────────────────────────┘
```

Lance Extension

The `lance` extension adds support for reading and writing Lance tables. Lance is a modern lakehouse format optimized for ML/AI workloads, with native cloud storage support.

Installing and Loading

You can install the `lance` extension from DuckDB's core extensions repository and load it using the following commands:

```
INSTALL AND LOAD lance;
```

Usage

- Full SQL reference
- Cloud storage reference

Query a Lance Dataset

Local file:

```
SELECT *
FROM '<path/to/dataset.lance>'
LIMIT 10;
```

S3:

```
SELECT *
FROM 's3://<bucket/path/to/out.lance>'
LIMIT 10;
```

To access object store URIs (e.g., `s3://...`), configure a `TYPE lance` secret using the Secrets Manager:

```
CREATE SECRET (
    TYPE lance,
    PROVIDER credential_chain,
    SCOPE 's3://bucket/'
);

SELECT *
FROM 's3://<bucket/path/to/out.lance>'
LIMIT 10;
```

Write a Lance Dataset

Use the `COPY ... TO ...` statement to materialize query results as a Lance dataset.

```
-- Create/overwrite a Lance dataset from a query
COPY (
    SELECT 1::BIGINT AS id, 'a'::VARCHAR AS s
    UNION ALL
    SELECT 2::BIGINT AS id, 'b'::VARCHAR AS s
) TO '<path/to/dataset.lance>' (
```

```
    FORMAT lance,
    MODE 'overwrite'
);

-- Read it back via the replacement scan
SELECT count(*) FROM '<path/to/dataset.lance>';

-- Append more rows to an existing dataset
COPY (
    SELECT 3::BIGINT AS id, 'c'::VARCHAR AS s
) TO '<path/to/dataset.lance>' (
    FORMAT lance,
    MODE 'append'
);

-- Optionally create an empty dataset (schema only)
COPY (
    SELECT 1::BIGINT AS id, 'x'::VARCHAR AS s
    WITH NO DATA
) TO 'path/to/empty.lance' (
    FORMAT lance,
    MODE 'overwrite',
    WRITE_EMPTY_FILE true
);
```

To write to `s3://...` paths, configure a `TYPE lance` secret for that scope using the Secrets Manager:

```
CREATE SECRET (
    TYPE lance,
    PROVIDER credential_chain,
    SCOPE 's3://<bucket>/'
);

COPY (SELECT 1 AS id)
TO 's3://<bucket/path/to/out.lance>'
(FORMAT lance, MODE 'overwrite');
```

Create a Lance Dataset via CREATE TABLE (Directory Namespace)

When you ATTACH a directory as a Lance namespace, you can create new datasets using `CREATE TABLE` (schema-only) or `CREATE TABLE AS SELECT` (CTAS). The dataset is written to `<namespace_root>/<table_name.lance>`{:.language-sql .highlight}.

```
ATTACH '<path/to/dir>' AS lance_ns (TYPE lance);

-- Schema-only (creates an empty dataset)
CREATE TABLE lance_ns.main.my_empty (id BIGINT, s VARCHAR);

-- CTAS (writes query results)
CREATE TABLE lance_ns.main.my_dataset AS
    SELECT 1::BIGINT AS id, 'a'::VARCHAR AS s
    UNION ALL
    SELECT 2::BIGINT AS id, 'b'::VARCHAR AS s;

SELECT count(*) FROM lance_ns.main.my_dataset;
```

Vector Search

```
-- Search a vector column, returning distances in `_distance` (smaller is closer)
SELECT id, label, _distance
FROM lance_vector_search(
    '<path/to/dataset.lance>', 'vec',
    [0.1, 0.2, 0.3, 0.4]::FLOAT[4],
    k = 5,
```

```
    prefilter = true
)
ORDER BY _distance ASC;
```

See the SQL reference for full parameter documentation.

Full-Text Search (FTS)

```
-- Search a text column, returning BM25-like scores in `_score`
SELECT id, text, _score
FROM lance_fts(
    '<path/to/dataset.lance>',
    'text',
    'puppy',
    k = 10,
    prefilter = true
)
ORDER BY _score DESC;
```

See the SQL reference for full parameter documentation.

Hybrid Search (Vector + FTS)

```
-- Combine vector and text scores, returning `_hybrid_score` in addition to `_distance` / `_score`
SELECT id, _hybrid_score, _distance, _score
FROM lance_hybrid_search('<path/to/dataset.lance>',
                         'vec', [0.1, 0.2, 0.3, 0.4]::FLOAT[4],
                         'text', 'puppy',
                         k = 10, prefilter = false,
                         alpha = 0.5, oversample_factor = 4)
ORDER BY _hybrid_score DESC;
```

Limitations

The `lance` extension is currently available for the following platforms:

- `linux_amd64`
- `linux_arm64`
- `osx_arm64`
- `windows_amd64`

Legacy Parser Extension

> New The `legacy_parser` extension was introduced in DuckDB 2.0.

DuckDB 2.0.0 replaced the PostgreSQL-derived parser with a new PEG-based parser. The `legacy_parser` extension allows reverting to the old parser.

> The PEG parser is designed to be compatible with the previous parser, so queries that parsed before should continue to parse. If a query that parsed on an earlier version fails on v2.0.0, please file an issue.

Installing and Loading

To install and load the `legacy_parser` extension, run:

```
INSTALL AND LOAD legacy_parser;
```

Usage

To activate the legacy parser, run:

```
SET enable_legacy_parser = true;
```

To switch back to the PEG parser, run:

```
SET enable_legacy_parser = false;
```

MotherDuck Extension

The `motherduck` extension allows connecting to MotherDuck, a cloud data warehouse built on DuckDB.

Installing and Loading

The `motherduck` extension will be transparently autoinstalled and autoloaded on first use from the official extension repository. If you would like to install and load it manually, you can use the `motherduck` extension name or the `md` shorthand:

```
INSTALL AND LOAD md;
```

Usage

You can connect to MotherDuck by executing the following command:

```
ATTACH 'md:';
```

Platforms

The `motherduck` extension supports the following platforms:

- `linux_amd64`
- `linux_arm64`
- `osx_amd64`
- `osx_arm64`
- `windows_amd64`

The `windows_arm64` platform is currently not supported.

MotherDuck Documentation

For more details, see the MotherDuck documentation.

MySQL Extension

The `mysql` extension allows DuckDB to directly read and write data from/to a running MySQL instance. The data can be queried directly from the underlying MySQL database. Data can be loaded from MySQL tables into DuckDB tables, or vice versa.

Installing and Loading

To install the `mysql` extension, run:

```
INSTALL mysql;
```

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

```
LOAD mysql;
```

Reading Data from MySQL

To make a MySQL database accessible to DuckDB use the ATTACH command with the `mysql` or the `mysql_scanner` type:

```
ATTACH 'host=localhost user=root port=0 database=mysql' AS mysqldb (TYPE mysql);
USE mysqldb;
```

> **Warning.** Avoid including credentials, such as a password, directly in the connection string. If a connection error occurs, the full connection string, including your credentials, may appear in the resulting error message. For better security, store credentials using secrets instead, as described in Configuring via Secrets.

Configuration

The connection string determines the parameters for how to connect to MySQL as a set of `key=value` pairs. Any options not provided are replaced by their default values, as per the table below. Connection information can also be specified with environment variables. If no option is provided explicitly, the MySQL extension tries to read it from an environment variable.

Setting	Default	Environment variable
database	NULL	MYSQL_DATABASE
host	localhost	MYSQL_HOST
password		MYSQL_PWD
port	0	MYSQL_TCP_PORT
socket	NULL	MYSQL_UNIX_PORT
user	*current user*	MYSQL_USER
ssl_mode	preferred	
ssl_ca		
ssl_capath		
ssl_cert		
ssl_cipher		

Setting	Default	Environment variable
ssl_crl		
ssl_crlpath		
ssl_key		

Configuring via Secrets

MySQL connection information can also be specified with secrets. The following syntax can be used to create a secret.

```
CREATE SECRET (
    TYPE mysql,
    HOST '127.0.0.1',
    PORT 0,
    DATABASE mysql,
    USER 'mysql',
    PASSWORD ''
);
```

The information from the secret will be used when ATTACH is called. We can leave the connection string empty to use all of the information stored in the secret.

```
ATTACH '' AS mysql_db (TYPE mysql);
```

We can use the connection string to override individual options. For example, to connect to a different database while still using the same credentials, we can override only the database name in the following manner.

```
ATTACH 'database=my_other_db' AS mysql_db (TYPE mysql);
```

By default, created secrets are temporary. Secrets can be persisted using the `CREATE PERSISTENT SECRET` command. Persistent secrets can be used across sessions.

Managing Multiple Secrets Named secrets can be used to manage connections to multiple MySQL database instances. Secrets can be given a name upon creation.

```
CREATE SECRET mysql_secret_one (
    TYPE mysql,
    HOST '127.0.0.1',
    PORT 0,
    DATABASE mysql,
    USER 'mysql',
    PASSWORD ''
);
```

The secret can then be explicitly referenced using the SECRET parameter in the ATTACH.

```
ATTACH '' AS mysql_db_one (TYPE mysql, SECRET mysql_secret_one);
```

SSL Connections

The `ssl` connection parameters can be used to make SSL connections. Below is a description of the supported parameters.

Setting	Description
ssl_mode	The security state to use for the connection to the server: `disabled, required, verify_ca, verify_identity or preferred` (default: `preferred`)
ssl_ca	The path name of the Certificate Authority (CA) certificate file

Setting	Description
ssl_capath	The path name of the directory that contains trusted SSL CA certificate files
ssl_cert	The path name of the client public key certificate file
ssl_cipher	The list of permissible ciphers for SSL encryption
ssl_crl	The path name of the file containing certificate revocation lists
ssl_crlpath	The path name of the directory that contains files containing certificate revocation lists
ssl_key	The path name of the client private key file

Reading MySQL Tables

The tables in the MySQL database can be read as if they were normal DuckDB tables, but the underlying data is read directly from MySQL at query time.

```
SHOW ALL TABLES;
```

name
signed_integers

```
SELECT * FROM signed_integers;
```

t	s	m	i	b
-128	-32768	-8388608	-2147483648	-9223372036854775808
127	32767	8388607	2147483647	9223372036854775807
NULL	NULL	NULL	NULL	NULL

It might be desirable to create a copy of the MySQL databases in DuckDB to prevent the system from re-reading the tables from MySQL continuously, particularly for large tables.

Data can be copied over from MySQL to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM mysqlscanner.mysql_table;
```

Writing Data to MySQL

In addition to reading data from MySQL, create tables, ingest data into MySQL and make other modifications to a MySQL database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a MySQL database to Parquet, or read data from a Parquet file into MySQL.

Below is a brief example of how to create a new table in MySQL and load data into it.

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql);
CREATE TABLE mysql_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

Many operations on MySQL tables are supported. All these operations directly modify the MySQL database, and the result of subsequent operations can then be read using MySQL. Note that if modifications are not desired, ATTACH can be run with the READ_ONLY property which prevents making modifications to the underlying database. For example:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql, READ_ONLY);
```

Supported Operations

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE mysql_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM mysql_db.tbl;
```

id	name
42	DuckDB

COPY

```
COPY mysql_db.tbl TO 'data.parquet';
COPY mysql_db.tbl FROM 'data.parquet';
```

You may also create a full copy of the database using the COPY FROM DATABASE statement:

```
COPY FROM DATABASE mysql_db TO my_duckdb_db;
```

UPDATE

```
UPDATE mysql_db.tbl
SET name = 'Woohoo'
WHERE id = 42;
```

DELETE

```
DELETE FROM mysql_db.tbl
WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE mysql_db.tbl
ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE mysql_db.tbl;
```

CREATE VIEW

```
CREATE VIEW mysql_db.v1 AS SELECT 42;
```

CREATE SCHEMA and DROP SCHEMA

```
CREATE SCHEMA mysql_db.s1;
CREATE TABLE mysql_db.s1.integers (i INTEGER);
INSERT INTO mysql_db.s1.integers VALUES (42);
SELECT * FROM mysql_db.s1.integers;
```

i
42

```
DROP SCHEMA mysql_db.s1;
```

Transactions

```
CREATE TABLE mysql_db.tmp (i INTEGER);
BEGIN;
INSERT INTO mysql_db.tmp VALUES (42);
SELECT * FROM mysql_db.tmp;
```

This returns:

i
42

```
ROLLBACK;
SELECT * FROM mysql_db.tmp;
```

This returns an empty table.

> The DDL statements are not transactional in MySQL.

Running SQL Queries in MySQL

The `mysql_query` Table Function

The `mysql_query` table function allows you to run arbitrary read queries within an attached database. `mysql_query` takes the name of the attached MySQL database to execute the query in, as well as the SQL query to execute. The result of the query is returned. Single-quote strings are escaped by repeating the single quote twice.

```
mysql_query(attached_database::VARCHAR, query::VARCHAR)
```

For example:

```
ATTACH 'host=localhost database=mysql' AS mysqldb (TYPE mysql);
SELECT * FROM mysql_query('mysqldb', 'SELECT * FROM cars LIMIT 3');
```

The `mysql_execute` Function

The `mysql_execute` function allows running arbitrary queries within MySQL, including statements that update the schema and content of the database.

```
ATTACH 'host=localhost database=mysql' AS mysqldb (TYPE mysql);
CALL mysql_execute('mysqldb', 'CREATE TABLE my_table (i INTEGER)');
```

Routing a Session to MySQL

> New As of DuckDB 2.0, you can route an entire session to an attached MySQL database with the CONNECT statement. While connected, the remote pushdown optimizer ships queries directly to the MySQL server instead of pulling tables over the wire, and `DISCONNECT` returns the session to the local database.

Settings

Name	Description	Default
`mysql_bit1_as_boolean`	Whether or not to convert `BIT(1)` columns to `BOOLEAN`	`true`
`mysql_debug_show_queries`	DEBUG SETTING: print all queries sent to MySQL to stdout	`false`
`mysql_enable_filter_pushdown`	Whether or not to use filter pushdown (without predicate analyzer)	`true`
`mysql_enable_transactions`	Whether to run `START TRANSACTION` / `COMMIT` / `ROLLBACK` on MySQL connections	`true`
`mysql_incomplete_dates_as_nulls`	Whether to return `DATE`s with a zero month or day as `NULL`s	`false`
`mysql_pool_acquire_mode`	How to acquire connections from the pool: `force` (always connect, ignoring the pool limit), `wait` (block until one is available), or `try` (fail immediately if none is available)	`force`
`mysql_pool_connection_idle_timeout_millis`	Maximum time in milliseconds a connection can sit idle in the cache before being closed	`60000`
`mysql_pool_connection_max_lifetime_millis`	Maximum age in milliseconds of a pooled connection since it was first opened; when exceeded, the connection is closed instead of being returned to the cache (`0` disables it)	`0`
`mysql_pool_enable_reaper_thread`	Whether to run a dedicated thread that periodically scans the pool and removes expired connections	`true`
`mysql_pool_enable_thread_local_cache`	Enable thread-local connection caching for faster same-thread connection reuse	`false`
`mysql_pool_size`	Maximum number of connections per MySQL catalog	automatic (based on the CPU count)
`mysql_pool_wait_timeout_millis`	Timeout in milliseconds when waiting for a connection from the pool	`30000`
`mysql_session_time_zone`	Session time zone to set for newly opened connections to the MySQL server	`''`
`mysql_time_as_time`	Whether or not to convert MySQL's `TIME` columns to DuckDB's `TIME`	`false`

Name	Description	Default
`mysql_tinyint1_as_boolean`	Whether or not to convert `TINYINT(1)` columns to `BOOLEAN`	`true`

Schema Cache

To avoid having to continuously fetch schema data from MySQL, DuckDB keeps schema information – such as the names of tables, their columns, etc. – cached. If changes are made to the schema through a different connection to the MySQL instance, such as new columns being added to a table, the cached schema information might be outdated. In this case, the function `mysql_clear_cache` can be executed to clear the internal caches.

```
CALL mysql_clear_cache();
```

ODBC

ODBC Extension

> Please note that DuckDB also offers an ODBC client, which allows you to connect *to DuckDB* via ODBC from other applications.

The `odbc_scanner` extension allows connecting to other databases (using their ODBC drivers) and run queries with the `odbc_query` or copy data from DuckDB with the `odbc_copy` functions. The extension is also available under the alias `odbc`.

Installing and Loading

> On Linux and macOS the extension requires unixODBC driver manager to be installed. See below for installation instructions.

The extension can be installed automatically, but needs to be loaded manually with:

```
LOAD odbc;
```

Usage Example

```
-- load extension
LOAD odbc;

-- open ODBC connection to a remote DB
SET VARIABLE conn = odbc_connect('Driver={Oracle driver};DBQ=//127.0.0.1:1521/XE', 'scott', 'tiger');

-- simple query
FROM odbc_query(getvariable('conn'), 'SELECT SYSTIMESTAMP FROM DUAL');

-- query with parameters
FROM odbc_query(getvariable('conn'),
    'SELECT CAST(? AS NVARCHAR2(2)) || CAST(? AS VARCHAR2(5)) FROM DUAL',
    params = row('🦆', 'quack'));

-- copy data into remote DB
FROM odbc_copy(getvariable('conn'),
    source_file = 'https://blobs.duckdb.org/nl_stations.csv',
    dest_table = 'NL_TRAIN_STATIONS',
    create_table = true);

-- close connection
SELECT odbc_close(getvariable('conn'));
```

Installing the Nightly Version

ODBC extension is built using the version-independent DuckDB C API. The same binary (for the specific platform, for example: `windows_amd64`) can be installed and loaded on DuckDB version `1.2.0` or any newer version.

Binaries with the most recent changes, that are published to the DuckDB nightly repository, can be installed the following way:

```
INSTALL 'http://nightly-extensions.duckdb.org/v1.2.0/<platform>/odbc_scanner.duckdb_extension.gz';
```

> The URL with the version `1.2.0` in it should be used even if you are running later version of DuckDB.

Where the <platform>{:.language-sql .highlight} is one of:

- `linux_amd64`
- `linux_arm64`
- `linux_amd64_musl`
- `linux_arm64_musl`
- `osx_amd64`
- `osx_arm64`
- `windows_amd64`
- `windows_arm64`

To update installed extension to the latest nightly version run:

```
FORCE INSTALL 'http://nightly-extensions.duckdb.org/v1.2.0/<platform>/odbc_scanner.duckdb_extension.gz';
```

Installed version (commit ID) can be checked using the following query:

```
FROM duckdb_extensions()
WHERE extension_name = 'odbc_scanner';
```

To install a version built from a specific commit run:

```
FORCE INSTALL 'http://nightly-extensions.duckdb.org/odbc_scanner/<7_character_commit_
id>/v1.2.0/<platform>/odbc_scanner.duckdb_extension.gz';
```

Support Status of DBMS-Specific Types

Tier 1:

- Oracle: types coverage status
- SQL Server: types coverage status
- DB2: types coverage status

Tier 2:

- PostgreSQL: basic types covered
- MySQL/MariaDB: basic types covered
- Firebird: types coverage status

Tier 3:

- Snowflake: types coverage status
- ClickHouse: basic types covered
- Spark: basic types covered
- Arrow Flight SQL: basic types covered

Installing unixODBC Driver Manager on Linux or macOS

On Linux `unixODBC` can be installed using the system package manager. Depending on the Linux distribution one of the following installation commands can be used.

Debian, Ubuntu:

```
sudo apt-get install unixodbc
```

RHEL, Alma, Rocky, Amazon, Fedora:

```
sudo dnf install unixODBC
```

Alpine:

```
sudo apk add unixodbc
```

On macOS unixODBC can be installed using the Homebrew package manager:

```
brew install unixodbc
```

To use legacy `x86_64` ODBC drivers under the Rosetta translator, the unixODBC must be installed using the `x86_64` version of Homebrew:

```
arch -x86_64 /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
/usr/local/bin/brew install unixodbc
```

Connection String Examples

ODBC connection can be established using a data source name in a form `DSN=data_source1_name` or without a configured data source in a form `Driver={Driver name};parameter1=values1;...`.

`odbc_list_drivers` and `odbc_list_data_sources` functions can be used to find out available drivers and data sources.

Example of connection strings without a configured data source:

Oracle:

```
Driver={Oracle in instantclient_23_0};DBQ=//127.0.0.1:1521/XE;UID=scott;PWD=tiger;
```

SQL Server:

```
Driver={ODBC Driver 18 for SQL
Server};Server=tcp:127.0.0.1,1433;UID=sa;PWD=pwd;TrustServerCertificate=Yes;Database=test_db;
```

DB2:

```
Driver={IBM DB2 ODBC DRIVER};HostName=127.0.0.1;Port=50000;Database=testdb;UID=db2inst1;PWD=pwd;
```

PostgreSQL:

```
Driver={PostgreSQL Unicode};Server=127.0.0.1;Port=5432;Username=postgres;Password=postgres;Database=test_
db;
```

MySQL/MariaDB:

```
Driver={MariaDB ODBC 3.1 Driver};SERVER=127.0.0.1;PORT=3306;USER=root;PASSWORD=root;DATABASE=test_db;
```

Firebird:

```
Driver={Firebird ODBC Driver};Database=127.0.0.1/3050:C:/path/to/test.fdb;UID=SYSDBA;PWD=pwd;CHARSET=UTF8;
```

Snowflake:

```
Driver={SnowflakeDSIIDriver};Server=foobar-ab12345.snowflakecomputing.com;Database=SNOWFLAKE_SAMPLE_
DATA;UID=username;PWD=pwd;
```

ClickHouse:

```
Driver={ClickHouse ODBC Driver (Unicode)};Server=127.0.0.1;Port=8123;
```

Spark:

```
Driver={Simba Spark ODBC Driver};Host=127.0.0.1;Port=10000;
```

Arrow Flight SQL (Dremio ODBC + GizmoSQL):

```
Driver={Dremio Flight SQL ODBC Driver};Host=127.0.0.1;Port=31337;UID=gizmosql_username;PWD=gizmosql_
password;useEncryption=true;
```

Query Parameters

When a DuckDB query is run using prepared statement, it is possible to pass input parameters from the client code. The extension allows to forward such input parameters over ODBC API to the queries to remote databases.

Two methods of passing query parameters are supported, using either `params` or `params_handle` named argument to `odbc_query` function.

`params` argument takes a STRUCT value as an input. Struct field names are ignored, so the `row()` function can be used to create a STRUCT value inline:

```
FROM odbc_query(
    getvariable('conn'),
    '
      SELECT CAST(? AS VARCHAR2(3)) || CAST(? AS VARCHAR2(3)) FROM DUAL
    ',
    params = row(?, ?))
```

If we prepare this query with `duckdb_prepare()`, bind `foo` and `bar` VARCHAR values to it with `duckdb_bind_value()` and execute it with `duckdb_execute_prepared()` - the input parameters `foo` and `bar` will be forwarded to the ODBC query in the remote DB.

The problem with this approach, is that DuckDB is unable to resolve parameter types (specified in the outer query) before `duckdb_execute_prepared()` is called - such types may be different in subsequent invocations of `duckdb_execute_prepared()` and there is no way to specify these types explicitly.

This will result in re-preparing the inner query in remote DB every time `duckdb_execute_prepared()` is called.

To avoid this problem is it possible to use 2-step parameter binding with `params_handle` named argument to `odbc_query`:

```
-- create parameters handle
SET VARIABLE params = odbc_create_params();

-- when 'duckdb_prepare()' is called, the inner query will be prepared in the remote DB
FROM odbc_query(
    getvariable('conn'),
    '
      SELECT CAST(? AS VARCHAR2(3)) || CAST(? AS VARCHAR2(3)) FROM DUAL
    ',
    params_handle = getvariable('params'));

-- now we can repeatedly bind new parameters to the handle using 'odbc_bind_params()'
-- and call 'duckdb_execute_prepared()' to run the prepared query with
-- these new parameters in remote DB
SELECT odbc_bind_params(getvariable('conn'), getvariable('params'), row(?, ?));
```

Parameter handle is tied to the prepared statement and will be freed when the statement is destroyed.

Connections and Concurrency

DuckDB uses a multi-threaded execution engine to run parts of the query in parallel. ODBC drivers may or may not support using the same connection from different threads concurrently. To prevent possible concurrency problems the extension does not allow to use the same connection from multiple threads. For example, the following query:

```
FROM odbc_query(getvariable('conn'), 'SELECT ''foo'' col1 FROM DUAL')
UNION ALL
FROM odbc_query(getvariable('conn'), 'SELECT ''bar'' col1 FROM DUAL');
```

will fail with:

```
Invalid Input Error:
'odbc_query' error: ODBC connection not found on global init, id: 139760181976192
```

This can be avoided by using multiple ODBC connections:

```
FROM odbc_query(getvariable('conn1'), 'SELECT ''foo'' col1 FROM DUAL')
UNION ALL
FROM odbc_query(getvariable('conn2'), 'SELECT ''bar'' col1 FROM DUAL');
```

Or by disabling multi-threaded execution setting `threads` DuckDB option to `1`.

Transaction Management

According to ODBC specification, connections to remote DB are expected to have auto-commit mode enabled by default.

As a general rule, transaction commands `BEGIN TRANSACTION`/`COMMIT`/`ROLLBACK` are not supposed to be sent over ODBC as SQL commands. Doing so may or may not be supported by the particular driver. Instead ODBC provides the API to manage transactions.

This API is exposed in the following functions:

- `odbc_begin_transaction`
- `odbc_commit`
- `odbc_rollback`

When `odbc_begin_transaction` is called on the connection, the auto-commit mode on this connection is disabled and an implicit transaction is started. There is currently no support for enabling auto-commit back on such connection.

After the transaction is started, call `odbc_commit` or `odbc_rollback` to complete this transaction. After the completion is performed, new implicit transaction is started on this connection automatically.

Performance

ODBC is not a high-performance API, `odbc_query` uses multiple API calls per-row and performs `UCS-2` to `UTF-8` conversion for every `VARCHAR` value. Besides that, query processing is strictly single-threaded.

When submitting issues related only to performance please check the performance in comparable scenarios, for example with pyodbc.

ODBC Extension Functions

- `odbc_begin_transaction`
- `odbc_bind_params`
- `odbc_close`
- `odbc_commit`
- `odbc_connect`
- `odbc_copy`
- `odbc_create_params`
- `odbc_list_data_sources`
- `odbc_list_drivers`
- `odbc_query`
- `odbc_rollback`

odbc_begin_transaction

```
odbc_begin_transaction(conn_handle BIGINT) -> VARCHAR
```

Sets the `SQL_ATTR_AUTOCOMMIT` attribute to `SQL_AUTOCOMMIT_OFF` on the specified connection thus effectively starting an implicit transaction. `odbc_commit` or `odbc_rollback` must be called on such connection to complete the transaction. The completion starts another implicit transaction on this connection. See Transaction management for details.

Parameters:

- `conn_handle` (`BIGINT`): ODBC connection handle created with `odbc_connect`

Returns:

Always returns `NULL` (`VARCHAR`).

Example:

```
SELECT odbc_begin_transaction(getvariable('conn'))
```

odbc_bind_params

```
odbc_bind_params(conn_handle BIGINT, params_handle BIGINT, params STRUCT) -> BIGINT
```

Binds specified parameter values to the specified parameters handle. Only necessary with 2-step parameters binding, see Query parameters for details.

Parameters:

- `conn_handle` (`BIGINT`): ODBC connection handle created with `odbc_connect`
- `params_handle` (`BIGINT`): parameters handle created with `odbc_create_params`
- `params` (`STRUCT`): parameters values

Returns:

Parameters handle (`BIGINT`), the same one that was passed as a second argument.

Example:

```
SELECT odbc_bind_params(getvariable('conn'), getvariable('params1'), row(42, 'foo'))
```

odbc_close

```
odbc_close(conn_handle BIGINT) -> VARCHAR
```

Closes specified ODBC connection to a remote DB. Does not throw errors if the connection is already closed.

Parameters:

- `conn_handle` (`BIGINT`): ODBC connection handle created with `odbc_connect`

Returns:

Always returns `NULL` (`VARCHAR`).

Example:

```
SELECT odbc_close(getvariable('conn'))
```

odbc_commit

```
odbc_commit(conn_handle BIGINT) -> VARCHAR
```

Calls SQLEndTran with SQL_COMMIT argument on the specified connection, completing the current transaction. odbc_begin_transaction must be called on this connection before this call for the completion to be effective. See Transaction management for details.

Parameters:

- conn_handle (BIGINT): ODBC connection handle created with odbc_connect

Returns:

Always returns NULL (VARCHAR).

Example:

```
SELECT odbc_commit(getvariable('conn'))
```

odbc_connect

```
odbc_connect(conn_string VARCHAR) -> BIGINT
```

```
odbc_connect(conn_string VARCHAR, username VARCHAR, password VARCHAR) -> BIGINT
```

Opens an ODBC connection to a remote DB.

If username and password (positional) parameters are specified, they are appended to the connection string as UID and PWD.

Parameters:

- conn_string (VARCHAR): ODBC connection string, passed to the Driver Manager.

Returns:

Connection handle that can be placed into a VARIABLE. Connection is not closed automatically, must be closed with odbc_close.

Example:

```
SET VARIABLE conn = odbc_connect('Driver={Oracle Driver};DBQ=//127.0.0.1:1521/XE;UID=scott;PWD=tiger')
```

```
SET VARIABLE conn = odbc_connect('Driver={Oracle Driver};DBQ=//127.0.0.1:1521/XE', 'scott', 'tiger')
```

odbc_copy

```
odbc_copy(conn_handle BIGINT, [, <optional named parameters>]) -> TABLE
```

```
odbc_copy(conn_string VARCHAR, [, <optional named parameters>]) -> TABLE
```

Copies rows from a DuckDB accessible file or table into the remote DB.

> **Warning.** Using `odbc_copy` from Python Relational API.
>
> `odbc_copy` is a table function that returns one row for each 2048 copied rows. Called from Python as `duckdb.sql("SELECT * FROM odbc_copy(...)")`, it is subject to lazy evaluation: no rows are copied until a method that reads the relation is called and all result rows are consumed, and every such read copies the rows again. To copy once, use `duckdb.sql("CALL odbc_copy(...)")`, which runs at the call and consumes all rows, or `duckdb.execute("SELECT * FROM odbc_copy(...)")` followed by `fetchall()` or `complete()`.

Parameters:

- `conn_handle_or_string` (`BIGINT` or `VARCHAR`), one of:
  - ODBC connection handle created with `odbc_connect`
  - ODBC connection string, intended for one-off queries, in this case new ODBC connection will be opened and will be closed automatically after the query is complete

Optional named parameters (source):

> Source query is executed using a separate DB instance from the instance on which `odbc_copy` is being called. Thus `source_query` cannot refer to pre-existing in-memory tables and cannot open currently opened DuckDB files. As a workaround, for complex source queries it is suggested to export the query result into a local Parquet file first and then run `odbc_copy` on that file.

- `source_conn_string` (`VARCHAR`, default: `:memory:`): DuckDB connection string to the source DB, example: `ducklake:postgres:postgresql://username:pwd@127.0.0.1:5432/lake1`
- `source_file` (`VARCHAR`): path to a Parquet, CSV or JSON file (remote or local) to be read with DuckDB, example: `https://blobs.duckdb.org/nl_stations.csv`, equivalent to `source_query='SELECT * FROM '<source_file>'`
- `source_query` (`VARCHAR`): DuckDB SQL query to read the data, example: `FROM nl_train_stations`
- `source_queries` (`LIST(VARCHAR)`): multiple DuckDB SQL queries executed one by one, last query must return the result set to copy, results of previous queries are discarded, results of all queries are materialized in memory, example:

```
source_queries=[
  'CREATE SECRET s (TYPE s3 [...])',
  'FROM nl_train_stations'
],
```

- `source_limit` (`UBIGINT`, default: `0`): the number of records to read from source query/file at once, when this option is specified - the source query is run multiple times appending `LIMIT <limit> OFFSET <offset>` to it, must be more or equal to `2048`, `2048` must be dividable by it without a remainder

Optional named parameters (destination):

- `dest_table` (`VARCHAR`): destination table name in remote DB, will be used in `INSERT` and `CREATE TABLE` queries, cannot be specified if `dest_query` is specified; different DBs have different rules regarding case sensitivity and the default case, thus the name of the destination table may need to be specified in upper-case: `TAB1` or in quoted form: `"tab1"` (or with a schema name: `"schema1"."tab1"`)
- `dest_query` (`VARCHAR`): query to be executed in remote DB for each source batch, must have the number of ODBC parameter placeholders `?` equal to the `source_columns_count * batch_size`, cannot be specified if `dest_table` is specified, example: `CALL import_city(?,?,?,?)`
- `dest_query_single` (`VARCHAR`): only used when the `batch_size>0` and the number of rows read in the last source batch are less than the `batch_size`, in this case used instead of `dest_query`, must have the number of ODBC parameter placeholders `?` equal to the `source_columns_count`

Optional named parameters (create table):

- `create_table` (`BOOLEAN`, default: `FALSE`): whether to create a table in the destination remote DB using the column names and column types from the source query, effectively implements CTAS (create table as select)

- `column_types` (`MAP(VARCHAR, VARCHAR)`): when `create_table=TRUE` is specified, allows to provide/override the type mapping between source DuckDB types and destination RDBMS types, example:

```
create_table=TRUE,
column_types=MAP {
    'DUCKDB_TYPE_VARCHAR': 'VARCHAR2(10)',
    'DUCKDB_TYPE_DECIMAL': 'NUMBER({typmod1},{typmod2})'}
```

- `column_quotes` (VARCHAR, default: `"`): quotation character (or string) to be used to quote column names in the generated `CREATE TABLE` and `INSERT` queries
- `commit_after_create_table` (BOOLEAN, default: `FALSE`): whether to issue a `COMMIT` after executing `CREATE TABLE`, enabled automatically for Firebird

Optional named parameters (query parameters handling):

- `decimal_params_as_chars` (BOOLEAN, default: `false`): pass `DECIMAL` parameters as VARCHARs
- `integral_params_as_decimals` (BOOLEAN, default: `false`): pass (unsigned) `TINYINT`, `SMALLINT`, `INTEGER` and `BIGINT` parameters as `SQL_C_NUMERIC`.

Optional named parameters (other):

- `batch_size` (UINTEGER, default: 16): number of records to be inserted (or executed in case of `dest_query`) in a single `SQLExecute` ODBC call to remote DB, allowed values: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
- `use_insert_all` (BOOLEAN, default: `FALSE`): generate `INSERT ALL` batch insert query instead of batch insert with `INSERT ... VALUES (...), (...), ... (...)`, enabled automatically for Oracle
- `use_insert_union` (BOOLEAN, default: `FALSE`): generate `INSERT ... SELECT FROM ... UNION ALL ...` batch insert query instead of batch insert with `INSERT ... VALUES (...), (...), ... (...)`, enabled automatically for Firebird
- `dummy_table_name` (VARCHAR): name of the dummy table use for `INSERT ALL` and `INSERT UNION` queries, `dual` for Oracle
- `copy_in_transaction` (BOOLEAN, default: `TRUE`): begin a transaction in remote DB for this copy call, commit transaction when all rows are processed, roll it back on error
- `max_records_in_transaction` (UBIGINT, default: `0`): when specified causes the remote transaction to be committed every time after the specified number of rows is processed
- `close_connection` (BOOLEAN, default: `false`): closes the passed connection after the function call is completed, intended to be used with one-shot invocations of the `odbc_copy`

Returns:

A table with the following columns:

- `completed` (BOOLEAN): a flag whether this output row is the last row in result set
- `rows_processed` (UBIGINT): a number of rows read from the source
- `elapsed_seconds` (FLOAT): a number of seconds passed after the copy process has started
- `rows_per_second` (FLOAT): a number of rows processed in one second
- `table_ddl` (VARCHAR): generated `CREATE TABLE` query that was executed in remote DB before starting the copy process

One resulting row is emitted for every `2048` rows read from source. Only the last row has the `completed=TRUE` and non null `table_ddl` (only when `create_table=TRUE` is specified) values.

Examples:

```
FROM odbc_copy(getvariable('conn'),
  source_file='https://blobs.duckdb.org/nl_stations.csv',
  dest_table='NL_TRAIN_STATIONS',
  create_table=TRUE)
```

```
FROM odbc_copy(getvariable('conn'),
  source_conn_string='ducklake:postgres:postgresql://username:pwd@127.0.0.1:5432/lake1',
```

```
  source_queries=[
    'CREATE SECRET s (TYPE s3 [...])',
    'FROM nl_train_stations'
  ],
  dest_table='NL_TRAIN_STATIONS',
  create_table=TRUE,
  batch_size=32,
  max_records_in_transaction=42);
```

odbc_create_params

```
odbc_create_params() -> BIGINT
```

Creates a parameters handle. Only necessary with 2-step parameters binding, see Query parameters for details.

Parameters:

None.

Returns:

Parameters handle (`BIGINT`). When the handle is passed to `odbc_query` it gets tied to the underlying prepared statement and is closed automatically when the statement is closed.

Example:

```
SET VARIABLE params1 = odbc_create_params()
```

odbc_list_data_sources

```
odbc_list_data_sources() -> TABLE(name VARCHAR, description VARCHAR, type VARCHAR)
```

Returns the list of ODBC data sources registered in the OS. Uses driver manager call `SQLDataSources`.

Parameters:

None.

Returns:

A table with the following columns:

- `name` (VARCHAR): data source name
- `description` (VARCHAR): data source description
- `type` (VARCHAR): data source type, USER or SYSTEM

Example:

```
FROM odbc_list_data_sources()
```

odbc_list_drivers

```
odbc_list_drivers() -> TABLE(description VARCHAR, attributes MAP(VARCHAR, VARCHAR))
```

Returns the list of ODBC drivers registered in the OS. Uses driver manager call `SQLDrivers`.

Parameters:

None.

Returns:

A table with the following columns:

- `description` (VARCHAR): driver description
- `attributes` (`MAP(VARCHAR, VARCHAR)`): driver attributes as a `name->value` map

Example:

```
FROM odbc_list_drivers()
```

odbc_query

```
odbc_query(conn_handle BIGINT, query VARCHAR[, <optional named parameters>]) -> TABLE
```

```
odbc_query(conn_string VARCHAR, query VARCHAR[, <optional named parameters>]) -> TABLE
```

Runs specified query in a remote DB and returns the query results table.

Parameters:

- `conn_handle_or_string` (`BIGINT` or VARCHAR), one of:
  - ODBC connection handle created with `odbc_connect`
  - ODBC connection string, intended for one-off queries, in this case new ODBC connection will be opened and will be closed automatically after the query is complete
- `query` (VARCHAR): SQL query, passed to the remote DBMS

Optional named parameters that can be used to pass query parameters:

- `params` (STRUCT): query parameters to pass to remote DBMS
- `params_handle` (`BIGINT`): parameters handle created with `odbc_create_params`. Only used with 2-step parameters binding, see Query parameters for details.

Optional named parameters that can change types mapping:

The extension supports a number of options that can be used to change how the query parameters are passed and how the resulting data is handled. For known DBs these options are set automatically. They also can be passed as named parameters to `odbc_query` function to override the autoconfiguration:

- `decimal_columns_as_chars` (`BOOLEAN`, default: `false`): read `DECIMAL` values as VARCHARs that are parsed back into `DECIMAL`s before returning them to client
- `decimal_columns_precision_through_ard` (`BOOLEAN`, default: `false`): when reading a `DECIMAL` specify its `precision` and `scale` through "Application Row Descriptor"

- `decimal_columns_as_ard_type` (BOOLEAN, default: `false`): when reading a DECIMAL use SQL_ARD_TYPE instead of SQL_C_NUMERIC
- `decimal_params_as_chars` (BOOLEAN, default: `false`): pass DECIMAL parameters as VARCHARs
- `integral_params_as_decimals` (BOOLEAN, default: `false`): pass (unsigned) TINYINT, SMALLINT, INTEGER and BIGINT parameters as SQL_C_NUMERIC.
- `reset_stmt_before_execute` (BOOLEAN, default: `false`): reset the prepared statement (using `SQLFreeStmt(h, SQL_CLOSE)`) before executing it
- `time_params_as_ss_time2` (BOOLEAN, default: `false`): pass TIME parameters as SQL Server's TIME2 values
- `timestamp_columns_as_timestamp_ns` (BOOLEAN, default: `false`): read TIMESTAMP-like (`TIMESTAMP WITH LOCAL TIME ZONE`, DATETIME2, TIMESTAMP_NTZ etc) columns with nanosecond precision (with nine fractional digits)
- `timestamp_columns_with_typename_date_as_date` (BOOLEAN, default: `false`): read TIMESTAMP columns that have a type name DATE as DuckDB DATEs
- `timestamp_max_fraction_precision` (UTINYINT, default: 9): maximum number of fractional digits to use when reading a TIMESTAMP column with nanosecond precision
- `timestamp_params_as_sf_timestamp_ntz` (BOOLEAN, default: `false`): pass TIMESTAMP parameters as Snowflake's TIMESTAMP_NTZ
- `timestamptz_params_as_ss_timestampoffset` (BOOLEAN, default: `false`): pass TIMESTAMP_TZ parameters as SQL Server's DATETIMEOFFSET
- `var_len_data_single_part` (BOOLEAN, default: `false`): read long VARCHAR or VARBINARY values as a single read (used when a driver does not support Retrieving Variable-Length Data in Parts)
- `var_len_params_long_threshold_bytes` (UINTEGER, default: `4000`): a length threshold after that SQL_WVARCHAR parameters are passed as SQL_WLONGVARCHAR
- `enable_columns_binding` (BOOLEAN, default: `false`): whether to allow using `SQLBindCol` instead of `SQLGetData` for fixed-size columns

Other optional named parameters:

- `ignore_exec_failure` (BOOLEAN, default: `false`): when a query, that is run in remote DB, can be prepared successfully, but may or may not fail at execution time (for example, because of schema state like table existence), then this flag can be used to not throw an error when query execution fails. Empty result set is returned if query execution fails.
- `close_connection` (BOOLEAN, default: `false`): closes the passed connection after the function call is completed, intended to be used with one-shot invocations of the `odbc_query`, example:

```
FROM odbc_query(
   odbc_connect('Driver={Oracle Driver};DBQ=//127.0.0.1:1521/XE', 'scott', 'tiger'),
   'SELECT 42 FROM dual',
   close_connection=TRUE);
```

Returns:

A table with the query result.

Example:

```
FROM odbc_query(getvariable('conn'),
  'SELECT CAST(? AS NVARCHAR2(2)) || CAST(? AS VARCHAR2(5)) FROM dual',
  params=row('🦆', 'quack')
)
```

odbc_rollback

```
odbc_rollback(conn_handle BIGINT) -> VARCHAR
```

Calls `SQLEndTran` with `SQL_ROLLBACK` argument on the specified connection, completing the current transaction. `odbc_begin_transaction` must be called on this connection before this call for the completion to be effective. See Transaction management for details.

Parameters:

- `conn_handle` (`BIGINT`): ODBC connection handle created with `odbc_connect`

Returns:

Always returns `NULL` (VARCHAR).

Example:

```
SELECT odbc_rollback(getvariable('conn'))
```

Quack Extension

The `quack` extension adds support for the Quack remote protocol.

Usage

> New DuckDB 2.0.0 shipped a polished version of the Quack protocol that is faster and more stable. Expect further improvements in upcoming patch releases.

Quack is transparently autoinstalled and autoloaded on first use.

If you would like to install Quack explicitly, run:

```
INSTALL quack;
```

If you would like to load Quack explicitly, run:

```
LOAD quack;
```

For usage details, see the Quack remote protocol page.

PostgreSQL

PostgreSQL Extension

The `postgres` extension allows DuckDB to directly read and write data from a running PostgreSQL database instance. The data can be queried directly from the underlying PostgreSQL database. Data can be loaded from PostgreSQL tables into DuckDB tables, or vice versa. See the official announcement for implementation details and background.

Installing and Loading

The `postgres` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD postgres;
```

Connecting

To make a PostgreSQL database accessible to DuckDB, use the ATTACH command with the `postgres` or `postgres_scanner` type.

To connect to the `public` schema of the PostgreSQL instance running on localhost in read-write mode, run:

```
ATTACH '' AS postgres_db (TYPE postgres);
```

To connect to the PostgreSQL instance with the given parameters in read-only mode, run:

```
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE postgres, READ_ONLY);
```

By default, all schemas are attached. When working with large instances, it can be useful to only attach a specific schema. This can be accomplished using the SCHEMA command.

```
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE postgres, SCHEMA 'public');
```

The transaction isolation level can be set with the `ISOLATION_LEVEL` option. Supported values are `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`; the default is `REPEATABLE READ`.

```
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE postgres, ISOLATION_LEVEL 'READ
COMMITTED');
```

> **Warning.** Avoid including credentials, such as a password, directly in the connection string. If a connection error occurs, the full connection string, including your credentials, may appear in the resulting error message. For better security, store credentials using secrets instead, as described in Configuring via Secrets.

> **Deprecated.** The old `postgres_attach` function was removed in DuckDB 2.0. Use the ATTACH statement instead.

Configuration

The ATTACH command takes as input either a `libpq` connection string or a PostgreSQL URI.

Below are some example connection strings and commonly used parameters. A full list of available parameters can be found in the PostgreSQL documentation.

```
dbname=postgresscanner
host=localhost port=5432 dbname=mydb connect_timeout=10
```

Name	Description	Default
`dbname`	Database name	[user]
`host`	Name of host to connect to	`localhost`
`hostaddr`	Host IP address	`localhost`
`passfile`	Name of file passwords are stored in	`~/.pgpass`
`password`	PostgreSQL password	(empty)
`port`	Port number	`5432`
`user`	PostgreSQL user name	*current user*

An example URI is `postgresql://username@hostname/dbname`.

Configuring via Secrets

PostgreSQL connection information can also be specified with secrets, see details on the corresponding page.

Configuring via Environment Variables

PostgreSQL connection information can also be specified with environment variables. This can be useful in a production environment where the connection information is managed externally and passed in to the environment.

```
export PGPASSWORD="secret"
export PGHOST=localhost
export PGUSER=owner
export PGDATABASE=mydatabase
```

Then, to connect, start the `duckdb` process and run:

```
ATTACH '' AS p (TYPE postgres);
```

Usage

The tables in the PostgreSQL database can be read as if they were normal DuckDB tables, but the underlying data is read directly from PostgreSQL at query time.

```
SHOW ALL TABLES;
```

name
uuids

```
SELECT * FROM uuids;
```

u
6d3d2541-710b-4bde-b3af-4711738636bf
NULL

u
00000000-0000-0000-0000-000000000001
ffffffff-ffff-ffff-ffff-ffffffffffff

It might be desirable to create a copy of the PostgreSQL databases in DuckDB to prevent the system from re-reading the tables from PostgreSQL continuously, particularly for large tables.

Data can be copied over from PostgreSQL to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM postgres_db.postgres_tbl;
```

Writing Data to PostgreSQL

In addition to reading data from PostgreSQL, the extension allows you to create tables, ingest data into PostgreSQL and make other modifications to a PostgreSQL database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a PostgreSQL database to Parquet, or read data from a Parquet file into PostgreSQL.

Below is a brief example of how to create a new table in PostgreSQL and load data into it.

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres);
CREATE TABLE postgres_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO postgres_db.tbl VALUES (42, 'DuckDB');
```

Many operations on PostgreSQL tables are supported. All these operations directly modify the PostgreSQL database, and the result of subsequent operations can then be read using PostgreSQL. Note that if modifications are not desired, `ATTACH` can be run with the `READ_ONLY` property which prevents making modifications to the underlying database. For example:

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres, READ_ONLY);
```

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE postgres_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO postgres_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM postgres_db.tbl;
```

id	name
42	DuckDB

COPY

You can copy tables back and forth between PostgreSQL and DuckDB:

```
COPY postgres_db.tbl TO 'data.parquet';
COPY postgres_db.tbl FROM 'data.parquet';
```

These copies use PostgreSQL binary wire encoding. DuckDB can also write data using this encoding to a file which you can then load into PostgreSQL using a client of your choosing if you would like to do your own connection management:

```
COPY 'data.parquet' TO 'pg.bin' WITH (FORMAT postgres_binary);
```

The file produced will be the equivalent of copying the file to PostgreSQL using DuckDB and then dumping it from PostgreSQL using `psql` or another client:

DuckDB:

```
COPY postgres_db.tbl FROM 'data.parquet';
```

PostgreSQL:

```
\copy tbl TO 'data.bin' WITH (FORMAT BINARY);
```

You may also create a full copy of the database using the `COPY FROM DATABASE` statement:

```
COPY FROM DATABASE postgres_db TO my_duckdb_db;
```

UPDATE

```
UPDATE postgres_db.tbl
SET name = 'Woohoo'
WHERE id = 42;
```

DELETE

```
DELETE FROM postgres_db.tbl
WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE postgres_db.tbl
ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE postgres_db.tbl;
```

CREATE VIEW

```
CREATE VIEW postgres_db.v1 AS SELECT 42;
```

`CREATE SCHEMA` / `DROP SCHEMA`

```
CREATE SCHEMA postgres_db.s1;
CREATE TABLE postgres_db.s1.integers (i INTEGER);
INSERT INTO postgres_db.s1.integers VALUES (42);
SELECT * FROM postgres_db.s1.integers;
```

i
42

```
DROP SCHEMA postgres_db.s1;
```

`DETACH`

```
DETACH postgres_db;
```

Transactions

```
CREATE TABLE postgres_db.tmp (i INTEGER);
BEGIN;
INSERT INTO postgres_db.tmp VALUES (42);
SELECT * FROM postgres_db.tmp;
```

This returns:

i
42

```
ROLLBACK;
SELECT * FROM postgres_db.tmp;
```

This returns an empty table.

Running SQL Queries in PostgreSQL

The `postgres_query` Table Function

The `postgres_query` table function allows you to run arbitrary read queries within an attached database. `postgres_query` takes the name of the attached PostgreSQL database to execute the query in, as well as the SQL query to execute. The result of the query is returned. Single-quote strings are escaped by repeating the single quote twice.

```
postgres_query(attached_database::VARCHAR, query::VARCHAR)
```

For example:

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres);
SELECT * FROM postgres_query('postgres_db', 'SELECT * FROM cars LIMIT 3');
```

brand	model	color
Ferrari	Testarossa	red
Aston Martin	DB2	blue

brand	model	color
Bentley	Mulsanne	gray

The `postgres_execute` Function

The `postgres_execute` function allows running arbitrary queries within PostgreSQL, including statements that update the schema and content of the database.

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres);
CALL postgres_execute('postgres_db', 'CREATE TABLE my_table (i INTEGER)');
```

Routing a Session to PostgreSQL

> New As of DuckDB 2.0, you can route an entire session to an attached PostgreSQL database with the `CONNECT` statement. While connected, the remote pushdown optimizer ships queries directly to the PostgreSQL server instead of pulling tables over the wire, and `DISCONNECT` returns the session to the local database.

Settings

The extension exposes the following configuration parameters.

Name	Description	Default
`pg_array_as_varchar`	Read PostgreSQL arrays as varchar - enables reading mixed dimensional arrays	`false`
`pg_connection_cache`	Deprecated: whether or not to use connection pooling. To disable pooling, use `SET pg_pool_max_connections = 0` instead (see Connection Pool)	`true`
`pg_connection_limit`	Deprecated: the maximum amount of concurrent PostgreSQL connections. Use `pg_pool_max_connections` instead (see Connection Pool)	depends on the number of CPU cores
`pg_debug_show_queries`	DEBUG SETTING: print all queries sent to PostgreSQL to stdout	`false`
`pg_enable_filter_pushdown`	Whether or not to use filter pushdown	`true`
`pg_idle_in_transaction_timeout_millis`	PostgreSQL `idle_in_transaction_session_timeout` in milliseconds to set on scan connections	`NULL`
`pg_null_byte_replacement`	When writing NULL bytes to PostgreSQL, replace them with the given character	`NULL`
`pg_numeric_as_varchar`	Read PostgreSQL `NUMERIC` values without precision and scale or with precision greater than 38 as `VARCHAR` instead of `DOUBLE`	`false`
`pg_numeric_nan_as_null`	Read PostgreSQL `NUMERIC` `NaN` values as `NULL` instead of throwing an error	`true`

Name	Description	Default
`pg_oauth_token`	OAuth bearer token for PostgreSQL OAUTHBEARER authentication. Takes priority over the PGOAUTHTOKEN environment variable	`NULL`
`pg_pages_per_task`	The amount of pages per task	`1000`
`pg_query_mode`	Query mode used to read data, one of `'AUTO'`, `'COPY_BINARY'`, `'PLAIN_QUERY_BINARY'`, `'PLAIN_QUERY_TEXT'`	`AUTO`
`pg_statement_timeout_millis`	PostgreSQL statement timeout in milliseconds to set on scan connections	`NULL`
`pg_use_binary_copy`	Whether or not to use BINARY copy to read data	`true`
`pg_use_ctid_scan`	Whether or not to parallelize scanning using table ctids	`true`
`pg_use_information_schema_introspection`	Use the SQL-standard `information_schema` views instead of `pg_catalog` for schema discovery at ATTACH time. Useful for PostgreSQL-wire-compatible databases (e.g., Spanner, CockroachDB, Redshift)	`false`
`pg_use_text_protocol`	Whether or not to use the text protocol to read data. This is slower, but provides better compatibility with non-PostgreSQL systems	`false`

The connection pool settings (`pg_pool_*`) are documented on the Connection Pool page and the schema cache settings (`pg_staleness_query`, `pg_staleness_query_enabled`) in the Schema Cache section.

Schema Cache

To avoid having to continuously fetch schema data from PostgreSQL, DuckDB keeps schema information – such as the names of tables, their columns, etc. – cached. If changes are made to the schema through a different connection to the PostgreSQL instance, such as new columns being added to a table, the cached schema information might be outdated. In this case, the function `postgres_clear_cache` can be executed to clear the internal caches.

```
CALL postgres_clear_cache();
```

In version 1.5.5 a support for automatic detection of schema changes was added using a ”staleness query” (contributed by Brandon Freeman in duckdb/duckdb-postgres#514):

- when `pg_staleness_query_enabled` option (BOOLEAN, default: enabled for PostgreSQL, disabled when `pg_use_information_schema_introspection` is set) is enabled, on every catalog access a query is run that checks Postgres' `pg_class.xmin` column value in every table and reloads the cache automatically, if a change is detected
- for Postgres-wire-compatible databases a custom ”staleness query” can be set using `pg_staleness_query` option.

Working with hstore Columns

DuckDB will return data from hstore columns as VARCHAR of their text representation, like `key=>value, foo=>bar`. You can use the postgres_hstore_get function to read the value for a given key, or postgres_hstore_to_json to convert the whole set of key/value pairs to JSON for further processing.

```
SELECT postgres_hstore_get('a=>b, c=>d', 'a');
-- b

SELECT postgres_hstore_get('a=>b, c=>d', 'missingkey');
-- NULL

SELECT postgres_hstore_to_json('a=>b, c=>d, e => null');
-- {"a": "b", "c": "d", e: null}
```

PostgreSQL Extension and the Secret Manager

User credentials and other PostgreSQL database connection details can be stored using the DuckDB Secrets Manager. The following syntax can be used to create a secret:

```
CREATE SECRET (
    TYPE postgres,
    HOST '127.0.0.1',
    PORT 5432,
    DATABASE postgres,
    USER 'postgres',
    PASSWORD ''
);
```

The information from the secret will be used when ATTACH is called. We can leave the PostgreSQL connection string empty to use all of the information stored in the secret.

```
ATTACH '' AS postgres_db (TYPE postgres);
```

We can use the PostgreSQL connection string to override individual options. For example, to connect to a different database while still using the same credentials, we can override only the database name in the following manner.

```
ATTACH 'dbname=my_other_db' AS postgres_db (TYPE postgres);
```

By default, created secrets are temporary. Secrets can be persisted using the `CREATE PERSISTENT SECRET` command. Persistent secrets can be used across sessions.

Managing Multiple Secrets

Named secrets can be used to manage connections to multiple PostgreSQL database instances. Secrets can be given a name upon creation.

```
CREATE SECRET postgres_secret_one (
    TYPE postgres,
    HOST '127.0.0.1',
    PORT 5432,
    DATABASE postgres,
    USER 'postgres',
    PASSWORD ''
);
```

The secret can then be explicitly referenced using the SECRET parameter in the ATTACH.

```
ATTACH '' AS postgres_db_one (TYPE postgres, SECRET postgres_secret_one);
```

> **Warning.** Avoid including credentials, such as a password, directly in the connection string. If a connection error occurs, the full connection string, including your credentials, may appear in the resulting error message. For better security, store credentials using DuckDB-managed secrets.

Secret Configuration Options

Secrets of type `postgres` support a number of configuration options. The following options are named after the corresponding connection options of libpq:

- `HOST`
- `HOSTADDR`
- `PORT`
- `DBNAME`
- `USER`
- `PASSWORD`
- `PASSFILE`
- `REQUIRE_AUTH`
- `CHANNEL_BINDING`
- `CONNECT_TIMEOUT`
- `CLIENT_ENCODING`
- `OPTIONS`
- `APPLICATION_NAME`
- `FALLBACK_APPLICATION_NAME`
- `KEEPALIVES`
- `KEEPALIVES_IDLE`
- `KEEPALIVES_INTERVAL`
- `KEEPALIVES_COUNT`
- `TCP_USER_TIMEOUT`
- `REPLICATION`
- `GSSENCMODE`
- `SSLMODE`
- `REQUIRESSL`
- `SSLNEGOTIATION`
- `SSLCOMPRESSION`
- `SSLCERT`
- `SSLKEY`
- `SSLKEYLOGFILE`
- `SSLPASSWORD`
- `SSLCERTMODE`
- `SSLROOTCERT`
- `SSLCRL`
- `SSLCRLDIR`
- `SSLSNI`
- `REQUIREPEER`
- `SSL_MIN_PROTOCOL_VERSION`
- `SSL_MAX_PROTOCOL_VERSION`
- `MIN_PROTOCOL_VERSION`
- `MAX_PROTOCOL_VERSION`
- `KRBSRVNAME`
- `GSSLIB`
- `GSSDELEGATION`
- `SCRAM_CLIENT_KEY`

- SCRAM_SERVER_KEY
- SERVICE
- TARGET_SESSION_ATTRS
- LOAD_BALANCE_HOSTS
- OAUTH_ISSUER
- OAUTH_CLIENT_ID
- OAUTH_CLIENT_SECRET
- OAUTH_SCOPE

The following options are renamed to the corresponding keys:

- DATABASE – alias to DBNAME
- HOSTNAME – alias to HOST
- USERNAME – alias to USER

Instead of separate connection options, the full connection URI can be specified instead:

- URI – connection URI

Additional option is used for AWS RDS IAM authentication, see details in the next section below:

- AWS_RDS_SECRET – the name of the secret of type rds

AWS RDS IAM Authentication

Managed PostgreSQL databases running on RDS/Aurora services allow to use IAM authentication. In that case the authentication token is generated using AWS SDK and must be refreshed every 15 minutes.

The postgres extension supports IAM authentication, when the password is not specified in the secret, but instead one of the configured AWS Credential Providers is used to generate the password, that is refreshed by the postgres extension automatically.

Connecting with IAM Authentication from Command Line

> This section illustrates the process with psql utility, not with DuckDB. This method can be used to check the configuration before setting up DuckDB secrets.

When IAM authentication is performed using psql command line tool, the following are the AWS-recommended connection steps:

- Generate authentication token using aws CLI.
- Pass this token as a password connection option:

```
export RDSHOST="database-1-instance-1.xxx.eu-west-1.rds.amazonaws.com"
psql "host=$RDSHOST port=5432 dbname=postgres user=postgres sslmode=require password=$(aws rds
generate-db-auth-token --hostname $RDSHOST --port 5432 --username postgres --region eu-west-1)"
```

Configuring Secrets for IAM Authentication

Authentication from the postgres extension uses the same logic as with psql:

- The secret of type rds is used to generate the authentication token, it takes the same configuration parameters as the aws rds generate-db-auth-token command in the example above:

```
CREATE SECRET aws_rds_secret1 (
    TYPE rds,
    PROVIDER credential_chain,
    CHAIN 'env;sso;',
    REGION 'eu-west-1',
    RDS_USER 'postgres',
```

```
    RDS_HOST 'database-1-instance-1.xxxxxxxxxxxx.eu-west-1.rds.amazonaws.com',
    RDS_PORT '5432'
);
```

- The secret of type `postgres` is used to create the remaining of the connection string. It takes the same parameters as the `psql` utility in the example above (and additionally any relevant additional `libpq` configuration options) and requires to specify the name of the `rds` secret, that is used to generate and periodically refresh (automatically) the authentication token that is passed to server as a `password`:

```
CREATE SECRET pg_rds_secret1 (
    TYPE postgres,
    HOST 'database-1-instance-1.xxxxxxxxxxxx.eu-west-1.rds.amazonaws.com',
    PORT '5432',
    USER 'postgres',
    DATABASE 'postgres',
    SSLMODE require,
    AWS_RDS_SECRET aws_rds_secret1
);
```

The secret of type `rds` requires the `aws` extension to be installed and allows to configure AWS Credential Chain the same way as with the secret of type `s3`, see details in the AWS extension documentation.

Storing Secrets inside a PostgreSQL Database

DuckDB Secrets Manager supports pluggable Storage Providers. The `postgres` extension implements storing the secrets (of any type) as records in the PostgreSQL database table.

The following example initializes the secrets storage and insers the secret into the `duckdb_secrets` table:

```
ATTACH 'postgres:' AS p1 (
    SECRET pg_rds_secret1,
    SECRET_STORAGE_TABLE duckdb_secrets
);

CREATE OR REPLACE SECRET s3_secret1 IN postgres_p1 (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'env;sso;',
    PROFILE 'DatabaseAdministrator-<account_id>',
    REGION 'eu-west-1'
);
```

When `IN postgres_<attached_database>{:.language-sql .highlight}` clause of `CREATE SECRET` is specified, the secret is persisted to the `duckdb_secrets` table in the specified attached database.

> **Warning.** Secrets are written to the database table in unencrypted binary format. It is advised to use this provider for secrets that do not include confidential credentials (like the `s3` example above). For multi-tenant scenarios it is expected that Postgres Row-Level Security policies can be used to prevent users to see secrets of other users.

Different table name (for example, in a different schema) can be specified in the `SECRET_STORAGE_TABLE` parameter to the `ATTACH` command.

When the default `duckdb_secrets` table is used, it is not necessary to specify the `SECRET_STORAGE_TABLE` parameter. When a PostgreSQL database is attached, the table with the default name `duckdb_secrets` is probed automatically. If it exists then the Secret Storage instance is registerd for this attached database making all the stored secrets available in the current session. This allows to use such persistent secrets with "direct attach" scenarios, when only a connection string is specified to the `duckdb` command (other DuckDB clients may require slightly different syntax):

```
duckdb postgres:postgresql://username:password@127.0.0.1:5432/db1
```

This method can also be used with DuckLake (with PostgreSQL catalog), when an S3/object storage access secret is stored in the catalog database:

```
duckdb ducklake:postgres:postgresql://username:password@127.0.0.1:5432/db1
```

Secrets the are stored in the specific attached database can be listed using the following query:

```
FROM duckdb_secrets() WHERE storage = 'postgres_<attached_database>';
```

To disable the PostgreSQL Secret Storage completely pass the empty string `''` in the `SECRET_STORAGE_TABLE` parameter.

PostgreSQL Extension Connection Pool

PostgreSQL server spawns a backend process for every incoming client connection. This model leads to the two following points with the `postgres` extension that can cause performance problems:

- opening a new connection is relatively expensive
- the number of the connections opened at the same time should not be too high

To deal with these points the extensions uses an in-memory connection pool. With it a limited number of connections are kept open (and idle) after use and can be reused for subsequent queries.

Parallel Scans

Due to DuckDB's multithreaded query processing, `postgres` extension can open multiple connections even for queries that performs a scan for a single PostgreSQL table. The number of parallel queries running for a single table scan can be as high as the number of worker threads of the current DB instance (`threads` global configuration option).

When the `threads` parameter is set too high (that may be necessary for local processing of other queries on DuckDB side) this can lead to attempts to open an excessive number of PostgreSQL connections. To prevent this, these parallel queries (for the same table scan) are additionally limited by the maximum number of allowed connection in the connection pool (no more than 32 by default).

Even when the connection pool is configured to allow opening additional connections when all connection slots are taken (`force` acquire mode), the addtional parallel queries are always use the `try` mode. If connection for them is not available (the maximum number of connection in the pool is exceeded) - then that worked thread does not participate futher in the processing of this table scan, that proceeds with the worker threads that were able to acquire a connection from the pool.

See the description of the `postgres_configure_pool` function for details.

Connection Proxies

Connection pool can be the most effective when DuckDB connects directly to a PostgreSQL server and a new backend process is started for every new connection. When an intermediate transparent proxy is used between DuckDB and PostgreSQL (for example, PgBouncer) then opening a new connections only causes a new networking socket to be opened, not a full backend process.

Thus in deployment environments with connection proxies it may be better to set the idle timeout to a lower value (using `pg_pool_idle_timeout_millis` configuration option or `postgres_configure_pool` function) or to disable the pooling completely (by setting `pg_pool_max_connections = 0`).

Reaper Thread

The `postgres` extension supports running a background thread (so called "reaper thread") that periodically scans idle connections in the pool and closes the ones of them that exceed `idle_timeout_millis` or `max_lifetime_millis` values.

Separate thread is run for each connections pool, it can be enabled/disabled using `pg_pool_enable_reaper_thread` configuraton option.

When the reaper thread is not running, `idle_timeout_millis` or `max_lifetime_millis` values of the connections still can be checked, but they are only checked when a connection is taken from the pool or returned to the pool.

Thread-local Cache

Idle connection in a pool are available for any caller thread, be it a DuckDB internal worker thread or a new client thread. In some cases it may be beneficial to ensure, that subsequent queries from the same thread are run on the same connection as the first query. To support this the `pg_pool_enable_thread_local_cache` configuration option can be used - it makes an idle connection to be returned to a thread-local (and thread-private) cache instead of the main cache shared between all threads.

> **Warning.** Thread-local connection are not checked and not cleaned up by the reaper thread. Thread-local cache should be used with caution as cached connections, while not available to other threads, are still take the place in the pool, so can cause a "pool startvation".

Configuration Options

The following global configuration options can be used to configure the connection pool. These options are only effective for the databases attached after the option is set. To change the settings of a connection pool for a database that is already attached the `postgres_configure_pool` function can be used instead.

- `pg_pool_acquire_mode` (VARCHAR, default: `'force'`): how to acquire connections from the pool: `'force'` (always connect, ignore pool limit), `'wait'` (block until available), `'try'` (fail immediately if unavailable)
- `pg_pool_max_connections` (UBIGINT, default: `4 <= cpu_count * 1.5 <= 32`): maximum number of connections that are allowed to be cached in a connection pool for each attached Postgres database. This number can be temporary exceeded when parallel scans are used.
- `pg_pool_wait_timeout_millis` (UBIGINT, default: `30000`): maximum number of milliseconds to wait when acquiring a connection from a pool where all available connections are already taken.
- `pg_pool_enable_thread_local_cache` (BOOLEAN, default: `FALSE`): whether to enable the connection caching in thread-local cache. Such connections are getting pinned to the threads and are not made available to other threads, while still taking the place in the pool.
- `pg_pool_max_lifetime_millis` (UBIGINT, default: `0` not enforced): maximum number of milliseconds the connection can be kept open. This number is checked when the connection is taken from the pool and returned to the pool. When the connection pool reaper thread is enabled (`pg_pool_enable_reaper_thread` option), then this number is checked in background periodically.
- `pg_pool_idle_timeout_millis` (UBIGINT, default: `60000`): maximum number of milliseconds the connection can be kept idle in the pool. This number is checked when the connection is taken from the pool. When the connection pool reaper thread is enabled (`pg_pool_enable_reaper_thread` option), then this number is checked in background periodically.
- `pg_pool_enable_reaper_thread` (default: `TRUE`): whether to enable the connection pool reaper thread, that periodically scans the pool to check the `pg_pool_max_lifetime_millis` and `pg_pool_idle_timeout_millis` and closes the connection which exceed the specified values. Either `pg_pool_max_lifetime_millis` or `pg_pool_idle_timeout_millis` must be set to a non-zero value for this option to be effective.
- `pg_pool_health_check_query` (VARCHAR, default: `SELECT 1`): the query that is used to check that the connection is healthy. Setting this option to an empty string disables the health check.

PostgreSQL Extension Functions

- `postgres_clear_cache`
- `postgres_configure_pool`
- `postgres_execute`
- `postgres_hstore_get`
- `postgres_hstore_to_json`
- `postgres_query`
- `read_postgres_binary`

postgres_clear_cache

```
postgres_clear_cache() -> TABLE
```

Clears cached schema entries (like table names with column lists) for all attached PostgreSQL catalogs. Attached schema is going to be re-read on the next access.

Parameters

None.

Returns

A table with the following columns:

- `Success` (`BOOLEAN`): a flag whether the cache clearing was successful

> Currently the table result is always returned with zero rows, so the flag value is not available.

Example

```
CALL postgres_clear_cache()
```

postgres_configure_pool

```
FROM postgres_configure_pool([<optional named parameters>]) -> TABLE
```

When a PostgreSQL database is attached, a connection pool is created for this database. This function allows to change the configuration options of a connection pool for the specified attached database. It also allows to list the current effective configuration options and the collected runtime statistics of a connection pool.

Parameters

- `catalog_name` (`VARCHAR`): the name (alias) of the attached Postgres database to which pool the configuration change is applied and details are returned. When `NULL` (default) returns the current state of pools for all attached catalogs without changing their configuration. Must be specified and non-NULL when any other option is specified.
- `acquire_mode` (`VARCHAR`, default: `'force'`): how to acquire connections from the pool: `'force'` (always connect, ignore pool limit), `'wait'` (block until available), `'try'` (fail immediately if unavailable)
- `max_connections` (`UBIGINT`): maximum number of connections that are allowed to be cached in a connection pool for each attached Postgres database. This number can be temporary exceeded when parallel scans are used.
- `wait_timeout_millis` (`UBIGINT`): maximum number of milliseconds to wait when acquiring a connection from a pool where all available connections are already taken.
- `enable_thread_local_cache` (`BOOLEAN`): whether to enable the connection caching in thread-local cache. Such connections are getting pinned to the threads and are not made available to other threads, while still taking the place in the pool.
- `max_lifetime_millis` (`UBIGINT`): maximum number of milliseconds the connection can be kept open. This number is checked when the connection is taken from the pool and returned to the pool. When the connection pool reaper thread is enabled (`enable_reaper_thread` argument), then this number is checked in background periodically.
- `idle_timeout_millis` (`UBIGINT`): maximum number of milliseconds the connection can be kept idle in the pool. This number is checked when the connection is taken from the pool. When the connection pool reaper thread is enabled (`enable_reaper_thread` option), then this number is checked in background periodically.
- `enable_reaper_thread` (`BOOLEAN`): whether to enable the connection pool reaper thread, that periodically scans the pool to check the `max_lifetime_millis` and `idle_timeout_millis` and closes the connection which exceed the specified values. Either `max_lifetime_millis` or `idle_timeout_millis` must be set to a non-zero value for this option to be effective.

- `health_check_query` (VARCHAR): the query that is used to check that the connection is healthy. Setting this option to an empty string disables the health check.

Returns

A table with the following columns:

- `catalog_name` (VARCHAR): the name (alias) of the attached Postgres database
- `acquire_mode` (VARCHAR): how to acquire connections from the pool: `'force'` (always connect, ignore pool limit), `'wait'` (block until available), `'try'` (fail immediately if unavailable)
- `available_connections` (UBIGINT): the number of idle connection that are currently available in the pool
- `max_connections` (UBIGINT): maximum number of connections that are allowed to be cached in the pool.
- `wait_timeout_millis` (UBIGINT): maximum number of milliseconds to wait when acquiring a connection from a pool where all available connections are already taken; only applicable to the `wait` acquire-mode
- `cache_hits` (UBIGINT): number of times a cached connection was successfully returned from the pool
- `cache_misses` (UBIGINT): number of times a new connection was created by the pool
- `try_failures` (UBIGINT): number of times a connection was requested by the pool in `try` acquire-mode and no connection was available at the time - so no connection was provided by the pool; note that when the parallel scans are performed by the Postgres extension - worker threads always use the `try` acquire mode; when `SET threads = <number_higher_then_pool_size>{:.language-sql .highlight}` is used, then some of the worker threads are unable to obtain a connection - they return without doing any work and without throwing an error
- `thread_local_cache_enabled` (BOOLEAN): whether caching the connections caching in thread-local cache is enabled; thread-local connections are **NOT** cleared by the reaper thread
- `thread_local_cache_hits` (UBIGINT): the number of times connections were successfully acquired from a thread-local cache without going to the main pool
- `thread_local_cache_misses` (UBIGINT): the number of times connections were not available in a thread-local cache and were taken from the main pool instead
- `max_lifetime_millis` (UBIGINT): maximum number of milliseconds a connection can be kept open
- `idle_timeout_millis` (UBIGINT): maximum number of milliseconds a connection can be kept idle in the pool
- `reaper_thread_running` (BOOLEAN): whether the pool reaper thread is running; this thread periodically scans the pool to check the `max_lifetime_millis` and `idle_timeout_millis` and closes connections which exceed the specified values
- `reaper_thread_period_millis` (UBIGINT): the period of time when a reaper thread performs the checks
- `health_check_query` (VARCHAR): the query that is used to check that the connection is healthy

Examples

List the current effective configuration options and the collected runtime statistics of all connections pools (for all attached databases):

```
FROM postgres_configure_pool()
```

Change one or more configuration option for the connection pool of the specified attached database:

```
FROM postgres_configure_pool(catalog_name = 'db1', acquire_mode = 'wait', max_connections = 42)
```

postgres_execute

```
postgres_execute(attached_db_name VARCHAR, sql_query VARCHAR[, <optional named parameters>]) -> TABLE
```

Executes the `<sql_query>{:.language-sql .highlight}` in the specified remote Postgres instance previously attached with `ATTACH ... AS <attached_db_name>{:.language-sql .highlight}`. This function returns an empty result.

Parameters

- `attached_db_name` (VARCHAR): the name of the attached PostgreSQL database
- `sql_query` (VARCHAR): query that is passed to PostgreSQL for execution; no transformation or analysis of this query is performed by DuckDB

Optional named parameters:

- `use_transaction` (BOOLEAN, default: TRUE): whether to start a PostgreSQL transaction if it was not started before.

Returns

A table with the following columns:

- `Success` (BOOLEAN): a flag whether the cache clearing was successful

> Currently the table result is always returned with zero rows, so the flag value is not available.

Example

```
CALL postgres_execute('db1', 'VACUUM ANALYZE', use_transaction = false)
```

postgres_hstore_get

```
postgres_hstore_get(hstore_string VARCHAR, hstore_key VARCHAR) -> VARCHAR
```

Parses the external representation of and PostgreSQL hstore column and returns the value of the specified hstore key.

Parameters

- `hstore_string` (VARCHAR): PostgreSQL `hstore` value in `key => value` form.
- `hstore_kay` (VARCHAR): Key name to return the value for.

Returns

The value for the specified key. `NULL` if the key not found.

Example

```
SELECT postgres_hstore_get('a=>b, c=>d', 'a')
```

postgres_hstore_to_json

```
postgres_hstore_to_json(hstore_string VARCHAR) -> JSON
```

Converts the external representation of and PostgreSQL hstore column into `JSON`.

Parameters

- `hstore_string` (VARCHAR): PostgreSQL `hstore` value in `key => value` form.

Returns

JSON dictionary with the same key/value pairs as the input `hstore` string. All values are returned as strings.

Example

```
SELECT postgres_hstore_to_json('z=>1, a=>2, m=>3')
```

postgres_query

```
postgres_query(attached_db_name VARCHAR, sql_query VARCHAR[, <optional named parameters>]) -> TABLE
```

Executes the query in the specified remote DB that was previously attached with `ATTACH .. AS <attached_db_name>{:.language-sql .highlight}` and returns the query result as a table.

Parameters

- `attached_db_name` (VARCHAR): the name of the attached PostgreSQL database
- `sql_query` (VARCHAR): query that is passed to PostgreSQL for execution; no transformation or analysis of this query is performed by DuckDB

Optional named parameters:

- `use_transaction` (BOOLEAN, default: TRUE): whether to start a PostgreSQL transaction if it was not started before.
- `params` (STRUCT): query parameters that are passed to the PostgreSQL server; only supported when the text protocol is used

Returns

A table with the query result.

Example

```
FROM postgres_query('db11', 'SELECT $1::INTEGER, $2::TEXT', params=row(42::INTEGER, 'foo'::VARCHAR))
```

read_postgres_binary

```
FROM read_postgres_binary(file_path VARCHAR[, <optional named parameters>]) -> TABLE
```

Reads PostgreSQL binary dump files from file system.

Parameters

- `file_path` (VARCHAR): FS path to the binary dump file in a PostgreSQL format.

Optional named parameters:

- `columns` (STRUCT): type mapping in form of the `column_name -> column_type` structure.
- `buffer_size` (UBIGINT, default: 32KB): size of the read buffer in bytes.

Returns

Contents of the binary dump file as a table.

Example

```
COPY (SELECT 42::INTEGER AS a, 'foo'::VARCHAR AS b) TO 'path/to/test.bin' (FORMAT postgres_binary);

FROM read_postgres_binary('path/to/test.bin', columns = {a: 'INTEGER', b: 'VARCHAR'});
```

Sirius Extension

> The `sirius` extension is not yet available. It is coming soon.

The `sirius` extension adds GPU-accelerated query execution to DuckDB. Sirius is a GPU-native SQL engine that runs DuckDB queries on NVIDIA GPUs.

Spatial

Spatial Extension

The `spatial` extension provides support for geospatial data processing in DuckDB. For an overview of the extension, see our blog post.

Installing and Loading

To install the `spatial` extension, run:

```
INSTALL spatial;
```

Note that the `spatial` extension is not autoloadable. Therefore, you need to load it before using it:

```
LOAD spatial;
```

Spatial Functions

Function Index

Scalar Functions

Function	Summary
DuckDB_PROJ_Compiled_Version	Returns a text description of the PROJ library version that this instance of DuckDB was compiled against.
DuckDB_Proj_Version	Returns a text description of the PROJ library version that is being used by this instance of DuckDB.
ST_Affine	Applies an affine transformation to a geometry.
ST_Area	Compute the area of a geometry.
ST_Area_Spheroid	Returns the area of a geometry in meters, using an ellipsoidal model of the earth
ST_AsGeoJSON	Returns the geometry as a GeoJSON fragment
ST_AsHEXWKB	Returns the geometry as a HEXWKB string
ST_AsMVTGeom	Transform and clip geometry to a tile boundary
ST_AsSVG	Convert the geometry into a SVG fragment or path
ST_AsText	Returns the geometry as a WKT string
ST_AsWKB	Returns the geometry as a WKB (Well-Known-Binary) blob
ST_Azimuth	Returns the azimuth (a clockwise angle measured from north) of two points in radian.
ST_Boundary	Returns the ”boundary” of a geometry
ST_Buffer	Returns a buffer around the input geometry at the target distance

Function	Summary
ST_BuildArea	Creates a polygonal geometry by attempting to ”fill in” the input geometry.
ST_Centroid	Returns the centroid of a geometry
ST_Collect	Collects a list of geometries into a collection geometry.
ST_CollectionExtract	Extracts geometries from a GeometryCollection into a typed multi geometry.
ST_ConcaveHull	Returns the 'concave' hull of the input geometry, containing all of the source input's points, and which can be used to create polygons from points. The ratio parameter dictates the level of concavity; 1.0 returns the convex hull; and 0 indicates to return the most concave hull possible. Set allowHoles to a non-zero value to allow output containing holes.
ST_Contains	Returns true if the first geometry contains the second geometry
ST_ContainsProperly	Returns true if the first geometry ”properly” contains the second geometry
ST_ConvexHull	Returns the convex hull enclosing the geometry
ST_CoverageInvalidEdges	Returns the invalid edges in a polygonal coverage, which are edges that are not shared by two polygons.
ST_CoverageSimplify	Simplify the edges in a polygonal coverage, preserving the coverage by ensuring that there are no seams between the resulting simplified polygons.
ST_CoverageUnion	Union all geometries in a polygonal coverage into a single geometry.
ST_CoveredBy	Returns true if geom1 is ”covered by” geom2
ST_Covers	Returns true if the geom1 ”covers” geom2
ST_Crosses	Returns true if geom1 ”crosses” geom2
ST_DWithin	Returns if two geometries are within a target distance of each-other
ST_DWithin_GEOS	Returns if two geometries are within a target distance of each-other
ST_DWithin_Spheroid	Returns if two POINT_2D's are within a target distance in meters, using an ellipsoidal model of the earths surface
ST_Difference	Returns the ”difference” between two geometries
ST_Dimension	Returns the ”topological dimension” of a geometry.
ST_Disjoint	Returns true if the geometries are disjoint
ST_Distance	Returns the planar distance between two geometries
ST_Distance_GEOS	Returns the planar distance between two geometries
ST_Distance_Sphere	Returns the haversine (great circle) distance between two geometries.
ST_Distance_Spheroid	Returns the distance between two geometries in meters using an ellipsoidal model of the earths surface
ST_Dump	Dumps a geometry into a list of sub-geometries and their ”path” in the original geometry.
ST_EndPoint	Returns the end point of a LINESTRING.

Function	Summary
ST_Envelope	Returns the minimum bounding rectangle of a geometry as a polygon geometry
ST_Equals	Returns true if the geometries are ”equal”
ST_Extent	Returns the minimal bounding box enclosing the input geometry
ST_Extent_Approx	Returns the approximate bounding box of a geometry, if available.
ST_ExteriorRing	Returns the exterior ring (shell) of a polygon geometry.
ST_FlipCoordinates	Returns a new geometry with the coordinates of the input geometry ”flipped” so that x = y and y = x
ST_Force2D	Forces the vertices of a geometry to have X and Y components
ST_Force3DM	Forces the vertices of a geometry to have X, Y and M components
ST_Force3DZ	Forces the vertices of a geometry to have X, Y and Z components
ST_Force4D	Forces the vertices of a geometry to have X, Y, Z and M components
ST_GeomFromGeoJSON	Deserializes a GEOMETRY from a GeoJSON fragment.
ST_GeomFromHEXEWKB	Deserialize a GEOMETRY from a HEX(E)WKB encoded string
ST_GeomFromHEXWKB	Deserialize a GEOMETRY from a HEX(E)WKB encoded string
ST_GeomFromText	Deserialize a GEOMETRY from a WKT encoded string
ST_GeomFromWKB	Deserializes a GEOMETRY from a WKB encoded blob
ST_GeometryType	Returns a GEOMETRY_TYPE enum identifying the input geometry type. Possible enum return types are: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON and GEOMETRYCOLLECTION.
ST_HasM	Check if the input geometry has M values.
ST_HasZ	Check if the input geometry has Z values.
ST_Hilbert	Encodes the X and Y values as the hilbert curve index for a curve covering the given bounding box.
ST_InterpolatePoint	Computes the closest point on a LINESTRING to a given POINT and returns the interpolated M value of that point.
ST_Intersection	Returns the intersection of two geometries
ST_Intersects	Returns true if the geometries intersect
ST_Intersects_Extent	Returns true if the extent of two geometries intersects
ST_IsClosed	Check if a geometry is 'closed'
ST_IsEmpty	Returns true if the geometry is ”empty”.
ST_IsRing	Returns true if the geometry is a ring (both ST_IsClosed and ST_IsSimple).
ST_IsSimple	Returns true if the geometry is simple
ST_IsValid	Returns true if the geometry is valid
ST_Length	Returns the length of the input line geometry
ST_Length_Spheroid	Returns the length of the input geometry in meters, using an ellipsoidal model of the earth
ST_LineInterpolatePoint	Returns a point interpolated along a line at a fraction of total 2D length.

Function	Summary
ST_LineInterpolatePoints	Returns a multi-point interpolated along a line at a fraction of total 2D length.
ST_LineLocatePoint	Returns the location on a line closest to a point as a fraction of the total 2D length of the line.
ST_LineMerge	”Merges” the input line geometry, optionally taking direction into account.
ST_LineString2DFromWKB	Deserialize a LINESTRING_2D from a WKB encoded blob
ST_LineSubstring	Returns a substring of a line between two fractions of total 2D length.
ST_LocateAlong	Returns a point or multi-point, containing the point(s) at the geometry with the given measure
ST_LocateBetween	Returns a geometry or geometry collection created by filtering and interpolating vertices within a range of ”M” values
ST_M	Returns the M coordinate of a point geometry
ST_MMax	Returns the maximum M coordinate of a geometry
ST_MMin	Returns the minimum M coordinate of a geometry
ST_MakeBox2D	Create a BOX2D from two POINT geometries
ST_MakeEnvelope	Create a rectangular polygon from min/max coordinates
ST_MakeLine	Create a LINESTRING from a list of POINT geometries
ST_MakePoint	Creates a GEOMETRY point from an pair of floating point numbers.
ST_MakePolygon	Create a POLYGON from a LINESTRING shell
ST_MakeValid	Returns a valid representation of the geometry
ST_MaximumInscribedCircle	Returns the maximum inscribed circle of the input geometry, optionally with a tolerance.
ST_MinimumRotatedRectangle	Returns the minimum rotated rectangle that bounds the input geometry, finding the surrounding box that has the lowest area by using a rotated rectangle, rather than taking the lowest and highest coordinate values as per ST_Envelope().
ST_Multi	Turns a single geometry into a multi geometry.
ST_NGeometries	Returns the number of component geometries in a collection geometry.
ST_NInteriorRings	Returns the number of interior rings of a polygon
ST_NPoints	Returns the number of vertices within a geometry
ST_Node	Returns a ”noded” MultiLinestring, produced by combining a collection of input linestrings and adding additional vertices where they intersect.
ST_Normalize	Returns the ”normalized” representation of the geometry
ST_NumGeometries	Returns the number of component geometries in a collection geometry.
ST_NumInteriorRings	Returns the number of interior rings of a polygon
ST_NumPoints	Returns the number of vertices within a geometry
ST_Overlaps	Returns true if the geometries overlap

Function	Summary
ST_Perimeter	Returns the length of the perimeter of the geometry
ST_Perimeter_Spheroid	Returns the length of the perimeter in meters using an ellipsoidal model of the earths surface
ST_Point	Creates a GEOMETRY point
ST_Point2D	Creates a POINT_2D
ST_Point2DFromWKB	Deserialize a POINT_2D from a WKB encoded blob
ST_Point3D	Creates a POINT_3D
ST_Point4D	Creates a POINT_4D
ST_PointN	Returns the n'th vertex from the input geometry as a point geometry
ST_PointOnSurface	Returns a point guaranteed to lie on the surface of the geometry
ST_Points	Collects all the vertices in the geometry into a MULTIPOINT
ST_Polygon2DFromWKB	Deserialize a POLYGON_2D from a WKB encoded blob
ST_Polygonize	Returns a polygonized representation of the input geometries
ST_QuadKey	Compute the quadkey for a given lon/lat point at a given level.
ST_ReducePrecision	Returns the geometry with all vertices reduced to the given precision
ST_RemoveRepeatedPoints	Remove repeated points from a LINESTRING.
ST_Reverse	Returns the geometry with the order of its vertices reversed
ST_ShortestLine	Returns the shortest line between two geometries
ST_Simplify	Returns a simplified version of the geometry
ST_SimplifyPreserveTopology	Returns a simplified version of the geometry that preserves topology
ST_StartPoint	Returns the start point of a LINESTRING.
ST_TileEnvelope	The `ST_TileEnvelope` scalar function generates tile envelope rectangular polygons from specified zoom level and tile indices.
ST_Touches	Returns true if the geometries touch
ST_Transform	Transforms a geometry between two coordinate systems
ST_Union	Returns the union of two geometries
ST_VoronoiDiagram	Returns the Voronoi diagram of the supplied MultiPoint geometry
ST_Within	Returns true if the first geometry is within the second
ST_WithinProperly	Returns true if the first geometry ”properly” is contained by the second geometry
ST_X	Returns the X coordinate of a point geometry
ST_XMax	Returns the maximum X coordinate of a geometry
ST_XMin	Returns the minimum X coordinate of a geometry
ST_Y	Returns the Y coordinate of a point geometry
ST_YMax	Returns the maximum Y coordinate of a geometry
ST_YMin	Returns the minimum Y coordinate of a geometry
ST_Z	Returns the Z coordinate of a point geometry

Function	Summary
ST_ZMFlag	Returns a flag indicating the presence of Z and M values in the input geometry.
ST_ZMax	Returns the maximum Z coordinate of a geometry
ST_ZMin	Returns the minimum Z coordinate of a geometry

Aggregate Functions

Function	Summary
ST_AsMVT	Make a Mapbox Vector Tile from a set of geometries and properties
ST_CoverageInvalidEdges_Agg	Returns the invalid edges of a coverage geometry
ST_CoverageSimplify_Agg	Simplifies a set of geometries while maintaining coverage
ST_CoverageUnion_Agg	Unions a set of geometries while maintaining coverage
ST_Envelope_Agg	Alias for ST_Extent_Agg.
ST_Extent_Agg	Computes the minimal-bounding-box polygon containing the set of input geometries
ST_Intersection_Agg	Computes the intersection of a set of geometries
ST_MemUnion_Agg	Computes the union of a set of input geometries.
ST_Union_Agg	Computes the union of a set of input geometries

Macro Functions

Function	Summary
ST_Rotate	Alias of ST_RotateZ
ST_RotateX	Rotates a geometry around the X axis. This is a shorthand macro for calling ST_Affine.
ST_RotateY	Rotates a geometry around the Y axis. This is a shorthand macro for calling ST_Affine.
ST_RotateZ	Rotates a geometry around the Z axis. This is a shorthand macro for calling ST_Affine.
ST_Scale	
ST_TransScale	Translates and then scales a geometry in X and Y direction. This is a shorthand macro for calling ST_Affine.
ST_Translate	

Table Functions

Function	Summary
ST_Drivers	Returns the list of supported GDAL drivers and file formats
ST_GeneratePoints	Generates a set of random points within the specified bounding box.

Function	Summary
ST_Read	Read and import a variety of geospatial file formats using the GDAL library.
ST_ReadOSM	The ST_ReadOsm() table function enables reading compressed OpenStreetMap data directly from a .osm.pbf file.
ST_ReadSHP	Read a Shapefile without relying on the GDAL library
ST_Read_Meta	Read the metadata from a variety of geospatial file formats using the GDAL library.

Scalar Functions

DuckDB_PROJ_Compiled_Version

Signature

```
VARCHAR DuckDB_PROJ_Compiled_Version ()
```

Description Returns a text description of the PROJ library version that this instance of DuckDB was compiled against.

Example

```
SELECT duckdb_proj_compiled_version();
```

duckdb_proj_compiled_version()
varchar
Rel. 9.1.1, December 1st, 2022

DuckDB_Proj_Version

Signature

```
VARCHAR DuckDB_Proj_Version ()
```

Description Returns a text description of the PROJ library version that is being used by this instance of DuckDB.

Example

```
SELECT duckdb_proj_version();
```

duckdb_proj_version()
varchar
9.1.1

ST_Affine

Signatures

```
GEOMETRY ST_Affine (geom GEOMETRY, a DOUBLE, b DOUBLE, c DOUBLE, d DOUBLE, e DOUBLE, f DOUBLE, g DOUBLE, h
DOUBLE, i DOUBLE, xoff DOUBLE, yoff DOUBLE, zoff DOUBLE)
GEOMETRY ST_Affine (geom GEOMETRY, a DOUBLE, b DOUBLE, d DOUBLE, e DOUBLE, xoff DOUBLE, yoff DOUBLE)
```

Description Applies an affine transformation to a geometry.

For the 2D variant, the transformation matrix is defined as follows:

```
| a b xoff |
| d e yoff |
| 0 0 1    |
```

For the 3D variant, the transformation matrix is defined as follows:

```
| a b c xoff |
| d e f yoff |
| g h i zoff |
| 0 0 0 1    |
```

The transformation is applied to all vertices of the geometry.

Example

```
-- Translate a point by (2, 3)
SELECT ST_Affine(ST_Point(1, 1),
                 1, 0,   -- a, b
                 0, 1,   -- d, e
                 2, 3);  -- xoff, yoff
----
POINT (3 4)

-- Scale a geometry by factor 2 in X and Y
SELECT ST_Affine(ST_Point(1, 1),
                 2, 0, 0,   -- a, b, c
                 0, 2, 0,   -- d, e, f
                 0, 0, 1,   -- g, h, i
                 0, 0, 0);  -- xoff, yoff, zoff
----
POINT (2 2)
```

ST_Area

Signatures

```
DOUBLE ST_Area (geom GEOMETRY)
DOUBLE ST_Area (polygon POLYGON_2D)
DOUBLE ST_Area (linestring LINESTRING_2D)
DOUBLE ST_Area (point POINT_2D)
DOUBLE ST_Area (box BOX_2D)
```

Description Compute the area of a geometry.

Returns `0.0` for any geometry that is not a `POLYGON`, `MULTIPOLYGON` or `GEOMETRYCOLLECTION` containing polygon geometries.

The area is in the same units as the spatial reference system of the geometry.

The `POINT_2D` and `LINESTRING_2D` overloads of this function always return `0.0` but are included for completeness.

Example

```
select ST_Area('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
-- 1.0
```

ST_Area_Spheroid

Signatures

```
DOUBLE ST_Area_Spheroid (geom GEOMETRY)
DOUBLE ST_Area_Spheroid (poly POLYGON_2D)
```

Description Returns the area of a geometry in meters, using an ellipsoidal model of the earth

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the area is returned in square meters. This function uses the GeographicLib library, calculating the area using an ellipsoidal model of the earth. This is a highly accurate method for calculating the area of a polygon taking the curvature of the earth into account, but is also the slowest.

Returns `0.0` for any geometry that is not a `POLYGON`, `MULTIPOLYGON` or `GEOMETRYCOLLECTION` containing polygon geometries.

ST_AsGeoJSON

Signature

```
JSON ST_AsGeoJSON (geom GEOMETRY)
```

Description Returns the geometry as a GeoJSON fragment

This does not return a complete GeoJSON document, only the geometry fragment. To construct a complete GeoJSON document or feature, look into using the DuckDB JSON extension in conjunction with this function. This function supports geometries with Z values, but not M values. M values are ignored.

Example

```
select ST_AsGeoJSON('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
----
{"type":"Polygon","coordinates":[[[0.0,0.0],[0.0,1.0],[1.0,1.0],[1.0,0.0],[0.0,0.0]]]}

-- Convert a geometry into a full GeoJSON feature (requires the JSON extension to be loaded)
SELECT CAST({
    type: 'Feature',
    geometry: ST_AsGeoJSON(ST_Point(1,2)),
    properties: {
        name: 'my_point'
    }
} AS JSON);
----
{"type":"Feature","geometry":{"type":"Point","coordinates":[1.0,2.0]},"properties":{"name":"my_point"}}
```

ST_AsHEXWKB

Signature

```
VARCHAR ST_AsHEXWKB (geom GEOMETRY)
```

Description Returns the geometry as a HEXWKB string

Example

```
SELECT ST_AsHexWKB('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
----
0103000000010000000500000000000000000000000000000000...
```

ST_AsMVTGeom

Signatures

```
GEOMETRY ST_AsMVTGeom (geom GEOMETRY, bounds BOX_2D, extent BIGINT, buffer BIGINT, clip_geom BOOLEAN)
GEOMETRY ST_AsMVTGeom (geom GEOMETRY, bounds BOX_2D, extent BIGINT, buffer BIGINT)
GEOMETRY ST_AsMVTGeom (geom GEOMETRY, bounds BOX_2D, extent BIGINT)
GEOMETRY ST_AsMVTGeom (geom GEOMETRY, bounds BOX_2D)
```

Description Transform and clip geometry to a tile boundary

See ST_AsMVT for more details

ST_AsSVG

Signature

```
VARCHAR ST_AsSVG (geom GEOMETRY, relative BOOLEAN, precision INTEGER)
```

Description Convert the geometry into a SVG fragment or path

The SVG fragment is returned as a string. The fragment is a path element that can be used in an SVG document. The second boolean argument specifies whether the path should be relative or absolute. The third argument specifies the maximum number of digits to use for the coordinates.

Points are formatted as cx/cy using absolute coordinates or x/y using relative coordinates.

Example

```
SELECT ST_AsSVG('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::GEOMETRY, false, 15);
----
M 0 0 L 0 -1 1 -1 1 0 Z
```

ST_AsText

Signatures

```
VARCHAR ST_AsText (geom GEOMETRY)
VARCHAR ST_AsText (point POINT_2D)
VARCHAR ST_AsText (linestring LINESTRING_2D)
VARCHAR ST_AsText (polygon POLYGON_2D)
VARCHAR ST_AsText (box BOX_2D)
```

Description Returns the geometry as a WKT string

Example

```
SELECT ST_MakeEnvelope(0,0,1,1);
----
POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))
```

ST_AsWKB

Signature

```
WKB_BLOB ST_AsWKB (geom GEOMETRY)
```

Description Returns the geometry as a WKB (Well-Known-Binary) blob

Example

```
SELECT ST_AsWKB('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::GEOMETRY)::BLOB;
----
\x01\x03\x00\x00\x00\x01\x00\x00\x00\x05...
```

ST_Azimuth

Signatures

```
DOUBLE ST_Azimuth (origin GEOMETRY, target GEOMETRY)
DOUBLE ST_Azimuth (origin POINT_2D, target POINT_2D)
```

Description Returns the azimuth (a clockwise angle measured from north) of two points in radian.

Example

```
SELECT degrees(ST_Azimuth(ST_Point(0, 0), ST_Point(0, 1)));
----
90.0
```

ST_Boundary

Signature

```
GEOMETRY ST_Boundary (geom GEOMETRY)
```

Description Returns the ”boundary” of a geometry

ST_Buffer

Signatures

```
GEOMETRY ST_Buffer (geom GEOMETRY, distance DOUBLE)
GEOMETRY ST_Buffer (geom GEOMETRY, distance DOUBLE, num_triangles INTEGER)
GEOMETRY ST_Buffer (geom GEOMETRY, distance DOUBLE, num_triangles INTEGER, cap_style VARCHAR, join_style
VARCHAR, mitre_limit DOUBLE)
```

Description Returns a buffer around the input geometry at the target distance

`geom` is the input geometry.

`distance` is the target distance for the buffer, using the same units as the input geometry.

`num_triangles` represents how many triangles that will be produced to approximate a quarter circle. The larger the number, the smoother the resulting geometry. The default value is 8.

`cap_style` must be one of `'CAP_ROUND'`, `'CAP_FLAT'`, `'CAP_SQUARE'`. This parameter is case-insensitive.

`join_style` must be one of `'JOIN_ROUND'`, `'JOIN_MITRE'`, `'JOIN_BEVEL'`. This parameter is case-insensitive.

`mitre_limit` only applies when `join_style` is `'JOIN_MITRE'`. It is the ratio of the distance from the corner to the mitre point to the corner radius. The default value is 1.0.

This is a planar operation and will not take into account the curvature of the earth.

ST_BuildArea

Signature

```
GEOMETRY ST_BuildArea (geom GEOMETRY)
```

Description Creates a polygonal geometry by attempting to ”fill in” the input geometry.

Unlike ST_Polygonize, this function does not fill in holes.

ST_Centroid

Signatures

```
GEOMETRY ST_Centroid (geom GEOMETRY)
POINT_2D ST_Centroid (point POINT_2D)
POINT_2D ST_Centroid (linestring LINESTRING_2D)
POINT_2D ST_Centroid (polygon POLYGON_2D)
POINT_2D ST_Centroid (box BOX_2D)
POINT_2D ST_Centroid (box BOX_2DF)
```

Description Returns the centroid of a geometry

ST_Collect

Signature

```
GEOMETRY ST_Collect (geoms GEOMETRY[])
```

Description Collects a list of geometries into a collection geometry.

- If all geometries are `POINT`'s, a `MULTIPOINT` is returned.
- If all geometries are `LINESTRING`'s, a `MULTILINESTRING` is returned.
- If all geometries are `POLYGON`'s, a `MULTIPOLYGON` is returned.
- Otherwise if the input collection contains a mix of geometry types, a `GEOMETRYCOLLECTION` is returned.

Empty and `NULL` geometries are ignored. If all geometries are empty or `NULL`, a `GEOMETRYCOLLECTION EMPTY` is returned.

Example

```
-- With all POINT's, a MULTIPOINT is returned
SELECT ST_Collect([ST_Point(1, 2), ST_Point(3, 4)]);
----
MULTIPOINT (1 2, 3 4)

-- With mixed geometry types, a GEOMETRYCOLLECTION is returned
SELECT ST_Collect([ST_Point(1, 2), ST_GeomFromText('LINESTRING(3 4, 5 6)')]);
----
GEOMETRYCOLLECTION (POINT (1 2), LINESTRING (3 4, 5 6))

-- Note that the empty geometry is ignored, so the result is a MULTIPOINT
SELECT ST_Collect([ST_Point(1, 2), NULL, ST_GeomFromText('GEOMETRYCOLLECTION EMPTY')]);
----
MULTIPOINT (1 2)

-- If all geometries are empty or NULL, a GEOMETRYCOLLECTION EMPTY is returned
SELECT ST_Collect([NULL, ST_GeomFromText('GEOMETRYCOLLECTION EMPTY')]);
----
GEOMETRYCOLLECTION EMPTY

-- Tip: You can use the `ST_Collect` function together with the `list()` aggregate function to collect
multiple rows of geometries into a single geometry collection:

CREATE TABLE points (geom GEOMETRY);

INSERT INTO points VALUES (ST_Point(1, 2)), (ST_Point(3, 4));

SELECT ST_Collect(list(geom)) FROM points;
----
MULTIPOINT (1 2, 3 4)
```

ST_CollectionExtract

Signatures

```
GEOMETRY ST_CollectionExtract (geom GEOMETRY, type INTEGER)
GEOMETRY ST_CollectionExtract (geom GEOMETRY)
```

Description Extracts geometries from a GeometryCollection into a typed multi geometry.

If the input geometry is a GeometryCollection, the function will return a multi geometry, determined by the `type` parameter.

- If `type` = 1, returns a MultiPoint containing all the Points in the collection.
- If `type` = 2, returns a MultiLineString containing all the LineStrings in the collection.
- If `type` = 3, returns a MultiPolygon containing all the Polygons in the collection.

If no `type` parameters is provided, the function will return a multi geometry matching the highest "surface dimension" of the contained geometries. E.g. if the collection contains only Points, a MultiPoint will be returned. But if the collection contains both Points and LineStrings, a MultiLineString will be returned. Similarly, if the collection contains Polygons, a MultiPolygon will be returned. Contained geometries of a lower surface dimension will be ignored.

If the input geometry contains nested GeometryCollections, their geometries will be extracted recursively and included into the final multi geometry as well.

If the input geometry is not a GeometryCollection, the function will return the input geometry as is.

Example

```
select st_collectionextract('MULTIPOINT(1 2,3 4)'::geometry, 1);
-- MULTIPOINT (1 2, 3 4)
```

ST_ConcaveHull

Signature

```
GEOMETRY ST_ConcaveHull (geom GEOMETRY, ratio DOUBLE, allowHoles BOOLEAN)
```

Description Returns the 'concave' hull of the input geometry, containing all of the source input's points, and which can be used to create polygons from points. The ratio parameter dictates the level of concavity; 1.0 returns the convex hull; and 0 indicates to return the most concave hull possible. Set allowHoles to a non-zero value to allow output containing holes.

ST_Contains

Signatures

```
BOOLEAN ST_Contains (geom1 POLYGON_2D, geom2 POINT_2D)
BOOLEAN ST_Contains (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the first geometry contains the second geometry

In contrast to `ST_ContainsProperly`, this function will also return true if `geom2` is contained strictly on the boundary of `geom1`. A geometry always `ST_Contains` itself, but does not `ST_ContainsProperly` itself.

ST_ContainsProperly

Signature

```
BOOLEAN ST_ContainsProperly (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the first geometry "properly" contains the second geometry

In contrast to `ST_Contains`, this function does not return true if `geom2` is contained strictly on the boundary of `geom1`. A geometry always `ST_Contains` itself, but does not `ST_ContainsProperly` itself.

ST_ConvexHull

Signature

```
GEOMETRY ST_ConvexHull (geom GEOMETRY)
```

Description Returns the convex hull enclosing the geometry

ST_CoverageInvalidEdges

Signatures

```
GEOMETRY ST_CoverageInvalidEdges (geoms GEOMETRY[], tolerance DOUBLE)
GEOMETRY ST_CoverageInvalidEdges (geoms GEOMETRY[])
```

Description Returns the invalid edges in a polygonal coverage, which are edges that are not shared by two polygons. Returns NULL if the input is not a polygonal coverage, or if the input is valid. Tolerance is 0 by default.

ST_CoverageSimplify

Signatures

```
GEOMETRY ST_CoverageSimplify (geoms GEOMETRY[], tolerance DOUBLE, simplify_boundary BOOLEAN)
GEOMETRY ST_CoverageSimplify (geoms GEOMETRY[], tolerance DOUBLE)
```

Description Simplify the edges in a polygonal coverage, preserving the coverage by ensuring that there are no seams between the resulting simplified polygons.

By default, the boundary of the coverage is also simplified, but this can be controlled with the optional third `simplify_boundary` parameter.

ST_CoverageUnion

Signature

```
GEOMETRY ST_CoverageUnion (geoms GEOMETRY[])
```

Description Union all geometries in a polygonal coverage into a single geometry. This may be faster than using `ST_Union`, but may use more memory.

ST_CoveredBy

Signature

```
BOOLEAN ST_CoveredBy (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if geom1 is ”covered by” geom2

ST_Covers

Signature

```
BOOLEAN ST_Covers (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the geom1 ”covers” geom2

ST_Crosses

Signature

```
BOOLEAN ST_Crosses (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if geom1 ”crosses” geom2

ST_DWithin

Signature

```
BOOLEAN ST_DWithin (geom1 GEOMETRY, geom2 GEOMETRY, distance DOUBLE)
```

Description Returns if two geometries are within a target distance of each-other

---

ST_DWithin_GEOS

Signature

```
BOOLEAN ST_DWithin_GEOS (geom1 GEOMETRY, geom2 GEOMETRY, distance DOUBLE)
```

Description Returns if two geometries are within a target distance of each-other

---

ST_DWithin_Spheroid

Signature

```
BOOLEAN ST_DWithin_Spheroid (p1 POINT_2D, p2 POINT_2D, distance DOUBLE)
```

Description Returns if two POINT_2D's are within a target distance in meters, using an ellipsoidal model of the earths surface

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the distance is returned in meters. This function uses the GeographicLib library to solve the inverse geodesic problem, calculating the distance between two points using an ellipsoidal model of the earth. This is a highly accurate method for calculating the distance between two arbitrary points taking the curvature of the earths surface into account, but is also the slowest.

ST_Difference

Signature

```
GEOMETRY ST_Difference (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns the ”difference” between two geometries

ST_Dimension

Signature

```
INTEGER ST_Dimension (geom GEOMETRY)
```

Description Returns the ”topological dimension” of a geometry.

- For POINT and MULTIPOINT geometries, returns 0.
- For LINESTRING and MULTILINESTRING, returns 1.
- For POLYGON and MULTIPOLYGON, returns 2.
- For GEOMETRYCOLLECTION, returns the maximum dimension of the contained geometries, or 0 if the collection is empty.

Example

```
select st_dimension('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
----
2
```

ST_Disjoint

Signature

```
BOOLEAN ST_Disjoint (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the geometries are disjoint

ST_Distance

Signatures

```
DOUBLE ST_Distance (point1 POINT_2D, point2 POINT_2D)
DOUBLE ST_Distance (point POINT_2D, linestring LINESTRING_2D)
DOUBLE ST_Distance (linestring LINESTRING_2D, point POINT_2D)
DOUBLE ST_Distance (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns the planar distance between two geometries

Example

```
SELECT ST_Distance('POINT (0 0)'::GEOMETRY, 'POINT (3 4)'::GEOMETRY);
----
5.0

-- Z coordinates are ignored
SELECT ST_Distance('POINT Z (0 0 0)'::GEOMETRY, 'POINT Z (3 4 5)'::GEOMETRY);
----
5.0
```

ST_Distance_GEOS

Signature

```
DOUBLE ST_Distance_GEOS (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns the planar distance between two geometries

---

ST_Distance_Sphere

Signatures

```
DOUBLE ST_Distance_Sphere (geom1 GEOMETRY, geom2 GEOMETRY)
DOUBLE ST_Distance_Sphere (point1 POINT_2D, point2 POINT_2D)
```

Description Returns the haversine (great circle) distance between two geometries.

- Only supports POINT geometries.
- Returns the distance in meters.
- The input is expected to be in WGS84 (EPSG:4326) coordinates, using a [latitude, longitude] axis order.

ST_Distance_Spheroid

Signature

```
DOUBLE ST_Distance_Spheroid (p1 POINT_2D, p2 POINT_2D)
```

Description Returns the distance between two geometries in meters using an ellipsoidal model of the earths surface

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the distance limit is expected to be in meters. This function uses the GeographicLib library to solve the inverse geodesic problem, calculating the distance between two points using an ellipsoidal model of the earth. This is a highly accurate method for calculating the distance between two arbitrary points taking the curvature of the earths surface into account, but is also the slowest.

Example

```
-- Note: the coordinates are in WGS84 and [latitude, longitude] axis order
-- What's the distance between New York and Amsterdam (JFK and AMS airport)?
SELECT st_distance_spheroid(
st_point(40.6446, -73.7797),
st_point(52.3130, 4.7725)
);
----
5863418.7459356235
-- Roughly 5863km!
```

ST_Dump

Signature

```
STRUCT(geom GEOMETRY, path INTEGER[])[] ST_Dump (geom GEOMETRY)
```

Description Dumps a geometry into a list of sub-geometries and their ”path” in the original geometry.

You can use the UNNEST(`res, recursive := true`) function to explode resulting list of structs into multiple rows.

Example

```
select st_dump('MULTIPOINT(1 2,3 4)'::geometry);
----
[{'geom': 'POINT(1 2)', 'path': [0]}, {'geom': 'POINT(3 4)', 'path': [1]}]

select unnest(st_dump('MULTIPOINT(1 2,3 4)'::geometry), recursive := true);
--
--   geom      path
--   geometry  int32[]
--
--   POINT (1 2)  [1]
--   POINT (3 4)  [2]
--
```

ST_EndPoint

Signatures

```
GEOMETRY ST_EndPoint (geom GEOMETRY)
POINT_2D ST_EndPoint (line LINESTRING_2D)
```

Description Returns the end point of a LINESTRING.

ST_Envelope

Signature

```
GEOMETRY ST_Envelope (geom GEOMETRY)
```

Description Returns the minimum bounding rectangle of a geometry as a polygon geometry

---

ST_Equals

Signature

```
BOOLEAN ST_Equals (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the geometries are ”equal”

ST_Extent

Signatures

```
BOX_2D ST_Extent (geom GEOMETRY)
BOX_2D ST_Extent (wkb WKB_BLOB)
```

Description Returns the minimal bounding box enclosing the input geometry

---

ST_Extent_Approx

Signature

```
BOX_2DF ST_Extent_Approx (geom GEOMETRY)
```

Description Returns the approximate bounding box of a geometry, if available.

This function is only really used internally, and returns the cached bounding box of the geometry if it exists. This function may be removed or renamed in the future.

ST_ExteriorRing

Signatures

```
GEOMETRY ST_ExteriorRing (geom GEOMETRY)
LINESTRING_2D ST_ExteriorRing (polygon POLYGON_2D)
```

Description Returns the exterior ring (shell) of a polygon geometry.

ST_FlipCoordinates

Signatures

```
GEOMETRY ST_FlipCoordinates (geom GEOMETRY)
POINT_2D ST_FlipCoordinates (point POINT_2D)
LINESTRING_2D ST_FlipCoordinates (linestring LINESTRING_2D)
POLYGON_2D ST_FlipCoordinates (polygon POLYGON_2D)
BOX_2D ST_FlipCoordinates (box BOX_2D)
```

Description Returns a new geometry with the coordinates of the input geometry ”flipped” so that x = y and y = x

---

ST_Force2D

Signature

```
GEOMETRY ST_Force2D (geom GEOMETRY)
```

Description Forces the vertices of a geometry to have X and Y components

This function will drop any Z and M values from the input geometry, if present. If the input geometry is already 2D, it will be returned as is.

ST_Force3DM

Signature

```
GEOMETRY ST_Force3DM (geom GEOMETRY, m DOUBLE)
```

Description Forces the vertices of a geometry to have X, Y and M components

The following cases apply:

- If the input geometry has a Z component but no M component, the Z component will be replaced with the new M value.
- If the input geometry has a M component but no Z component, it will be returned as is.
- If the input geometry has both a Z component and a M component, the Z component will be removed.
- Otherwise, if the input geometry has neither a Z or M component, the new M value will be added to the vertices of the input geometry.

ST_Force3DZ

Signature

```
GEOMETRY ST_Force3DZ (geom GEOMETRY, z DOUBLE)
```

Description Forces the vertices of a geometry to have X, Y and Z components

The following cases apply:

- If the input geometry has a M component but no Z component, the M component will be replaced with the new Z value.
- If the input geometry has a Z component but no M component, it will be returned as is.
- If the input geometry has both a Z component and a M component, the M component will be removed.
- Otherwise, if the input geometry has neither a Z or M component, the new Z value will be added to the vertices of the input geometry.

ST_Force4D

Signature

```
GEOMETRY ST_Force4D (geom GEOMETRY, z DOUBLE, m DOUBLE)
```

Description Forces the vertices of a geometry to have X, Y, Z and M components

The following cases apply:

- If the input geometry has a Z component but no M component, the new M value will be added to the vertices of the input geometry.
- If the input geometry has a M component but no Z component, the new Z value will be added to the vertices of the input geometry.
- If the input geometry has both a Z component and a M component, the geometry will be returned as is.
- Otherwise, if the input geometry has neither a Z or M component, the new Z and M values will be added to the vertices of the input geometry.

ST_GeomFromGeoJSON

Signatures

```
GEOMETRY ST_GeomFromGeoJSON (geojson JSON)
GEOMETRY ST_GeomFromGeoJSON (geojson VARCHAR)
```

Description Deserializes a GEOMETRY from a GeoJSON fragment.

Example

```
SELECT ST_GeomFromGeoJSON('{"type":"Point","coordinates":[1.0,2.0]}');
----
POINT (1 2)
```

ST_GeomFromHEXEWKB

Signature

```
GEOMETRY ST_GeomFromHEXEWKB (hexwkb VARCHAR)
```

Description Deserialize a GEOMETRY from a HEX(E)WKB encoded string

DuckDB spatial doesn't currently differentiate between `WKB` and `EWKB`, so `ST_GeomFromHEXWKB` and `ST_GeomFromHEXEWKB` are just aliases of each other.

ST_GeomFromHEXWKB

Signature

```
GEOMETRY ST_GeomFromHEXWKB (hexwkb VARCHAR)
```

Description Deserialize a GEOMETRY from a HEX(E)WKB encoded string

DuckDB spatial doesn't currently differentiate between `WKB` and `EWKB`, so `ST_GeomFromHEXWKB` and `ST_GeomFromHEXEWKB` are just aliases of each other.

ST_GeomFromText

Signatures

```
GEOMETRY ST_GeomFromText (wkt VARCHAR)
GEOMETRY ST_GeomFromText (wkt VARCHAR, ignore_invalid BOOLEAN)
```

Description Deserialize a GEOMETRY from a WKT encoded string

ST_GeomFromWKB

Signatures

```
GEOMETRY ST_GeomFromWKB (wkb WKB_BLOB)
GEOMETRY ST_GeomFromWKB (blob BLOB)
```

Description Deserializes a GEOMETRY from a WKB encoded blob

ST_GeometryType

Signatures

```
ANY ST_GeometryType (geom GEOMETRY)
ANY ST_GeometryType (point POINT_2D)
ANY ST_GeometryType (linestring LINESTRING_2D)
ANY ST_GeometryType (polygon POLYGON_2D)
ANY ST_GeometryType (wkb WKB_BLOB)
```

Description Returns a `GEOMETRY_TYPE` enum identifying the input geometry type. Possible enum return types are: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` and `GEOMETRYCOLLECTION`.

Example

```
SELECT DISTINCT ST_GeometryType(ST_GeomFromText('POINT(1 1)'));
----
POINT
```

ST_HasM

Signatures

```
BOOLEAN ST_HasM (geom GEOMETRY)
BOOLEAN ST_HasM (wkb WKB_BLOB)
```

Description Check if the input geometry has M values.

Example

```
-- HasM for a 2D geometry
SELECT ST_HasM(ST_GeomFromText('POINT(1 1)'));
----
false

-- HasM for a 3DZ geometry
SELECT ST_HasM(ST_GeomFromText('POINT Z(1 1 1)'));
----
false

-- HasM for a 3DM geometry
SELECT ST_HasM(ST_GeomFromText('POINT M(1 1 1)'));
----
true

-- HasM for a 4D geometry
SELECT ST_HasM(ST_GeomFromText('POINT ZM(1 1 1 1)'));
----
true
```

ST_HasZ

Signatures

```
BOOLEAN ST_HasZ (geom GEOMETRY)
BOOLEAN ST_HasZ (wkb WKB_BLOB)
```

Description Check if the input geometry has Z values.

Example

```
-- HasZ for a 2D geometry
SELECT ST_HasZ(ST_GeomFromText('POINT(1 1)'));
----
false

-- HasZ for a 3DZ geometry
SELECT ST_HasZ(ST_GeomFromText('POINT Z(1 1 1)'));
```

```
----
true

-- HasZ for a 3DM geometry
SELECT ST_HasZ(ST_GeomFromText('POINT M(1 1 1)'));
----
false

-- HasZ for a 4D geometry
SELECT ST_HasZ(ST_GeomFromText('POINT ZM(1 1 1 1)'));
----
true
```

ST_Hilbert

Signatures

```
UINTEGER ST_Hilbert (x DOUBLE, y DOUBLE, bounds BOX_2D)
UINTEGER ST_Hilbert (geom GEOMETRY, bounds BOX_2D)
UINTEGER ST_Hilbert (geom GEOMETRY)
UINTEGER ST_Hilbert (box BOX_2D, bounds BOX_2D)
UINTEGER ST_Hilbert (box BOX_2DF, bounds BOX_2DF)
```

Description Encodes the X and Y values as the hilbert curve index for a curve covering the given bounding box. If a geometry is provided, the center of the approximate bounding box is used as the point to encode. If no bounding box is provided, the hilbert curve index is mapped to the full range of a single-presicion float. For the BOX_2D and BOX_2DF variants, the center of the box is used as the point to encode.

ST_InterpolatePoint

Signature

```
DOUBLE ST_InterpolatePoint (line GEOMETRY, point GEOMETRY)
```

Description Computes the closest point on a LINESTRING to a given POINT and returns the interpolated M value of that point.

First argument must be a linestring and must have a M dimension. The second argument must be a point. Neither argument can be empty.

ST_Intersection

Signature

```
GEOMETRY ST_Intersection (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns the intersection of two geometries

ST_Intersects

Signatures

```
BOOLEAN ST_Intersects (box1 BOX_2D, box2 BOX_2D)
BOOLEAN ST_Intersects (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the geometries intersect

ST_Intersects_Extent

Signature

```
BOOLEAN ST_Intersects_Extent (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the extent of two geometries intersects

---

ST_IsClosed

Signature

```
BOOLEAN ST_IsClosed (geom GEOMETRY)
```

Description Check if a geometry is 'closed'

ST_IsEmpty

Signatures

```
BOOLEAN ST_IsEmpty (geom GEOMETRY)
BOOLEAN ST_IsEmpty (linestring LINESTRING_2D)
BOOLEAN ST_IsEmpty (polygon POLYGON_2D)
```

Description Returns true if the geometry is ”empty”.

ST_IsRing

Signature

```
BOOLEAN ST_IsRing (geom GEOMETRY)
```

Description Returns true if the geometry is a ring (both ST_IsClosed and ST_IsSimple).

ST_IsSimple

Signature

```
BOOLEAN ST_IsSimple (geom GEOMETRY)
```

Description Returns true if the geometry is simple

ST_IsValid

Signature

```
BOOLEAN ST_IsValid (geom GEOMETRY)
```

Description Returns true if the geometry is valid

ST_Length

Signatures

```
DOUBLE ST_Length (geom GEOMETRY)
DOUBLE ST_Length (linestring LINESTRING_2D)
```

Description Returns the length of the input line geometry

ST_Length_Spheroid

Signatures

```
DOUBLE ST_Length_Spheroid (geom GEOMETRY)
DOUBLE ST_Length_Spheroid (line LINESTRING_2D)
```

Description Returns the length of the input geometry in meters, using an ellipsoidal model of the earth

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the length is returned in meters. This function uses the GeographicLib library, calculating the length using an ellipsoidal model of the earth. This is a highly accurate method for calculating the length of a line geometry taking the curvature of the earth into account, but is also the slowest.

Returns `0.0` for any geometry that is not a `LINESTRING`, `MULTILINESTRING` or `GEOMETRYCOLLECTION` containing line geometries.

ST_LineInterpolatePoint

Signature

```
GEOMETRY ST_LineInterpolatePoint (line GEOMETRY, fraction DOUBLE)
```

Description Returns a point interpolated along a line at a fraction of total 2D length.

ST_LineInterpolatePoints

Signature

```
GEOMETRY ST_LineInterpolatePoints (line GEOMETRY, fraction DOUBLE, repeat BOOLEAN)
```

Description Returns a multi-point interpolated along a line at a fraction of total 2D length.

if repeat is false, the result is a single point, (and equivalent to ST_LineInterpolatePoint), otherwise, the result is a multi-point with points repeated at the fraction interval.

ST_LineLocatePoint

Signature

```
DOUBLE ST_LineLocatePoint (line GEOMETRY, point GEOMETRY)
```

Description Returns the location on a line closest to a point as a fraction of the total 2D length of the line.

ST_LineMerge

Signatures

```
GEOMETRY ST_LineMerge (geom GEOMETRY)
GEOMETRY ST_LineMerge (geom GEOMETRY, preserve_direction BOOLEAN)
```

Description ”Merges” the input line geometry, optionally taking direction into account.

ST_LineString2DFromWKB

Signature

```
GEOMETRY ST_LineString2DFromWKB (linestring LINESTRING_2D)
```

Description Deserialize a LINESTRING_2D from a WKB encoded blob

---

ST_LineSubstring

Signature

```
GEOMETRY ST_LineSubstring (line GEOMETRY, start_fraction DOUBLE, end_fraction DOUBLE)
```

Description Returns a substring of a line between two fractions of total 2D length.

ST_LocateAlong

Signatures

```
GEOMETRY ST_LocateAlong (line GEOMETRY, measure DOUBLE, offset DOUBLE)
GEOMETRY ST_LocateAlong (line GEOMETRY, measure DOUBLE)
```

Description Returns a point or multi-point, containing the point(s) at the geometry with the given measure

For a LINESTRING, or MULTILINESTRING, the location is determined by interpolating between M values For a POINT and MULTIPOINT, the point is returned if the measure matches the M value of the vertex, otherwise an empty geometry is returned For a POLYGON, only the exterior ring is considered, and treated as a LINESTRING

If offset is provided, the resulting point(s) is offset by the given amount perpendicular to the line direction.

ST_LocateBetween

Signatures

```
GEOMETRY ST_LocateBetween (line GEOMETRY, start_measure DOUBLE, end_measure DOUBLE, offset DOUBLE)
GEOMETRY ST_LocateBetween (line GEOMETRY, start_measure DOUBLE, end_measure DOUBLE)
```

Description Returns a geometry or geometry collection created by filtering and interpolating vertices within a range of ”M” values

Creates a geometry or geometry collection, containing the parts formed by vertices that have an ”M” value within the `start_measure` and `end_measure` range

For LINESTRING or MULTILINESTRING, if a line segment would cross either the upper or lower bound, a vertex is added by interpolating the coordinates at the ”intersection” For a POINT and MULTIPOINT, the point is added to the collection if its vertex has an ”M” value within the range, otherwise it is skipped For a POLYGON, only the exterior ring is considered, and treated like a LINESTRING

If offset is provided, the resulting vertices are offset by the given amount perpendicular to the line direction.

ST_M

Signature

```
DOUBLE ST_M (geom GEOMETRY)
```

Description Returns the M coordinate of a point geometry

Example

```
SELECT ST_M(ST_Point(1, 2, 3, 4))
```

ST_MMax

Signature

```
DOUBLE ST_MMax (geom GEOMETRY)
```

Description Returns the maximum M coordinate of a geometry

Example

```
SELECT ST_MMax(ST_Point(1, 2, 3, 4))
```

ST_MMin

Signature

```
DOUBLE ST_MMin (geom GEOMETRY)
```

Description Returns the minimum M coordinate of a geometry

Example

```
SELECT ST_MMin(ST_Point(1, 2, 3, 4))
```

ST_MakeBox2D

Signature

```
BOX_2D ST_MakeBox2D (point1 GEOMETRY, point2 GEOMETRY)
```

Description Create a BOX2D from two POINT geometries

Example

```
SELECT ST_MakeBox2D(ST_Point(0, 0), ST_Point(1, 1));
----
BOX(0 0, 1 1)
```

ST_MakeEnvelope

Signature

```
GEOMETRY ST_MakeEnvelope (min_x DOUBLE, min_y DOUBLE, max_x DOUBLE, max_y DOUBLE)
```

Description Create a rectangular polygon from min/max coordinates

---

ST_MakeLine

Signatures

```
GEOMETRY ST_MakeLine (geoms GEOMETRY[])
GEOMETRY ST_MakeLine (start GEOMETRY, end GEOMETRY)
```

Description Create a LINESTRING from a list of POINT geometries

Example

```
SELECT ST_MakeLine([ST_Point(0, 0), ST_Point(1, 1)]);
----
LINESTRING(0 0, 1 1)
```

ST_MakePoint

Signatures

```
POINT_2D ST_MakePoint (x DOUBLE, y DOUBLE)
POINT_3D ST_MakePoint (x DOUBLE, y DOUBLE, z DOUBLE)
POINT_4D ST_MakePoint (x DOUBLE, y DOUBLE, z DOUBLE, m DOUBLE)
```

Description Creates a GEOMETRY point from an pair of floating point numbers.

For geodetic coordinate systems, x is typically the longitude value and y is the latitude value.

Note that ST_Point is equivalent. ST_MakePoint is provided for PostGIS compatibility.

Example

```
SELECT ST_AsText(ST_MakePoint(143.3, -24.2));
----
POINT (143.3 -24.2)
```

ST_MakePolygon

Signatures

```
GEOMETRY ST_MakePolygon (shell GEOMETRY)
GEOMETRY ST_MakePolygon (shell GEOMETRY, holes GEOMETRY[])
```

Description Create a POLYGON from a LINESTRING shell

Example

```
SELECT ST_MakePolygon(ST_LineString([ST_Point(0, 0), ST_Point(1, 0), ST_Point(1, 1), ST_Point(0, 0)]));
```

ST_MakeValid

Signature

```
GEOMETRY ST_MakeValid (geom GEOMETRY)
```

Description Returns a valid representation of the geometry

ST_MaximumInscribedCircle

Signatures

```
STRUCT(center GEOMETRY, nearest GEOMETRY, radius DOUBLE) ST_MaximumInscribedCircle (geom GEOMETRY)
STRUCT(center GEOMETRY, nearest GEOMETRY, radius DOUBLE) ST_MaximumInscribedCircle (geom GEOMETRY,
tolerance DOUBLE)
```

Description Returns the maximum inscribed circle of the input geometry, optionally with a tolerance.

By default, the tolerance is computed as `max(width, height) / 1000`. The return value is a struct with the center of the circle, the nearest point to the center on the boundary of the geometry and the radius of the circle.

Example

```
-- Find the maximum inscribed circle of a square
SELECT ST_MaximumInscribedCircle(
    ST_GeomFromText('POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))')
);
----
{'center': POINT (5 5), 'nearest': POINT (5 0), 'radius': 5.0}
```

ST_MinimumRotatedRectangle

Signature

```
GEOMETRY ST_MinimumRotatedRectangle (geom GEOMETRY)
```

Description Returns the minimum rotated rectangle that bounds the input geometry, finding the surrounding box that has the lowest area by using a rotated rectangle, rather than taking the lowest and highest coordinate values as per ST_Envelope().

ST_Multi

Signature

```
GEOMETRY ST_Multi (geom GEOMETRY)
```

Description Turns a single geometry into a multi geometry.

If the geometry is already a multi geometry, it is returned as is.

Example

```
SELECT ST_Multi(ST_GeomFromText('POINT(1 2)'));
----
MULTIPOINT (1 2)

SELECT ST_Multi(ST_GeomFromText('LINESTRING(1 1, 2 2)'));
----
MULTILINESTRING ((1 1, 2 2))

SELECT ST_Multi(ST_GeomFromText('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'));
----
MULTIPOLYGON (((0 0, 0 1, 1 1, 1 0, 0 0)))
```

ST_NGeometries

Signature

```
INTEGER ST_NGeometries (geom GEOMETRY)
```

Description Returns the number of component geometries in a collection geometry. If the input geometry is not a collection, this function returns 0 or 1 depending on if the geometry is empty or not.

ST_NInteriorRings

Signatures

```
INTEGER ST_NInteriorRings (geom GEOMETRY)
INTEGER ST_NInteriorRings (polygon POLYGON_2D)
```

Description Returns the number of interior rings of a polygon

---

ST_NPoints

Signatures

```
UINTEGER ST_NPoints (geom GEOMETRY)
UBIGINT ST_NPoints (point POINT_2D)
UBIGINT ST_NPoints (linestring LINESTRING_2D)
UBIGINT ST_NPoints (polygon POLYGON_2D)
UBIGINT ST_NPoints (box BOX_2D)
```

Description Returns the number of vertices within a geometry

ST_Node

Signature

```
GEOMETRY ST_Node (geom GEOMETRY)
```

Description Returns a ”noded” MultiLinestring, produced by combining a collection of input linestrings and adding additional vertices where they intersect.

Example

```
-- Create a noded multilinestring from two intersecting lines
SELECT ST_Node(
    ST_GeomFromText('MULTILINESTRING((0 0, 2 2), (0 2, 2 0))')
);
----
MULTILINESTRING ((0 0, 1 1), (1 1, 2 2), (0 2, 1 1), (1 1, 2 0))
```

ST_Normalize

Signature

```
GEOMETRY ST_Normalize (geom GEOMETRY)
```

Description Returns the ”normalized” representation of the geometry

---

ST_NumGeometries

Signature

```
INTEGER ST_NumGeometries (geom GEOMETRY)
```

Description Returns the number of component geometries in a collection geometry. If the input geometry is not a collection, this function returns 0 or 1 depending on if the geometry is empty or not.

ST_NumInteriorRings

Signatures

```
INTEGER ST_NumInteriorRings (geom GEOMETRY)
INTEGER ST_NumInteriorRings (polygon POLYGON_2D)
```

Description Returns the number of interior rings of a polygon

---

ST_NumPoints

Signatures

```
UINTEGER ST_NumPoints (geom GEOMETRY)
UBIGINT ST_NumPoints (point POINT_2D)
UBIGINT ST_NumPoints (linestring LINESTRING_2D)
UBIGINT ST_NumPoints (polygon POLYGON_2D)
UBIGINT ST_NumPoints (box BOX_2D)
```

Description Returns the number of vertices within a geometry

ST_Overlaps

Signature

```
BOOLEAN ST_Overlaps (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the geometries overlap

ST_Perimeter

Signatures

```
DOUBLE ST_Perimeter (geom GEOMETRY)
DOUBLE ST_Perimeter (polygon POLYGON_2D)
DOUBLE ST_Perimeter (box BOX_2D)
```

Description Returns the length of the perimeter of the geometry

---

ST_Perimeter_Spheroid

Signatures

```
DOUBLE ST_Perimeter_Spheroid (geom GEOMETRY)
DOUBLE ST_Perimeter_Spheroid (poly POLYGON_2D)
```

Description Returns the length of the perimeter in meters using an ellipsoidal model of the earths surface

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the length is returned in meters. This function uses the GeographicLib library, calculating the perimeter using an ellipsoidal model of the earth. This is a highly accurate method for calculating the perimeter of a polygon taking the curvature of the earth into account, but is also the slowest.

Returns `0.0` for any geometry that is not a `POLYGON`, `MULTIPOLYGON` or `GEOMETRYCOLLECTION` containing polygon geometries.

ST_Point

Signature

```
GEOMETRY ST_Point (x DOUBLE, y DOUBLE)
```

Description Creates a GEOMETRY point

ST_Point2D

Signature

```
POINT_2D ST_Point2D (x DOUBLE, y DOUBLE)
```

Description Creates a POINT_2D

ST_Point2DFromWKB

Signature

```
GEOMETRY ST_Point2DFromWKB (point POINT_2D)
```

Description Deserialize a POINT_2D from a WKB encoded blob

ST_Point3D

Signature

```
POINT_3D ST_Point3D (x DOUBLE, y DOUBLE, z DOUBLE)
```

Description Creates a POINT_3D

ST_Point4D

Signature

```
POINT_4D ST_Point4D (x DOUBLE, y DOUBLE, z DOUBLE, m DOUBLE)
```

Description Creates a POINT_4D

ST_PointN

Signatures

```
GEOMETRY ST_PointN (geom GEOMETRY, index INTEGER)
POINT_2D ST_PointN (linestring LINESTRING_2D, index INTEGER)
```

Description Returns the n'th vertex from the input geometry as a point geometry

---

ST_PointOnSurface

Signature

```
GEOMETRY ST_PointOnSurface (geom GEOMETRY)
```

Description Returns a point guaranteed to lie on the surface of the geometry

---

ST_Points

Signature

```
GEOMETRY ST_Points (geom GEOMETRY)
```

Description Collects all the vertices in the geometry into a MULTIPOINT

Example

```
select st_points('LINESTRING(1 1, 2 2)'::geometry);
----
MULTIPOINT (1 1, 2 2)

select st_points('MULTIPOLYGON Z EMPTY'::geometry);
----
MULTIPOINT Z EMPTY
```

ST_Polygon2DFromWKB

Signature

```
GEOMETRY ST_Polygon2DFromWKB (polygon POLYGON_2D)
```

Description Deserialize a POLYGON_2D from a WKB encoded blob

ST_Polygonize

Signature

```
GEOMETRY ST_Polygonize (geometries GEOMETRY[])
```

Description Returns a polygonized representation of the input geometries

Example

```
-- Create a polygon from a closed linestring ring
SELECT ST_Polygonize([
    ST_GeomFromText('LINESTRING(0 0, 0 10, 10 10, 10 0, 0 0)')
]);

GEOMETRYCOLLECTION (POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0)))
```

ST_QuadKey

Signatures

```
VARCHAR ST_QuadKey (longitude DOUBLE, latitude DOUBLE, level INTEGER)
VARCHAR ST_QuadKey (point GEOMETRY, level INTEGER)
```

Description Compute the quadkey for a given lon/lat point at a given level. Note that the parameter order is **longitude**, **latitude**.

`level` has to be between 1 and 23, inclusive.

The input coordinates will be clamped to the lon/lat bounds of the earth (longitude between -180 and 180, latitude between -85.05112878 and 85.05112878).

The geometry overload throws an error if the input geometry is not a `POINT`

Example

```
SELECT ST_QuadKey(st_point(11.08, 49.45), 10);
----
1333203202
```

ST_ReducePrecision

Signature

```
GEOMETRY ST_ReducePrecision (geom GEOMETRY, precision DOUBLE)
```

Description Returns the geometry with all vertices reduced to the given precision

---

ST_RemoveRepeatedPoints

Signatures

```
LINESTRING_2D ST_RemoveRepeatedPoints (line LINESTRING_2D)
LINESTRING_2D ST_RemoveRepeatedPoints (line LINESTRING_2D, tolerance DOUBLE)
GEOMETRY ST_RemoveRepeatedPoints (geom GEOMETRY)
GEOMETRY ST_RemoveRepeatedPoints (geom GEOMETRY, tolerance DOUBLE)
```

Description Remove repeated points from a LINESTRING.

ST_Reverse

Signature

```
GEOMETRY ST_Reverse (geom GEOMETRY)
```

Description Returns the geometry with the order of its vertices reversed

---

ST_ShortestLine

Signature

```
GEOMETRY ST_ShortestLine (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns the shortest line between two geometries

ST_Simplify

Signature

```
GEOMETRY ST_Simplify (geom GEOMETRY, tolerance DOUBLE)
```

Description Returns a simplified version of the geometry

ST_SimplifyPreserveTopology

Signature

```
GEOMETRY ST_SimplifyPreserveTopology (geom GEOMETRY, tolerance DOUBLE)
```

Description Returns a simplified version of the geometry that preserves topology

---

ST_StartPoint

Signatures

```
GEOMETRY ST_StartPoint (geom GEOMETRY)
POINT_2D ST_StartPoint (line LINESTRING_2D)
```

Description Returns the start point of a LINESTRING.

ST_TileEnvelope

Signature

```
GEOMETRY ST_TileEnvelope (tile_zoom INTEGER, tile_x INTEGER, tile_y INTEGER)
```

Description The `ST_TileEnvelope` scalar function generates tile envelope rectangular polygons from specified zoom level and tile indices.

This is used in MVT generation to select the features corresponding to the tile extent. The envelope is in the Web Mercator coordinate reference system (EPSG:3857). The tile pyramid starts at zoom level 0, corresponding to a single tile for the world. Each zoom level doubles the number of tiles in each direction, such that zoom level 1 is 2 tiles wide by 2 tiles high, zoom level 2 is 4 tiles wide by 4 tiles high, and so on. Tile indices start at `[x=0, y=0]` at the top left, and increase down and right. For example, at zoom level 2, the top right tile is `[x=3, y=0]`, the bottom left tile is `[x=0, y=3]`, and the bottom right is `[x=3, y=3]`.

```
SELECT ST_TileEnvelope(2, 3, 1);
```

Example

```
SELECT ST_TileEnvelope(2, 3, 1);
```

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                                    st_tileenvelope(2, 3, 1)                                                                  │
│                                                           geometry                                                                           │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│ POLYGON ((1.00188E+07 0, 1.00188E+07 1.00188E+07, 2.00375E+07 1.00188E+07, 2.00375E+07 0, 1.00188E+07 0))                                      │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

ST_Touches

Signature

```
BOOLEAN ST_Touches (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the geometries touch

ST_Transform

Signatures

```
BOX_2D ST_Transform (box BOX_2D, source_crs VARCHAR, target_crs VARCHAR)
BOX_2D ST_Transform (box BOX_2D, source_crs VARCHAR, target_crs VARCHAR, always_xy BOOLEAN)
POINT_2D ST_Transform (point POINT_2D, source_crs VARCHAR, target_crs VARCHAR)
POINT_2D ST_Transform (point POINT_2D, source_crs VARCHAR, target_crs VARCHAR, always_xy BOOLEAN)
GEOMETRY ST_Transform (geom GEOMETRY, source_crs VARCHAR, target_crs VARCHAR)
GEOMETRY ST_Transform (geom GEOMETRY, source_crs VARCHAR, target_crs VARCHAR, always_xy BOOLEAN)
```

Description Transforms a geometry between two coordinate systems

The source and target coordinate systems can be specified using any format that the PROJ library supports.

The third optional `always_xy` parameter can be used to force the input and output geometries to be interpreted as having a [easting, northing] coordinate axis order regardless of what the source and target coordinate system definition says. This is particularly useful when transforming to/from the WGS84/EPSG:4326 coordinate system (what most people think of when they hear "longitude"/"latitude" or "GPS coordinates"), which is defined as having a [latitude, longitude] axis order even though [longitude, latitude] is commonly used in practice (e.g. in GeoJSON). More details available in the PROJ documentation.

DuckDB spatial vendors its own static copy of the PROJ database of coordinate systems, so if you have your own installation of PROJ on your system the available coordinate systems may differ to what's available in other GIS software.

Example

```
-- Transform a geometry from EPSG:4326 to EPSG:3857 (WGS84 to WebMercator)
-- Note that since WGS84 is defined as having a [latitude, longitude] axis order
-- we follow the standard and provide the input geometry using that axis order,
-- but the output will be [easting, northing] because that is what's defined by
-- WebMercator.

SELECT
    ST_Transform(
        st_point(52.373123, 4.892360),
        'EPSG:4326',
        'EPSG:3857'
    );
----
POINT (544615.0239773799 6867874.103539125)

-- Alternatively, let's say we got our input point from e.g. a GeoJSON file,
-- which uses WGS84 but with [longitude, latitude] axis order. We can use the
-- `always_xy` parameter to force the input geometry to be interpreted as having
-- a [northing, easting] axis order instead, even though the source coordinate
-- reference system definition (WGS84) says otherwise.

SELECT
    ST_Transform(
        -- note the axis order is reversed here
        st_point(4.892360, 52.373123),
        'EPSG:4326',
        'EPSG:3857',
        always_xy := true
    );
----
POINT (544615.0239773799 6867874.103539125)

-- Transform a geometry from OSG36 British National Grid EPSG:27700 to EPSG:4326 WGS84
-- Standard transform is often fine for the first few decimal places before being wrong
-- which could result in an error starting at about 10m and possibly much more
SELECT ST_Transform(bng, 'EPSG:27700', 'EPSG:4326', xy := true) AS without_grid_file
FROM (SELECT ST_GeomFromText('POINT( 170370.718 11572.405 )') AS bng);
----
POINT (-5.202992651563592 49.96007490162923)

-- By using an official NTv2 grid file, we can reduce the error down around the 9th decimal place
-- which in theory is below a millimetre, and in practice unlikely that your coordinates are that precise
-- British National Grid "NTv2 format files" download available here:
-- https://www.ordnancesurvey.co.uk/products/os-net/for-developers
SELECT ST_Transform(bng
    , '+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000 +ellps=airy +units=m +no_
defs +nadgrids=/full/path/to/OSTN15-NTv2/OSTN15_NTv2_OSGBtoETRS.gsb +type=crs'
    , 'EPSG:4326', xy := true) AS with_grid_file
FROM (SELECT ST_GeomFromText('POINT( 170370.718 11572.405 )') AS bng) t;
----
POINT (-5.203046090608746 49.96006137018598)
```

ST_Union

Signature

```
GEOMETRY ST_Union (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns the union of two geometries

ST_VoronoiDiagram

Signature

```
GEOMETRY ST_VoronoiDiagram (geom GEOMETRY)
```

Description Returns the Voronoi diagram of the supplied MultiPoint geometry

---

ST_Within

Signatures

```
BOOLEAN ST_Within (geom1 POINT_2D, geom2 POLYGON_2D)
BOOLEAN ST_Within (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the first geometry is within the second

---

ST_WithinProperly

Signature

```
BOOLEAN ST_WithinProperly (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description Returns true if the first geometry "properly" is contained by the second geometry

This function functions the same as `ST_ContainsProperly`, but the arguments are swapped.

ST_X

Signatures

```
DOUBLE ST_X (geom GEOMETRY)
DOUBLE ST_X (point POINT_2D)
```

Description Returns the X coordinate of a point geometry

Example

```
SELECT ST_X(ST_Point(1, 2))
```

ST_XMax

Signatures

```
DOUBLE ST_XMax (geom GEOMETRY)
DOUBLE ST_XMax (point POINT_2D)
DOUBLE ST_XMax (line LINESTRING_2D)
DOUBLE ST_XMax (polygon POLYGON_2D)
DOUBLE ST_XMax (box BOX_2D)
FLOAT ST_XMax (box BOX_2DF)
```

Description Returns the maximum X coordinate of a geometry

Example

```
SELECT ST_XMax(ST_Point(1, 2))
```

ST_XMin

Signatures

```
DOUBLE ST_XMin (geom GEOMETRY)
DOUBLE ST_XMin (point POINT_2D)
DOUBLE ST_XMin (line LINESTRING_2D)
DOUBLE ST_XMin (polygon POLYGON_2D)
DOUBLE ST_XMin (box BOX_2D)
FLOAT ST_XMin (box BOX_2DF)
```

Description Returns the minimum X coordinate of a geometry

Example

```
SELECT ST_XMin(ST_Point(1, 2))
```

ST_Y

Signatures

```
DOUBLE ST_Y (geom GEOMETRY)
DOUBLE ST_Y (point POINT_2D)
```

Description Returns the Y coordinate of a point geometry

Example

```
SELECT ST_Y(ST_Point(1, 2))
```

ST_YMax

Signatures

```
DOUBLE ST_YMax (geom GEOMETRY)
DOUBLE ST_YMax (point POINT_2D)
DOUBLE ST_YMax (line LINESTRING_2D)
DOUBLE ST_YMax (polygon POLYGON_2D)
DOUBLE ST_YMax (box BOX_2D)
FLOAT ST_YMax (box BOX_2DF)
```

Description Returns the maximum Y coordinate of a geometry

Example

```
SELECT ST_YMax(ST_Point(1, 2))
```

ST_YMin

Signatures

```
DOUBLE ST_YMin (geom GEOMETRY)
DOUBLE ST_YMin (point POINT_2D)
DOUBLE ST_YMin (line LINESTRING_2D)
DOUBLE ST_YMin (polygon POLYGON_2D)
DOUBLE ST_YMin (box BOX_2D)
FLOAT ST_YMin (box BOX_2DF)
```

Description Returns the minimum Y coordinate of a geometry

Example

```
SELECT ST_YMin(ST_Point(1, 2))
```

ST_Z

Signature

```
DOUBLE ST_Z (geom GEOMETRY)
```

Description Returns the Z coordinate of a point geometry

Example

```
SELECT ST_Z(ST_Point(1, 2, 3))
```

ST_ZMFlag

Signatures

```
UTINYINT ST_ZMFlag (geom GEOMETRY)
UTINYINT ST_ZMFlag (wkb WKB_BLOB)
```

Description Returns a flag indicating the presence of Z and M values in the input geometry. 0 = No Z or M values 1 = M values only 2 = Z values only 3 = Z and M values

Example

```
-- ZMFlag for a 2D geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT(1 1)'));
----
0

-- ZMFlag for a 3DZ geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT Z(1 1 1)'));
----
2

-- ZMFlag for a 3DM geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT M(1 1 1)'));
----
1

-- ZMFlag for a 4D geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT ZM(1 1 1 1)'));
----
3
```

ST_ZMax

Signature

```
DOUBLE ST_ZMax (geom GEOMETRY)
```

Description Returns the maximum Z coordinate of a geometry

Example

```
SELECT ST_ZMax(ST_Point(1, 2, 3))
```

ST_ZMin

Signature

```
DOUBLE ST_ZMin (geom GEOMETRY)
```

Description Returns the minimum Z coordinate of a geometry

Example

```
SELECT ST_ZMin(ST_Point(1, 2, 3))
```

Aggregate Functions

ST_AsMVT

Signatures

```
BLOB ST_AsMVT (col0 ANY)
BLOB ST_AsMVT (col0 ANY, col1 VARCHAR)
BLOB ST_AsMVT (col0 ANY, col1 VARCHAR, col2 INTEGER)
BLOB ST_AsMVT (col0 ANY, col1 VARCHAR, col2 INTEGER, col3 VARCHAR)
BLOB ST_AsMVT (col0 ANY, col1 VARCHAR, col2 INTEGER, col3 VARCHAR, col4 VARCHAR)
```

Description Make a Mapbox Vector Tile from a set of geometries and properties The function takes as input a row type (STRUCT) containing a geometry column and any number of property columns. It returns a single binary BLOB containing the Mapbox Vector Tile.

The function has the following signature:

`ST_AsMVT(row STRUCT, layer_name VARCHAR DEFAULT 'layer', extent INTEGER DEFAULT 4096, geom_column_name VARCHAR DEFAULT NULL, feature_id_column_name VARCHAR DEFAULT NULL) -> BLOB`

- The first argument is a struct containing the geometry and properties.
- The second argument is the name of the layer in the vector tile. This argument is optional and defaults to `'layer'`.
- The third argument is the extent of the tile. This argument is optional and defaults to 4096.
- The fourth argument is the name of the geometry column in the input row. This argument is optional. If not provided, the first geometry column in the input row will be used. If multiple geometry columns are present, an error will be raised.
- The fifth argument is the name of the feature id column in the input row. This argument is optional. If provided, the values in this column will be used as feature ids in the vector tile. The column must be of type INTEGER or BIGINT. If set to negative or NULL, a feature id will not be assigned to the corresponding feature.

The input struct must contain exactly one geometry column of type GEOMETRY. It can contain any number of property columns of types VARCHAR, FLOAT, DOUBLE, INTEGER, BIGINT, or BOOLEAN.

Example:

```
SELECT ST_AsMVT({'geom': geom, 'id': id, 'name': name}, 'cities', 4096, 'geom', 'id') AS tile
FROM cities;
```

This example creates a vector tile named `cities` with an extent of 4096 from the `cities` table, using `geom` as the geometry column and `id` as the feature id column.

However, you probably want to use the `ST_AsMVTGeom` function to first transform and clip your geometries to the tile extent. The following example assumes the geometry is in WebMercator ("EPSG:3857") coordinates. Replace <z>{:.language-sql .highlight}, <x>{:.language-sql .highlight}, and <y>{:.language-sql .highlight} with the appropriate tile coordinates, <your_table>{:.language-sql .highlight} with your table name, and <tile_path> with the path to write the tile to.

```
COPY (
    SELECT ST_AsMVT({
        "geometry": ST_AsMVTGeom(
            geometry,
            ST_Extent(ST_TileEnvelope(<z>, <x>, <y>)),
            4096,
            256,
            false
        )
```

```
    })
    FROM <your_table> WHERE ST_Intersects(geometry, ST_TileEnvelope(<z>, <x>, <y>))
) to <tile_path> (FORMAT 'BLOB');
```

ST_CoverageInvalidEdges_Agg

Signatures

```
GEOMETRY ST_CoverageInvalidEdges_Agg (col0 GEOMETRY)
GEOMETRY ST_CoverageInvalidEdges_Agg (col0 GEOMETRY, col1 DOUBLE)
```

Description Returns the invalid edges of a coverage geometry

ST_CoverageSimplify_Agg

Signatures

```
GEOMETRY ST_CoverageSimplify_Agg (col0 GEOMETRY, col1 DOUBLE)
GEOMETRY ST_CoverageSimplify_Agg (col0 GEOMETRY, col1 DOUBLE, col2 BOOLEAN)
```

Description Simplifies a set of geometries while maintaining coverage

---

ST_CoverageUnion_Agg

Signature

```
GEOMETRY ST_CoverageUnion_Agg (col0 GEOMETRY)
```

Description Unions a set of geometries while maintaining coverage

---

ST_Envelope_Agg

Signature

```
GEOMETRY ST_Envelope_Agg (col0 GEOMETRY)
```

Description Alias for ST_Extent_Agg.

Computes the minimal-bounding-box polygon containing the set of input geometries.

Example

```
SELECT ST_Extent_Agg(geom) FROM UNNEST([ST_Point(1,1), ST_Point(5,5)]) AS _(geom);
-- POLYGON ((1 1, 1 5, 5 5, 5 1, 1 1))
```

ST_Extent_Agg

Signature

```
GEOMETRY ST_Extent_Agg (col0 GEOMETRY)
```

Description Computes the minimal-bounding-box polygon containing the set of input geometries

Example

```
SELECT ST_Extent_Agg(geom) FROM UNNEST([ST_Point(1,1), ST_Point(5,5)]) AS _(geom);
-- POLYGON ((1 1, 1 5, 5 5, 5 1, 1 1))
```

ST_Intersection_Agg

Signature

```
GEOMETRY ST_Intersection_Agg (col0 GEOMETRY)
```

Description Computes the intersection of a set of geometries

ST_MemUnion_Agg

Signature

```
GEOMETRY ST_MemUnion_Agg (col0 GEOMETRY)
```

Description Computes the union of a set of input geometries. ”Slower, but might be more memory efficient than ST_UnionAgg as each geometry is merged into the union individually rather than all at once.

ST_Union_Agg

Signature

```
GEOMETRY ST_Union_Agg (col0 GEOMETRY)
```

Description Computes the union of a set of input geometries

Macro Functions

ST_Rotate

Signature

```
GEOMETRY ST_Rotate (geom GEOMETRY, radians double)
```

Description Alias of ST_RotateZ

ST_RotateX

Signature

```
GEOMETRY ST_RotateX (geom GEOMETRY, radians double)
```

Description Rotates a geometry around the X axis. This is a shorthand macro for calling ST_Affine.

Example

```
-- Rotate a 3D point 90 degrees (π/2 radians) around the X-axis
SELECT ST_RotateX(ST_GeomFromText('POINT Z(0 1 0)'), pi()/2);
----
POINT Z (0 0 1)
```

ST_RotateY

Signature

```
GEOMETRY ST_RotateY (geom GEOMETRY, radians double)
```

Description Rotates a geometry around the Y axis. This is a shorthand macro for calling ST_Affine.

Example

```
-- Rotate a 3D point 90 degrees (π/2 radians) around the Y-axis
SELECT ST_RotateY(ST_GeomFromText('POINT Z(1 0 0)'), pi()/2);
----
POINT Z (0 0 -1)
```

ST_RotateZ

Signature

```
GEOMETRY ST_RotateZ (geom GEOMETRY, radians double)
```

Description Rotates a geometry around the Z axis. This is a shorthand macro for calling ST_Affine.

Example

```
-- Rotate a point 90 degrees (π/2 radians) around the Z-axis
SELECT ST_RotateZ(ST_Point(1, 0), pi()/2);
----
POINT (0 1)
```

ST_Scale

Signatures

```
GEOMETRY ST_Scale (geom GEOMETRY, xs double, ys double, zs double)
GEOMETRY ST_Scale (geom GEOMETRY, xs double, ys double)
```

ST_TransScale

Signature

```
GEOMETRY ST_TransScale (geom GEOMETRY, dx double, dy double, xs double, ys double)
```

Description Translates and then scales a geometry in X and Y direction. This is a shorthand macro for calling ST_Affine.

Example

```
-- Translate by (1, 2) then scale by (2, 3)
SELECT ST_TransScale(ST_Point(1, 1), 1, 2, 2, 3);
----
POINT (4 9)
```

ST_Translate

Signatures

```
GEOMETRY ST_Translate (geom GEOMETRY, dx double, dy double, dz double)
GEOMETRY ST_Translate (geom GEOMETRY, dx double, dy double)
```

Table Functions

ST_Drivers

Signature

```
ST_Drivers ()
```

Description Returns the list of supported GDAL drivers and file formats

Note that far from all of these drivers have been tested properly. Some may require additional options to be passed to work as expected. If you run into any issues please first consult the consult the GDAL docs.

Example

```
SELECT * FROM ST_Drivers();
```

ST_GeneratePoints

Signature

```
ST_GeneratePoints (col0 BOX_2D, col1 BIGINT)
ST_GeneratePoints (col0 BOX_2D, col1 BIGINT, col2 BIGINT)
```

Description Generates a set of random points within the specified bounding box.

Takes a bounding box (`min_x`, `min_y`, `max_x`, `max_y`), a count of points to generate and optionally a seed for the random number generator.

Example

```
SELECT * FROM ST_GeneratePoints({min_x: 0, min_y:0, max_x:10, max_y:10}::BOX_2D, 5, 42);
```

ST_Read

Signature

```
ST_Read (col0 VARCHAR, keep_wkb BOOLEAN, max_batch_size INTEGER, sequential_layer_scan BOOLEAN, layer
VARCHAR, sibling_files VARCHAR[], spatial_filter WKB_BLOB, spatial_filter_box BOX_2D, allowed_drivers
VARCHAR[], open_options VARCHAR[])
```

Description Read and import a variety of geospatial file formats using the GDAL library.

The `ST_Read` table function is based on the GDAL translator library and enables reading spatial data from a variety of geospatial vector file formats as if they were DuckDB tables.

> See ST_Drivers for a list of supported file formats and drivers.

Except for the `path` parameter, all parameters are optional.

Parameter	Type	Description
`path`	VARCHAR	The path to the file to read. Mandatory
`sequential_layer_scan`	BOOLEAN	If set to true, the table function will scan through all layers sequentially and return the first layer that matches the given layer name. This is required for some drivers to work properly, e.g., the OSM driver.
`spatial_filter`	WKB_BLOB	If set to a WKB blob, the table function will only return rows that intersect with the given WKB geometry. Some drivers may support efficient spatial filtering natively, in which case it will be pushed down. Otherwise the filtering is done by GDAL which may be much slower.
`open_options`	VARCHAR[]	A list of key-value pairs that are passed to the GDAL driver to control the opening of the file. E.g., the GeoJSON driver supports a FLATTEN_NESTED_ATTRIBUTES=YES option to flatten nested attributes.
`layer`	VARCHAR	The name of the layer to read from the file. If NULL, the first layer is returned. Can also be a layer index (starting at 0).

Parameter	Type	Description
`allowed_drivers`	VARCHAR[]	A list of GDAL driver names that are allowed to be used to open the file. If empty, all drivers are allowed.
`sibling_files`	VARCHAR[]	A list of sibling files that are required to open the file. E.g., the ESRI Shapefile driver requires a .shx file to be present. Although most of the time these can be discovered automatically.
`spatial_filter_box`	BOX_2D	If set to a BOX_2D, the table function will only return rows that intersect with the given bounding box. Similar to spatial_filter.
`keep_wkb`	BOOLEAN	If set, the table function will return geometries in a wkb_ geometry column with the type WKB_BLOB (which can be cast to BLOB) instead of GEOMETRY. This is useful if you want to use DuckDB with more exotic geometry subtypes that DuckDB spatial doesn't support representing in the GEOMETRY type yet.

Note that GDAL is single-threaded, so this table function will not be able to make full use of parallelism.

By using `ST_Read`, the spatial extension also provides "replacement scans" for common geospatial file formats, allowing you to query files of these formats as if they were tables directly.

```
SELECT * FROM './path/to/some/shapefile/dataset.shp';
```

In practice this is just syntax-sugar for calling ST_Read, so there is no difference in performance. If you want to pass additional options, you should use the ST_Read table function directly.

The following formats are currently recognized by their file extension:

Format	Extension
ESRI ShapeFile	.shp
GeoPackage	.gpkg
FlatGeoBuf	.fgb

Example

```
-- Read a Shapefile
SELECT * FROM ST_Read('some/file/path/filename.shp');

-- Read a GeoJSON file
CREATE TABLE my_geojson_table AS SELECT * FROM ST_Read('some/file/path/filename.json');
```

ST_ReadOSM

Signature

```
ST_ReadOSM (col0 VARCHAR)
```

Description The `ST_ReadOsm()` table function enables reading compressed OpenStreetMap data directly from a `.osm.pbf file.`

This function uses multithreading and zero-copy protobuf parsing which makes it a lot faster than using the `ST_Read()` OSM driver, however it only outputs the raw OSM data (Nodes, Ways, Relations), without constructing any geometries. For simple node entities (like PoI's) you can trivially construct POINT geometries, but it is also possible to construct LINESTRING and POLYGON geometries by manually joining refs and nodes together in SQL, although with available memory usually being a limiting factor. The `ST_ReadOSM()` function also provides a "replacement scan" to enable reading from a file directly as if it were a table. This is just syntax sugar for calling `ST_ReadOSM()` though. Example:

```
SELECT * FROM 'tmp/data/germany.osm.pbf' LIMIT 5;
```

Example

```
SELECT *
FROM ST_ReadOSM('tmp/data/germany.osm.pbf')
WHERE tags['highway'] != []
LIMIT 5;
```

```
┌──────────────────────┬─────────┬──────────────────────┬─────────┬──────────────────────┬──────────────────────┬──────────────────────┬──────────────────────┐
│         kind         │   id    │         tags         │  refs   │         lat          │         lon          │      ref_roles       │      ref_types       │
│ enum('node', 'way'…  │ int64   │ map(varchar, varch…  │ int64[] │        double        │        double        │      varchar[]       │ enum('node', 'way', … │
├──────────────────────┼─────────┼──────────────────────┼─────────┼──────────────────────┼──────────────────────┼──────────────────────┼──────────────────────┤
│ node                 │  122351 │ {bicycle=yes, butt…  │         │           53.5492951 │            9.977553 │                      │                      │
│ node                 │  122397 │ {crossing=no, high…  │         │ 53.520990100000006   │           10.0156924 │                      │                      │
│ node                 │  122493 │ {TMC:cid_58:tabcd_…  │         │ 53.129614600000004   │            8.1970173 │                      │                      │
│ node                 │  123566 │ {highway=traffic_s…  │         │ 54.617268200000005   │            8.9718171 │                      │                      │
│ node                 │  125801 │ {TMC:cid_58:tabcd_…  │         │ 53.070685000000005   │            8.7819939 │                      │                      │
└──────────────────────┴─────────┴──────────────────────┴─────────┴──────────────────────┴──────────────────────┴──────────────────────┴──────────────────────┘
```

ST_ReadSHP

Signature

```
ST_ReadSHP (col0 VARCHAR, encoding VARCHAR)
```

Description Read a Shapefile without relying on the GDAL library

---

ST_Read_Meta

Signature

```
ST_Read_Meta (col0 VARCHAR)
ST_Read_Meta (col0 VARCHAR[])
```

Description Read the metadata from a variety of geospatial file formats using the GDAL library.

The `ST_Read_Meta` table function accompanies the `ST_Read` table function, but instead of reading the contents of a file, this function scans the metadata instead. Since the data model of the underlying GDAL library is quite flexible, most of the interesting metadata is within the returned `layers` column, which is a somewhat complex nested structure of DuckDB `STRUCT` and `LIST` types.

Example

```
-- Find the coordinate reference system authority name and code for the first layers first geometry column
in the file
SELECT
    layers[1].geometry_fields[1].crs.auth_name as name,
    layers[1].geometry_fields[1].crs.auth_code as code
FROM st_read_meta('../../tmp/data/amsterdam_roads.fgb');
```

R-Tree Indexes

The `spatial` extension provides support for spatial indexing through the R-tree extension index type.

Why Should I Use an R-Tree Index?

When working with geospatial datasets, it is very common that you want to filter rows based on their spatial relationship with a specific region of interest. Unfortunately, even though DuckDB's vectorized execution engine is pretty fast, this sort of operation does not scale very well to large datasets as it always requires a full table scan to check every row in the table. However, by indexing a table with an R-tree, it is possible to accelerate these types of queries significantly.

How Do R-Tree Indexes Work?

An R-tree is a balanced tree data structure that stores the approximate *minimum bounding rectangle* of each geometry (and the internal ID of the corresponding row) in the leaf nodes, and the bounding rectangle enclosing all of the child nodes in each internal node.

> The *minimum bounding rectangle* (MBR) of a geometry is the smallest rectangle that completely encloses the geometry. Usually when we talk about the bounding rectangle of a geometry (or the bounding "box" in the context of 2D geometry), we mean the minimum bounding rectangle. Additionally, we tend to assume that bounding boxes/rectangles are *axis-aligned,* i.e., the rectangle is **not** rotated – the sides are always parallel to the coordinate axes. The MBR of a point is the point itself.

By traversing the R-tree from top to bottom, it is possible to very quickly search a R-tree-indexed table for only those rows where the indexed geometry column intersect a specific region of interest, as you can skip searching entire sub-trees if the bounding rectangles of their parent nodes don't intersect the query region at all. Once the leaf nodes are reached, only the specific rows whose geometries intersect the query region have to be fetched from disk, and the often much more expensive exact spatial predicate check (and any other filters) only have to be executed for these rows.

What Are the Limitations of R-Tree Indexes in DuckDB?

Before you get started using the R-tree index, there are some limitations to be aware of:

- The R-tree index is only supported for the `GEOMETRY` data type.
- The R-tree index will only be used to perform "index scans" when the table is filtered (using a `WHERE` clause) with one of the following spatial predicate functions (as they all imply intersection): `ST_Equals`, `ST_Intersects`, `ST_Touches`, `ST_Crosses`, `ST_Within`, `ST_Contains`, `ST_Overlaps`, `ST_Covers`, `ST_CoveredBy`, `ST_ContainsProperly`.

- One of the arguments to the spatial predicate function must be a ”constant” (i.e., an expression whose result is known at query planning time). This is because the query planner needs to know the bounding box of the query region *before* the query itself is executed to use the R-tree index scan.

In the future we want to enable R-tree indexes to be used to accelerate additional predicate functions and more complex queries such a spatial joins.

How to Use R-Tree Indexes in DuckDB

To create an R-tree index, simply use the `CREATE INDEX` statement with the `USING RTREE` clause, passing the geometry column to index within the parentheses. For example:

```
-- Create a table with a geometry column
CREATE TABLE my_table (geom GEOMETRY);

-- Create an R-tree index on the geometry column
CREATE INDEX my_idx ON my_table USING RTREE (geom);
```

You can also pass in additional options when creating an R-tree index using the `WITH` clause to control the behavior of the R-tree index. For example, to specify the maximum number of entries per node in the R-tree, you can use the `max_node_capacity` option:

```
CREATE INDEX my_idx ON my_table USING RTREE (geom) WITH (max_node_capacity = 16);
```

The impact tweaking these options will have on performance is highly dependent on the system setup DuckDB is running on, the spatial distribution of the dataset and the query patterns of your specific workload. The defaults should be good enough, but you if you want to experiment with different parameters, see the full list of options here.

Example

Here is an example that shows how to create an R-tree index on a geometry column and where we can see that the `RTREE_INDEX_SCAN` operator is used when the table is filtered with a spatial predicate:

```
INSTALL AND LOAD spatial;

-- Create a table with 10_000 random points
CREATE TABLE t1 AS SELECT point::GEOMETRY AS geom
FROM st_generatepoints({min_x: 0, min_y: 0, max_x: 100, max_y: 100}::BOX_2D, 10_000, 1337);

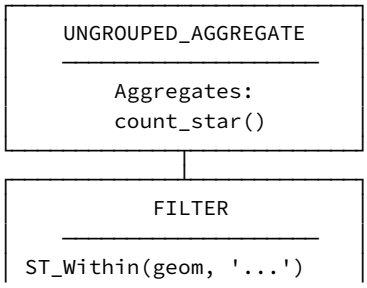
-- Create an index on the table.
CREATE INDEX my_idx ON t1 USING RTREE (geom);

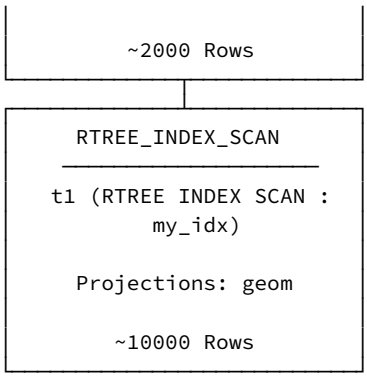
-- Perform a query with a "spatial predicate" on the indexed geometry column
-- Note how the second argument in this case, the ST_MakeEnvelope call is a "constant"
SELECT count(*) FROM t1 WHERE ST_Within(geom, ST_MakeEnvelope(45, 45, 65, 65));
```

```
390
```

We can check for ourselves that an R-tree index scan is used by using the `EXPLAIN` statement:

```
EXPLAIN SELECT count(*) FROM t1 WHERE ST_Within(geom, ST_MakeEnvelope(45, 45, 65, 65));
```





Performance Considerations

Bulk Loading & Maintenance

Creating R-trees on top of an already populated table is much faster than first creating the index and then inserting the data. This is because the R-tree will have to periodically rebalance itself and perform a somewhat costly splitting operation when a node reaches max capacity after an insert, potentially causing additional splits to cascade up the tree. However, when the R-tree index is created on an already populated table, a special bottom up "bulk loading algorithm" (Sort-Tile-Recursive) is used, which divides all entries into an already balanced tree as the total number of required nodes can be computed from the beginning.

Additionally, using the bulk loading algorithm tends to create a R-tree with a better structure (less overlap between bounding boxes), which usually leads to better query performance. If you find that the performance of querying the R-tree starts to deteriorate after a large number of updates or deletions, dropping and re-creating the index might produce a higher quality R-tree.

Memory Usage

Like DuckDB's built in ART-index, all the associated buffers containing the R-tree will be lazily loaded from disk (when running DuckDB in disk-backed mode), but they are currently never unloaded unless the index is dropped. This means that if you end up scanning the entire index, the entire index will be loaded into memory and stay there for the duration of the database connection. However, all memory used by the R-tree index (even during bulk-loading) is tracked by DuckDB, and will count towards the memory limit set by the `memory_limit` configuration parameter.

Tuning

Depending on you specific workload, you might want to experiment with the `max_node_capacity` and `min_node_capacity` options to change the structure of the R-tree and how it responds to insertions and deletions, see the full list of options here. In general, a tree with a higher total number of nodes (i.e., a lower `max_node_capacity`) *may* result in a more granular structure that enables more aggressive pruning of sub-trees during query execution, but it will also require more memory to store the tree itself and be more punishing when querying larger regions as more internal nodes will have to be traversed.

Options

The following options can be passed to the `WITH` clause when creating an R-tree index: (e.g., `CREATE INDEX my_idx ON my_table USING RTREE (geom) WITH (<option> = <value>);{:.language-sql .highlight}`)

Option	Description	Default
`max_node_capacity`	The maximum number of entries per node in the R-tree	`128`
`min_node_capacity`	The minimum number of entries per node in the R-tree	`0.4 * max_node_capacity`

Option	Description	Default

*Should a node fall under the minimum number of entries after a deletion, the node will be dissolved and all the entries reinserted from the top of the tree. This is a common operation in R-tree implementations to prevent the tree from becoming too unbalanced.

R-Tree Table Functions

The `rtree_index_dump(VARCHAR)` table function can be used to return all the nodes within an R-tree index which might come on handy when debugging, profiling or otherwise just inspecting the structure of the index. The function takes the name of the R-tree index as an argument and returns a table with the following columns:

Column name	Type	Description
`level`	`INTEGER`	The level of the node in the R-tree. The root node has level 0
`bounds`	`BOX_2DF`	The bounding box of the node
`row_id`	`ROW_TYPE`	If this is a leaf node, the `rowid` of the row in the table, otherwise `NULL`

Example:

```
-- Create a table with 64 random points
CREATE TABLE t1 AS SELECT point::GEOMETRY AS geom
FROM st_generatepoints({min_x: 0, min_y: 0, max_x: 100, max_y: 100}::BOX_2D, 64, 1337);

-- Create an R-tree index on the geometry column (with a low max_node_capacity for demonstration purposes)
CREATE INDEX my_idx ON t1 USING RTREE (geom) WITH (max_node_capacity = 4);

-- Inspect the R-tree index. Notice how the area of the bounding boxes of the branch nodes
-- decreases as we go deeper into the tree.
SELECT
  level,
  bounds::GEOMETRY AS geom,
  CASE WHEN row_id IS NULL THEN st_area(geom) ELSE NULL END AS area,
  row_id,
  CASE WHEN row_id IS NULL THEN 'branch' ELSE 'leaf' END AS kind
FROM rtree_index_dump('my_idx')
ORDER BY area DESC;
```

level int32	geom geometry	area double	row_id int64	kind varchar
0	POLYGON ((2.17285037040710…	3286.396482226409		branch
0	POLYGON ((6.00962591171264…	3193.725100864862		branch
0	POLYGON ((0.74995160102844…	3099.921458393704		branch
0	POLYGON ((14.6168870925903…	2322.2760491675654		branch
1	POLYGON ((2.17285037040710…	604.1520104388514		branch
1	POLYGON ((26.6022186279296…	569.1665467030252		branch
1	POLYGON ((35.7942314147949…	435.24662436250037		branch
1	POLYGON ((62.2643051147460…	396.39027683023596		branch
1	POLYGON ((59.5225715637207…	386.09153403820187		branch
1	POLYGON ((82.3060836791992…	369.15115640929434		branch
·	·	·	·	·
·	·	·	·	·
·	·	·	·	·
2	POLYGON ((20.5411434173584…		35	leaf
2	POLYGON ((14.6168870925903…		36	leaf

```
┌─────┬──────────────────────────────────┬──────────────────────┬──────┬──────────┐
│     2 │ POLYGON ((43.7271652221679…  │                      │   39 │ leaf     │
│     2 │ POLYGON ((53.4629211425781…  │                      │   44 │ leaf     │
│     2 │ POLYGON ((26.6022186279296…  │                      │   62 │ leaf     │
│     2 │ POLYGON ((53.1732063293457…  │                      │   63 │ leaf     │
│     2 │ POLYGON ((78.1427154541015…  │                      │   10 │ leaf     │
│     2 │ POLYGON ((75.1728591918945…  │                      │   15 │ leaf     │
│     2 │ POLYGON ((62.2643051147460…  │                      │   42 │ leaf     │
│     2 │ POLYGON ((80.5032577514648…  │                      │   49 │ leaf     │
└─────┴──────────────────────────────────┴──────────────────────┴──────┴──────────┘
84 rows (20 shown)                                                  5 columns
```

GDAL Integration

The spatial extension integrates the GDAL translator library to read and write spatial data from a variety of geospatial vector file formats. See the documentation for the `st_read` table function for how to make use of this in practice.

In order to spare users from having to setup and install additional dependencies on their system, the spatial extension bundles its own copy of the GDAL library. This also means that spatial's version of GDAL may not be the latest version available or provide support for all of the file formats that a system-wide GDAL installation otherwise would. Refer to the section on the `st_drivers` table function to inspect which GDAL drivers are currently available.

GDAL Based COPY Function

The spatial extension does not only enable *importing* geospatial file formats (through the `ST_Read` function), it also enables *exporting* DuckDB tables to different geospatial vector formats through a GDAL based COPY function.

For example, to export a table to a GeoJSON file, with generated bounding boxes, you can use the following query:

```
COPY <table> TO 'some/file/path/filename.geojson'
WITH (FORMAT gdal, DRIVER 'GeoJSON', LAYER_CREATION_OPTIONS ('WRITE_BBOX=YES', 'DESCRIPTION=a sample
geojson'), SRS 'EPSG:4326');
```

Available options:

- `FORMAT`: is the only required option and must be set to `GDAL` to use the GDAL based copy function.
- `DRIVER`: is the GDAL driver to use for the export. Use `ST_Drivers()` to list the names of all available drivers.
- `GEOMETRY_TYPE` : the geometry type for the created layer to pass to the GDAL driver. One of `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` and `GEOMETRYCOLLECTION`.
- `LAYER_NAME` : assign an alternate name to the new layer
- `LAYER_CREATION_OPTIONS`: list of layer creation options to pass to the GDAL driver. Use parenthesis to pass a list of options. See the GDAL docs for the driver you are using for a list of available options.
- `DATASET_CREATION_OPTIONS`: list of dataset creation options to pass to the GDAL driver. Use parenthesis to pass a list of options. See the GDAL docs for the driver you are using for a list of available options.
- SRS or CRS: Set a spatial reference system as metadata to use for the export. This can be a WKT string, an EPSG code or a proj-string, basically anything you would normally be able to pass to GDAL. Note that this will **not** perform any reprojection of the input geometry, it just sets the metadata if the target driver supports it.

Limitations

Note that only vector based drivers are supported by the GDAL integration. Reading and writing raster formats are not supported.

SQLite Extension

The SQLite extension allows DuckDB to directly read and write data from a SQLite database file. The data can be queried directly from the underlying SQLite tables. Data can be loaded from SQLite tables into DuckDB tables, or vice versa.

Installing and Loading

The `sqlite` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD sqlite;
```

Usage

To make a SQLite file accessible to DuckDB, use the ATTACH statement with the `sqlite` or `sqlite_scanner` type. Attached SQLite databases support both read and write operations.

For example, to attach to the `sakila.db` file, run:

```
ATTACH 'sakila.db' (TYPE sqlite);
USE sakila;
```

The tables in the file can be read as if they were normal DuckDB tables, but the underlying data is read directly from the SQLite tables in the file at query time.

```
SHOW TABLES;
```

name
actor
address
category
city
country
customer
customer_list
film
film_actor
film_category
film_list
film_text
inventory
language
payment

name
rental
sales_by_film_category
sales_by_store
staff
staff_list
store

You can query the tables using SQL, e.g., using the example queries from `sakila-examples.sql`:

```
SELECT
    cat.name AS category_name,
    sum(ifnull(pay.amount, 0)) AS revenue
FROM category cat
LEFT JOIN film_category flm_cat
       ON cat.category_id = flm_cat.category_id
LEFT JOIN film fil
       ON flm_cat.film_id = fil.film_id
LEFT JOIN inventory inv
       ON fil.film_id = inv.film_id
LEFT JOIN rental ren
       ON inv.inventory_id = ren.inventory_id
LEFT JOIN payment pay
       ON ren.rental_id = pay.rental_id
GROUP BY cat.name
ORDER BY revenue DESC
LIMIT 5;
```

Data Types

SQLite is a weakly typed database system. As such, when storing data in a SQLite table, types are not enforced. The following is valid SQL in SQLite:

```
CREATE TABLE numbers (i INTEGER);
INSERT INTO numbers VALUES ('hello');
```

DuckDB is a strongly typed database system, as such, it requires all columns to have defined types and the system rigorously checks data for correctness.

When querying SQLite, DuckDB must deduce a specific column type mapping. DuckDB follows SQLite's type affinity rules with a few extensions.

1. If the declared type contains the string `INT` then it is translated into the type `BIGINT`
2. If the declared type of the column contains any of the strings `CHAR`, `CLOB`, or `TEXT` then it is translated into `VARCHAR`.
3. If the declared type for a column contains the string `BLOB` or if no type is specified then it is translated into `BLOB`.
4. If the declared type for a column contains any of the strings `REAL`, `FLOA`, `DOUB`, `DEC` or `NUM` then it is translated into `DOUBLE`.
5. If the declared type is `DATE`, then it is translated into `DATE`.
6. If the declared type contains the string `TIME`, then it is translated into `TIMESTAMP`.
7. If none of the above apply, then it is translated into `VARCHAR`.

As DuckDB enforces the corresponding columns to contain only correctly typed values, we cannot load the string “hello” into a column of type `BIGINT`. As such, an error is thrown when reading from the “numbers” table above:

```
Mismatch Type Error: Invalid type in column "i": column was declared as integer, found "hello" of type
"text" instead.
```

This error can be avoided by setting the `sqlite_all_varchar` option:

```
SET GLOBAL sqlite_all_varchar = true;
```

When set, this option overrides the type conversion rules described above, and instead always converts the SQLite columns into a VARCHAR column. Note that this setting must be set *before* `sqlite_attach` is called.

Opening SQLite Databases Directly

SQLite databases can also be opened directly and can be used transparently instead of a DuckDB database file. In any client, when connecting, a path to a SQLite database file can be provided and the SQLite database will be opened instead.

For example, with the shell, a SQLite database can be opened as follows:

```
duckdb sakila.db
```

```
SELECT first_name
FROM actor
LIMIT 3;
```

first_name
PENELOPE
NICK
ED

Writing Data to SQLite

In addition to reading data from SQLite, the extension also allows you to create new SQLite database files, create tables, ingest data into SQLite and make other modifications to SQLite database files using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a SQLite database to Parquet, or read data from a Parquet file into SQLite.

Below is a brief example of how to create a new SQLite database and load data into it.

```
ATTACH 'new_sqlite_database.db' AS sqlite_db (TYPE sqlite);
CREATE TABLE sqlite_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

The resulting SQLite database can then be read into from SQLite.

```
sqlite3 new_sqlite_database.db
```

```
SQLite version 3.39.5 2022-10-14 20:58:05
sqlite> SELECT * FROM tbl;
```

```
id  name
--  ------
42  DuckDB
```

Many operations on SQLite tables are supported. All these operations directly modify the SQLite database, and the result of subsequent operations can then be read using SQLite.

Concurrency

DuckDB can read or modify a SQLite database while DuckDB or SQLite reads or modifies the same database from a different thread or a separate process. More than one thread or process can read the SQLite database at the same time, but only a single thread or process can write to the database at one time. Database locking is handled by the SQLite library, not DuckDB. Within the same process, SQLite uses mutexes. When accessed from different processes, SQLite uses file system locks. The locking mechanisms also depend on SQLite configuration, like WAL mode. Refer to the SQLite documentation on locking for more information.

> **Warning.** Linking multiple copies of the SQLite library into the same application can lead to application errors. See sqlite_scanner Issue #82 for more information.

Attach Options

The following options can be passed to ATTACH when attaching a SQLite database:

Option	Description	Default
BUSY_TIMEOUT	Time in milliseconds to wait when the SQLite file is locked by another process before returning a `database is locked` error. Set to `0` to fail immediately.	`5000`
JOURNAL_MODE	SQLite journaling mode, e.g., `'WAL'`.	(SQLite default)

```
ATTACH 'file.db' AS f (TYPE sqlite, BUSY_TIMEOUT 0);
ATTACH 'file.db' AS f (TYPE sqlite, JOURNAL_MODE 'WAL');
```

Settings

The extension exposes the following configuration parameters.

Name	Description	Default
`sqlite_debug_show_queries`	DEBUG SETTING: print all queries sent to SQLite to stdout	`false`

Supported Operations

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE sqlite_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM sqlite_db.tbl;
```

id	name
42	DuckDB

COPY

```
COPY sqlite_db.tbl TO 'data.parquet';
COPY sqlite_db.tbl FROM 'data.parquet';
```

UPDATE

```
UPDATE sqlite_db.tbl SET name = 'Woohoo' WHERE id = 42;
```

DELETE

```
DELETE FROM sqlite_db.tbl WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE sqlite_db.tbl ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE sqlite_db.tbl;
```

CREATE VIEW

```
CREATE VIEW sqlite_db.v1 AS SELECT 42;
```

Transactions

```
CREATE TABLE sqlite_db.tmp (i INTEGER);

BEGIN;
INSERT INTO sqlite_db.tmp VALUES (42);
SELECT * FROM sqlite_db.tmp;
```

i
42

```
ROLLBACK;
SELECT * FROM sqlite_db.tmp;
```

i

> **Deprecated.** The old `sqlite_attach` function is deprecated. It is recommended to switch over to the new ATTACH syntax.

Compatibility

The SQLite extension can read databases written by Turso, a Rust rewrite of SQLite.

SQLSmith Extension

The `sqlsmith` extension is used for testing.

Installing and Loading

```
INSTALL AND LOAD sqlsmith;
```

Functions

The `sqlsmith` extension registers the following functions:

- `sqlsmith`
- `fuzzyduck`
- `reduce_sql_statement`
- `fuzz_all_functions`

TPC-DS Extension

> New In DuckDB 2.0, the `dsdgen` data generator was updated to match the TPC-DS version 4 tools. As a result, the generated data differs slightly from the data generated by DuckDB v1.x. See the related PR for more details.

The `tpcds` extension implements the data generator and queries for the TPC-DS benchmark.

Installing and Loading

The `tpcds` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL AND LOAD tpcds;
```

Usage

To generate data for scale factor 1, use:

```
CALL dsdgen(sf = 1);
```

To run a query, e.g., query 8, use:

```
PRAGMA tpcds(8);
```

s_store_name	sum(ss_net_profit)
able	-10354620.18
ation	-10576395.52
bar	-10625236.01
ese	-10076698.16
ought	-10994052.78

Listing Queries

To list all 99 queries, run:

```
FROM tpcds_queries();
```

This function returns a table with columns `query_nr` and `query`.

Listing Expected Answers

To produce the expected results for all queries on scale factors 1 and 10, run the following (note that in DuckDB 2.0.0 this fails with a `Not implemented Error` for scale factor 10):

```
FROM tpcds_answers();
```

This function returns a table with columns `query_nr`, `scale_factor` and `answer`.

Generating the Schema

It's possible to generate the schema of TPC-DS without any data by setting the scale factor to 0:

```
CALL dsdgen(sf = 0);
```

Data Generator Parameters

The data generator function `dsdgen` has the following parameters:

Name	Type	Description
`catalog`	VARCHAR	Target catalog
`keys`	BOOLEAN	Generate primary and foreign keys
`overwrite`	BOOLEAN	(Not used)
`schema`	VARCHAR	Target schema
`sf`	DOUBLE	Scale factor
`suffix`	VARCHAR	Append the `suffix` to table names

Pre-Generated Datasets

Pre-generated DuckDB databases for TPC-DS are available for download. Note that these were generated with DuckDB v1.x. DuckDB v2.x generates slightly different TPC-DS files, compatible with TPC-DS v4.

- `tpcds-sf10.db` (2.9 GB)
- `tpcds-sf30.db` (7.7 GB)
- `tpcds-sf100.db` (26.6 GB)
- `tpcds-sf300.db` (79.3 GB)

Limitations

The `tpcds(<query_id>){:.language-sql .highlight}` function runs a fixed TPC-DS query with pre-defined bind parameters (a.k.a. substitution parameters). It is not possible to change the query parameters using the `tpcds` extension.

TPC-H Extension

The `tpch` extension implements the data generator and queries for the TPC-H benchmark.

Installing and Loading

The `tpch` extension is shipped by default in some DuckDB builds, otherwise it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL AND LOAD tpch;
```

Benchmarking with the TPC-H Workload

To run the full TPC-H workload with DuckDB, use the standalone DuckDB TPC-H implementation project.

Usage

Generating Data

To generate data for scale factor 1, use:

```
CALL dbgen(sf = 1);
```

Calling `dbgen` does not clean up existing TPC-H tables. To clean up existing tables, use `DROP TABLE` before running `dbgen`:

```
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS lineitem;
DROP TABLE IF EXISTS nation;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS part;
DROP TABLE IF EXISTS partsupp;
DROP TABLE IF EXISTS region;
DROP TABLE IF EXISTS supplier;
```

Running a Query

To run a query, e.g., query 4, use:

```
PRAGMA tpch(4);
```

o_orderpriority	order_count
1-URGENT	10594
2-HIGH	10476
3-MEDIUM	10410
4-NOT SPECIFIED	10556
5-LOW	10487

Listing Queries

To list all 22 queries, run:

```
FROM tpch_queries();
```

This function returns a table with columns `query_nr` and `query`.

Listing Expected Answers

To produce the expected results for all queries on scale factors 0.01, 0.1 and 1, run:

```
FROM tpch_answers();
```

This function returns a table with columns `query_nr`, `scale_factor` and `answer`.

Generating the Schema

It's possible to generate the schema of TPC-H without any data by setting the scale factor to 0:

```
CALL dbgen(sf = 0);
```

Data Generator Parameters

The data generator function dbgen has the following parameters:

Name	Type	Description
`catalog`	`VARCHAR`	Target catalog
`children`	`UINTEGER`	Number of partitions
`overwrite`	`BOOLEAN`	(Not used)
`sf`	`DOUBLE`	Scale factor
`step`	`UINTEGER`	Defines the partition to be generated, indexed from 0 to `children` - 1. Must be defined when the `children` arguments is defined
`suffix`	`VARCHAR`	Append the `suffix` to table names

Pre-Generated Datasets

Pre-generated DuckDB databases for TPC-H are available for download:

- `tpch-sf1.db` (250 MB)
- `tpch-sf3.db` (754 MB)
- `tpch-sf10.db` (2.5 GB)
- `tpch-sf30.db` (7.6 GB)
- `tpch-sf100.db` (26 GB)
- `tpch-sf300.db` (78 GB)
- `tpch-sf1000.db` (265 GB)
- `tpch-sf3000.db` (796 GB)

> **Warning.** Note that the TPC-H databases generated by the `tpch` extension have small differences from the official TPC-H specification. To obtain fully TPC-compliant datasets, use the `tpchgen-cli` project.

Limitation

The `tpch(<query_id>){:.language-sql .highlight}` function runs a fixed TPC-H query with pre-defined bind parameters (a.k.a. substitution parameters). It is not possible to change the query parameters using the `tpch` extension. To run the queries with the parameters prescribed by the TPC-H benchmark, use a TPC-H framework implementation.

UI Extension

The `ui` extension adds a user interface for your local DuckDB instance.

Usage

To start the UI from the command line:

```
duckdb -ui
```

To start the UI from SQL:

```
CALL start_ui();
```

Running either of these will open the UI in your default browser.

The UI connects to the DuckDB instance it was started from, so any data you've already loaded will be available. Since this instance is a native process (not Wasm), it can leverage all the resources of your local environment: all cores, memory and files. Closing this instance will cause the UI to stop working.

The UI is served from an HTTP server embedded in DuckDB. To start this server without launching the browser, run:

```
CALL start_ui_server();
```

You can then load the UI in your browser by navigating to `http://localhost:4213`.

To stop the HTTP server, run:

```
CALL stop_ui_server();
```

Local Query Execution

By default, the DuckDB UI runs your queries fully locally: your queries and data never leave your computer. If you would like to use MotherDuck through the UI, you have to opt-in explicitly and sign into MotherDuck.

Configuration

Local Port

The local port of the HTTP server can be configured with a SQL command like:

```
SET ui_local_port = 4213;
```

The environment variable `ui_local_port` can also be used.

The default port is 4213. (Why? 4 = D, 21 = U, 3 = C)

Remote URL

The local HTTP server fetches the files for the UI from a remote HTTP server so they can be kept up-to-date.

The default URL for the remote server is https://ui.duckdb.org.

An alternate remote URL can be configured with a SQL command like:

```
SET ui_remote_url = 'https://ui.duckdb.org';
```

The environment variable `ui_remote_url` can also be used.

This setting is available mainly for testing purposes.

Be sure you trust any URL you configure, as the application can access the data you load into DuckDB.

Because of this risk, the setting is only respected if `allow_unsigned_extensions` is enabled.

Polling Interval

The UI extension polls for some information on a background thread. It watches for changes to the list of attached databases, and it detects when you connect to MotherDuck.

These checks take very little time to complete, so the default polling interval is short (284 milliseconds). You can configure it with a SQL command like:

```
SET ui_polling_interval = 284;
```

The environment variable `ui_polling_interval` can also be used.

Setting the polling interval to 0 will disable polling entirely. This is not recommended, as the list of databases in the UI could get out of date, and some ways of connecting to MotherDuck will not work properly.

Tips

Opening a CSV File with the DuckDB UI

Using the DuckDB CLI client, you can start the UI with a CSV available as a view using the `-cmd` argument:

```
duckdb -cmd "CREATE VIEW <view_name> AS FROM '<filename>.csv';" -ui
```

Running the UI in Read-Only Mode

The DuckDB UI uses DuckDB tables as storage internally (e.g., for saving notebooks). Therefore, running the UI directly on a read-only database is not supported:

```
duckdb -ui -readonly read_only_test.db
```

In the UI, this results in:

```
Catalog Error: SET schema: No catalog + schema named "memory.main" found.
```

To work around this, run the UI on another database file:

```
duckdb -ui ui_catalog.db
```

Then, open a notebook and attach to the database:

```
ATTACH 'test.db' (READ_ONLY) AS my_db;
USE my_db;
```

Notes

The UI is built and maintained by MotherDuck. An overview of the UI's features can be found in the MotherDuck documentation.

Limitations

- The UI currently does not support `osx_amd64` and `windows_arm64`.

Unity Catalog Extension

The `unity_catalog` extension attaches a Unity Catalog catalog, on Databricks or Unity Catalog OSS, as a DuckDB database. You can list, read, insert into and create its Delta Lake tables. It uses the `delta` extension to access the tables.

For implementation details, see the "Delta Grows Up: Writes, Unity Catalog and Time Travel" blog post.

> **Note.** Both extensions are available on a subset of platforms.

Installing and Loading

The `unity_catalog` extension will be transparently autoloaded on first use from the official extension repository. It loads the `delta` extension itself. If you would like to install and load it manually, run:

```
INSTALL AND LOAD unity_catalog;
```

Usage

To connect to Unity Catalog on Databricks or Unity Catalog OSS, create a secret with your token, endpoint and region, then attach a catalog:

```
CREATE SECRET (
    TYPE unity_catalog,
    TOKEN '<token>',
    ENDPOINT '<endpoint>',
    AWS_REGION '<region>'
);
ATTACH '<catalog_name>' AS my_catalog (TYPE unity_catalog, DEFAULT_SCHEMA 'main');
```

- `ENDPOINT` is the Unity Catalog REST API endpoint. For Databricks, it is your Workspace URL, typically `https://<instance>.cloud.databricks.com/{:.language-sql .highlight}`.
- `TOKEN` is a credential. For Databricks, it can be a personal access token with `unity-catalog` scope; see Access Control in Unity Catalog for the other options. For Unity Catalog OSS, see the OSS Unity Catalog documentation.
- `AWS_REGION` is the region of the storage that holds the tables. It is required.

To read and write the table files, DuckDB also needs a secret for that storage, e.g., an S3 secret or an Azure secret.

> **Note.** Only Databricks reports a default namespace. When you attach a Unity Catalog OSS catalog, always set `DEFAULT_SCHEMA`: without it, queries against the catalog fail, even with fully qualified table names.

Attach Options

When attaching a Unity Catalog catalog, you can pass the following options to `ATTACH`:

Option	Type	Default	Description
SECRET	VARCHAR	unnamed secret	Name of the `unity_catalog` secret to use. Without it, the unnamed `unity_catalog` secret is used.
DEFAULT_SCHEMA	VARCHAR	detected	Schema that unqualified table names resolve to. Without it, DuckDB asks the server for the workspace's default namespace.
READ_ONLY	BOOLEAN	`false`	Attach the catalog in read-only mode.
USE_IRC_SCAN_PLAN	BOOLEAN	`false`	Plan scans on the server (Databricks only); see Server-Side Scan Planning.

Reading

Tables are named `<catalog>.<schema>.<table>`{:.language-sql .highlight}; names without a schema resolve to `DEFAULT_SCHEMA`:

```
SHOW ALL TABLES;
SELECT * FROM my_catalog.my_schema.my_table LIMIT 10;
```

Time Travel

> New Time travel by timestamp was introduced in DuckDB 2.0.

To read a table as of a version or a point in time, use the `AT` clause:

```
SELECT * FROM my_catalog.my_schema.my_table AT (VERSION => 5);
SELECT * FROM my_catalog.my_schema.my_table AT (TIMESTAMP => TIMESTAMPTZ '2026-10-01 12:00:00+00');
```

A timestamp reads the latest version committed at or before that time. A timestamp before the table's first commit or later than the current time fails with an error.

Writing

To append rows, use `INSERT INTO`:

```
INSERT INTO my_catalog.my_schema.my_table VALUES (1, 'hello');
INSERT INTO my_catalog.my_schema.my_table SELECT * FROM other_table;
```

Data Types

Unity Catalog column types map to DuckDB types as follows. Since v1.5.6, DuckDB reads them from the table's Delta log.

- `timestamp` maps to `TIMESTAMP WITH TIME ZONE` and, since v1.5.6, `timestamp_ntz` to `TIMESTAMP`.
- Since v1.5.6, a column of a type that DuckDB cannot read is listed with the type `UNKNOWN`, e.g., in `DESCRIBE`. Reading the table fails with an error that names the column.

VARIANT Columns

> New Support for VARIANT columns was introduced in DuckDB 2.0.

Unity Catalog's `variant` maps to `VARIANT`. `VARIANT` columns can be read whether their files are shredded or not. Inserts write shredded variants if the table sets `delta.enableVariantShredding`. An `INSERT` into a table with a `VARIANT` inside a struct, list or map fails with an error.

Creating Tables

> New This feature was introduced in DuckDB 2.0.

`CREATE TABLE` in a Unity Catalog schema writes the first version of a new Delta table and registers the table with Unity Catalog.

Without a location, the table is a managed table: Unity Catalog chooses where it is stored, and the table uses catalog-managed commits:

```
CREATE TABLE my_catalog.my_schema.my_table (id INTEGER, name VARCHAR);
```

With the `location` option, the table is an external table at that location. The location can be any expression that evaluates to an absolute path or a URI:

```
CREATE TABLE my_catalog.my_schema.my_external_table (id INTEGER, name VARCHAR)
WITH (location = 's3://<my_bucket>/<path>');
```

Both managed and external tables can be partitioned:

```
CREATE TABLE my_catalog.my_schema.my_partitioned_table (id INTEGER, part VARCHAR)
PARTITIONED BY (part);
```

`CREATE TABLE IF NOT EXISTS` does nothing if Unity Catalog already has a table with this name.

For the forms and options that `CREATE TABLE` does not support, see Limitations.

> **Warning.** Unity Catalog registers the table while `CREATE TABLE` runs, so a `ROLLBACK` of the surrounding transaction does not remove the table.

Checkpointing

Since v1.5.1, `unity_catalog_checkpoint_table` compacts the Delta log of a table into a checkpoint file, which speeds up reads of tables with long logs. Pass the table name as a string, either fully qualified or relative to the catalog and schema set with `USE`:

```
CALL unity_catalog_checkpoint_table('my_catalog.my_schema.my_table');
```

The `CHECKPOINT` statement does not apply to Unity Catalog tables and fails with an error.

Server-Side Scan Planning

> New This feature was introduced in DuckDB 2.0.

Databricks can plan scans on the server, through the Iceberg REST Catalog (IRC) API of Unity Catalog. The API uses catalog-wide statistics, which can improve filtering and reduce the number of files that a query scans. Unity Catalog OSS does not offer this API.

With `USE_IRC_SCAN_PLAN true`, DuckDB sends each scan's filters to the server and reads only the files that it returns, with their delete files. DuckDB still applies the filters itself:

```
ATTACH '<catalog_name>' AS my_catalog (
    TYPE unity_catalog,
    DEFAULT_SCHEMA 'main',
    USE_IRC_SCAN_PLAN true
);
```

- If the server cannot plan a scan, e.g., because it does not offer the API or refuses the table, DuckDB reads the table through its Delta log instead, and does not try scan planning again on this attached catalog for 15 minutes.
- Time travel queries (`AT (...)`) always read through the Delta log.
- Positional delete files and deletion vectors are applied. A table with equality delete files fails with an error; to read it, attach the catalog without `USE_IRC_SCAN_PLAN`.

Catalog-Managed Commits

Managed tables in Databricks and in Unity Catalog OSS may use catalog-managed commits (Catalog-Coordinated Commits / CCv2), where Unity Catalog coordinates commits rather than writers committing directly to the Delta log. Since v1.5.2 (v1.5.5 for Unity Catalog OSS), DuckDB uses this protocol when the attached table requires it. `INSERT` statements are identical in syntax, but they commit through the catalog-managed write protocol, so Unity Catalog coordinates concurrent writers. For details, see the "Delta Grows Up: Writes, Unity Catalog and Time Travel" blog post.

A managed table that DuckDB creates uses catalog-managed commits. Tables created by other clients, e.g., Spark or the Unity Catalog CLI, are read and written the same way.

Features

This extension supports:

- Listing available tables: `SHOW ALL TABLES;`{:.language-sql .highlight}
- Interacting with tables using standard SQL: `SELECT * FROM <catalog>.<schema>.<table>;`{:.language-sql .highlight}
- Time travel: `SELECT * FROM ... AT (VERSION => ...);`{:.language-sql .highlight} and `SELECT * FROM ... AT (TIMESTAMP => ...);`{:.language-sql .highlight}
- Inserts: `INSERT INTO ... VALUES (...);`{:.language-sql .highlight}
- Creating managed and external tables: `CREATE TABLE ... (...);`{:.language-sql .highlight}
- Checkpointing individual tables (since v1.5.1): `CALL unity_catalog_checkpoint_table('my_catalog.my_schema.my_table');`{:.language-sql .highlight}

Limitations

- `DELETE` and `UPDATE` are not supported.
- `DROP TABLE` and `ALTER TABLE` are not supported, nor is creating or changing views or schemas.
- Inserts into a table with column `DEFAULT` values or generated columns fail with an error, as DuckDB cannot compute these columns yet.
- Reading a table with an `INTERVAL` column fails with an error, as Delta intervals and DuckDB's `INTERVAL` differ in meaning.
- The following `CREATE TABLE` forms and options are not supported and fail with an error:
  - `CREATE OR REPLACE TABLE` and `CREATE TABLE ... AS SELECT`
  - constraints, including `NOT NULL` and `PRIMARY KEY`
  - column `DEFAULT` values and generated columns
  - columns of a type that Delta cannot store, e.g., `TIME`
  - `SORTED BY`
  - `WITH` options other than `location`: unlike with the `delta` extension, Delta table properties cannot be set

Supported Platforms

The `unity_catalog` and `delta` extensions support the following platforms:

- Linux (x86_64 and ARM64): `linux_amd64` and `linux_arm64`
- macOS Intel and Apple Silicon: `osx_amd64` and `osx_arm64`

- Windows AMD64: `windows_amd64`

Support for the other DuckDB platforms is work-in-progress.

Vortex Extension

The `vortex` extension allows you to read and write files using the Vortex file format. It is currently available for the Linux (`linux_amd64`, `linux_arm64`) and macOS (`osx_arm64`) distributions.

Installing and Loading

To install and load the extension, run:

```
INSTALL AND LOAD vortex;
```

Reading Vortex Files

Using the `read_vortex` function to read Vortex files:

```
SELECT * FROM read_vortex('my.vortex');
```

i
int64
0
1
2

Writing Vortex Files

You can write Vortex files as follows:

```
COPY (SELECT * FROM generate_series(0, 3) t(i))
TO 'my.vortex' (FORMAT vortex);
```

> **Warning.** Make sure to add the `FORMAT vortex` option. If the `vortex` extension is not loaded, using `COPY ... TO 'my.vortex` without the `FORMAT vortex` specifier will result in a CSV file.

Vector Similarity Search Extension

The `vss` extension is an experimental extension for DuckDB that adds indexing support to accelerate vector similarity search queries using DuckDB's new fixed-size ARRAY type.

See the announcement blog post and the "What's New in the Vector Similarity Search Extension?" post.

Usage

To create a new HNSW (Hierarchical Navigable Small Worlds) index on a table with an `ARRAY` column, use the `CREATE INDEX` statement with the `USING HNSW` clause. For example:

```
INSTALL AND LOAD vss;

CREATE TABLE my_vector_table (vec FLOAT[3]);
INSERT INTO my_vector_table
    SELECT array_value(a, b, c)
    FROM range(1, 10) ra(a), range(1, 10) rb(b), range(1, 10) rc(c);
CREATE INDEX my_hnsw_index ON my_vector_table USING HNSW (vec);
```

The index will then be used to accelerate queries that use an `ORDER BY` clause evaluating one of the supported distance metric functions against the indexed columns and a constant vector, followed by a `LIMIT` clause. For example:

```
SELECT *
FROM my_vector_table
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```

Additionally, the overloaded `min_by(col, arg, n)` can also be accelerated with the HNSW index if the `arg` argument is a matching distance metric function. This can be used to do quick one-shot nearest neighbor searches. For example, to get the top 3 rows with the closest vectors to `[1, 2, 3]`:

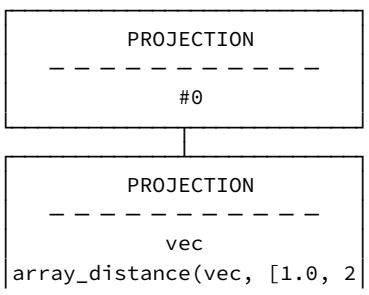
```
SELECT min_by(my_vector_table, array_distance(vec, [1, 2, 3]::FLOAT[3]), 3 ORDER BY vec) AS result
FROM my_vector_table;
```

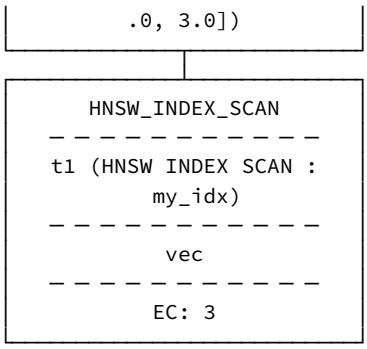
```
[{'vec': [1.0, 2.0, 3.0]}, {'vec': [2.0, 2.0, 3.0]}, {'vec': [1.0, 2.0, 4.0]}]
```

Note how we pass the table name as the first argument to `min_by` to return a struct containing the entire matched row.

We can verify that the index is being used by checking the `EXPLAIN` output and looking for the `HNSW_INDEX_SCAN` node in the plan:

```
EXPLAIN
SELECT *
FROM my_vector_table
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```





By default the HNSW index will be created using the euclidean distance `l2sq` (L2-norm squared) metric, matching DuckDBs `array_distance` function, but other distance metrics can be used by specifying the `metric` option during index creation. For example:

```
CREATE INDEX my_hnsw_cosine_index
ON my_vector_table
USING HNSW (vec)
WITH (metric = 'cosine');
```

The following table shows the supported distance metrics and their corresponding DuckDB functions

Metric	Function	Description
`l2sq`	`array_distance`	Euclidean distance
`cosine`	`array_cosine_distance`	Cosine similarity distance
`ip`	`array_negative_inner_product`	Negative inner product

Note that while each HNSW index only applies to a single column you can create multiple HNSW indexes on the same table each individually indexing a different column. Additionally, you can also create multiple HNSW indexes to the same column, each supporting a different distance metric.

Index Options

Besides the `metric` option, the HNSW index creation statement also supports the following options to control the hyperparameters of the index construction and search process:

Option	Default	Description
`ef_construction`	128	The number of candidate vertices to consider during the construction of the index. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.
`ef_search`	64	The number of candidate vertices to consider during the search phase of the index. A higher value will result in a more accurate index, but will also increase the time it takes to perform a search.
`M`	16	The maximum number of neighbors to keep for each vertex in the graph. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.
`M0`	`2 * M`	The base connectivity, or the number of neighbors to keep for each vertex in the zero-th level of the graph. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.

Additionally, you can also override the `ef_search` parameter set at index construction time by setting the `SET hnsw_ef_search = <int>`{:.language-sql .highlight} configuration option at runtime. This can be useful if you want to trade search performance for accuracy or vice-versa on a per-connection basis. You can also unset the override by calling `RESET hnsw_ef_search`{:.language-sql .highlight}.

Persistence

Due to some known issues related to persistence of custom extension indexes, the HNSW index can only be created on tables in in-memory databases by default, unless the `SET hnsw_enable_experimental_persistence = <bool>`{:.language-sql .highlight} configuration option is set to `true`.

The reasoning for locking this feature behind an experimental flag is that "WAL" recovery is not yet properly implemented for custom indexes, meaning that if a crash occurs or the database is shut down unexpectedly while there are uncommitted changes to a HNSW-indexed table, you can end up with **data loss or corruption of the index**.

If you enable this option and experience an unexpected shutdown, you can try to recover the index by first starting DuckDB separately, loading the `vss` extension and then ATTACHing the database file, which ensures that the HNSW index functionality is available during WAL-playback, allowing DuckDB's recovery process to proceed without issues. But we still recommend that you do not use this feature in production environments.

With the `hnsw_enable_experimental_persistence` option enabled, the index will be persisted into the DuckDB database file (if you run DuckDB with a disk-backed database file), which means that after a database restart, the index can be loaded back into memory from disk instead of having to be re-created. With that in mind, there are no incremental updates to persistent index storage, so every time DuckDB performs a checkpoint the entire index will be serialized to disk and overwrite itself. Similarly, after a restart of the database, the index will be deserialized back into main memory in its entirety. Although this will be deferred until you first access the table associated with the index. Depending on how large the index is, the deserialization process may take some time, but it should still be faster than simply dropping and re-creating the index.

Inserts, Updates, Deletes and Re-Compaction

The HNSW index does support inserting, updating and deleting rows from the table after index creation. However, there are two things to keep in mind:

- It's faster to create the index after the table has been populated with data as the initial bulk load can make better use of parallelism on large tables.
- Deletes are not immediately reflected in the index, but are instead "marked" as deleted, which can cause the index to grow stale over time and negatively impact query quality and performance.

To remedy the last point, you can call the `PRAGMA hnsw_compact_index('<index_name>')`{:.language-sql .highlight} pragma function to trigger a re-compaction of the index pruning deleted items, or re-create the index after a significant number of updates.

Bonus: Vector Similarity Search Joins

The `vss` extension also provides a couple of table macros to simplify matching multiple vectors against each other, so called "fuzzy joins". These are:

- `vss_join(left_table, right_table, left_col, right_col, k, metric := 'l2sq')`
- `vss_match(right_table", left_col, right_col, k, metric := 'l2sq')`

These **do not** currently make use of the HNSW index but are provided as convenience utility functions for users who are ok with performing brute-force vector similarity searches without having to write out the join logic themselves. In the future these might become targets for index-based optimizations as well.

These functions can be used as follows:

```
CREATE TABLE haystack (id int, vec FLOAT[3]);
CREATE TABLE needle (search_vec FLOAT[3]);

INSERT INTO haystack
    SELECT row_number() OVER (), array_value(a, b, c)
    FROM range(1, 10) ra(a), range(1, 10) rb(b), range(1, 10) rc(c);
```

```sql
INSERT INTO needle
    VALUES ([5, 5, 5]), ([1, 1, 1]);

SELECT *
FROM vss_join(needle, haystack, search_vec, vec, 3) res;
```

score float	left_tbl struct(search_vec float[3])	right_tbl struct(id integer, vec float[3])
0.0	{'search_vec': [5.0, 5.0, 5.0]}	{'id': 365, 'vec': [5.0, 5.0, 5.0]}
1.0	{'search_vec': [5.0, 5.0, 5.0]}	{'id': 364, 'vec': [5.0, 4.0, 5.0]}
1.0	{'search_vec': [5.0, 5.0, 5.0]}	{'id': 356, 'vec': [4.0, 5.0, 5.0]}
0.0	{'search_vec': [1.0, 1.0, 1.0]}	{'id': 1, 'vec': [1.0, 1.0, 1.0]}
1.0	{'search_vec': [1.0, 1.0, 1.0]}	{'id': 10, 'vec': [2.0, 1.0, 1.0]}
1.0	{'search_vec': [1.0, 1.0, 1.0]}	{'id': 2, 'vec': [1.0, 2.0, 1.0]}

Alternatively, we can use the `vss_match` macro as a "lateral join" to get the matches already grouped by the left table. Note that this requires us to specify the left table first, and then the `vss_match` macro which references the search column from the left table (in this case, `search_vec`):

```sql
SELECT *
FROM needle, vss_match(haystack, search_vec, vec, 3) res;
```

search_vec float[3]	matches struct(score float, "row" struct(id integer, vec float[3]))[]
[5.0, 5.0, 5.0]	[{'score': 0.0, 'row': {'id': 365, 'vec': [5.0, 5.0, 5.0]}}, {'score': 1.0, 'row': {'id': 364, 'vec': [5.0, 4.0, 5.0]}}, {'score': 1.0, 'row': {'id': 356, 'vec': [4.0, 5.0, 5.0]}}]
[1.0, 1.0, 1.0]	[{'score': 0.0, 'row': {'id': 1, 'vec': [1.0, 1.0, 1.0]}}, {'score': 1.0, 'row': {'id': 10, 'vec': [2.0, 1.0, 1.0]}}, {'score': 1.0, 'row': {'id': 2, 'vec': [1.0, 2.0, 1.0]}}]

Limitations

- Only vectors consisting of `FLOAT`s (32-bit, single precision) are supported at the moment.
- The index itself is not buffer managed and must be able to fit into RAM memory.
- The size of the index in memory does not count towards DuckDB's `memory_limit` configuration parameter.
- HNSW indexes can only be created on tables in in-memory databases, unless the `SET hnsw_enable_experimental_persistence = <bool>`{:.language-sql .highlight} configuration option is set to `true`, see Persistence for more information.
- The vector join table macros (`vss_join` and `vss_match`) do not require or make use of the HNSW index.

Quack Remote Protocol

Quack Remote Protocol

The Quack extension turns a DuckDB instance into a server that other DuckDB instances (clients) can connect to over HTTP.

This page covers the protocol at a glance and walks through basic usage on both sides of the wire. For the full list of functions, settings, and logging knobs, see the Reference. For configuring TLS and authentication / authorization, see Security. For user guides, see Guides.

Quack in a Nutshell

In short, the Quack protocol and its interactions work as follows:

- **HTTP-based.** Quack messages travel over plain HTTP or HTTPS via a reverse proxy (see Security). This means standard load balancers, firewalls, and reverse proxies handle Quack traffic the same way they handle any other HTTP service. There is no custom wire transport to operate.
- **Client-driven request and response.** Every interaction is initiated by the client. The server does not initiate interactions via pushing.
- **`application/duckdb` serialization.** Requests and responses are encoded with DuckDB's internal serialization primitives (the same code path used by the Write-Ahead Log). This avoids round-tripping data through an interchange format and keeps complex types (nested, decimals, intervals, ...) lossless across the wire.
- **Single round-trip per query.** After the initial connection handshake, a query needs only one request–response pair. Large results stream back in chunks via follow-up `FETCH` requests, optionally parallelized on multiple threads.
- **Default port: `9494`.** All URIs use the `quack:` scheme, e.g., `quack:hostname:port` with the port defaulting to `9494`.

Server-Side Usage

Starting a Server

A server is started from an existing DuckDB session. Everything that the session can see (in-memory tables, attached files, schemas) becomes reachable over the remote protocol.

To start listening on localhost, run:

{:.codebox-server}

```
CALL quack_serve('quack:localhost');
```

The URI argument is optional: calling `quack_serve()` without it also listens on `quack:localhost`.

`quack_serve` returns the listen URI, the HTTP URL, and, when the default authentication function is in use, the `auth_token` that clients need to connect. To choose the token yourself, pass the `token` parameter (see Security):

{:.codebox-server}

```
CALL quack_serve('quack:localhost', token = '<my_token>');
```

> **Tip.** The DuckDB CLI can start a server directly with the `-serve` argument and connect to one with `-connect`. See Command Line Arguments.

By default the server refuses to bind anything other than a local hostname. To listen on an externally-reachable address, pass `allow_other_hostname = true`:

{:.codebox-server}

```
CALL quack_serve('quack:0.0.0.0:9494', allow_other_hostname = true);
```

Non-local servers serve HTTPS by default, using a self-signed certificate unless you pass `ssl_cert_file` and `ssl_key_file` (see the Reference). For production deployments, we recommend binding the server to localhost and fronting it with a TLS-terminating reverse proxy. See Securing Quack with a Reverse Proxy.

URI Format

Quack endpoints use the `quack:` URI scheme and `9494` as the default port. Some examples:

URI	Host	Port	Comment
`quack:localhost`	`localhost`	`9494`	
`quack://localhost`	`localhost`	`9494`	
`quack:myhost:9000`	`myhost`	`9000`	
`quack:127.0.0.1`	`127.0.0.1`	`9494`	
`quack:[::1]:1234`	`::1`	`1234`	(IPv6)

You can parse and validate a URI with the `quack_uri_parser(uri, ssl)` scalar function.

Stopping a Server

To stop a server, run:

{:.codebox-server}

```
CALL quack_stop('quack:localhost');
```

Client-Side Usage

There are three ways to talk to a Quack server:

1. `ATTACH 'quack:host' AS name` followed by `CONNECT name`: route the whole session to the remote. This is the recommended approach.
2. `ATTACH 'quack:host' AS name`: attach the remote as a full catalog and query it by qualifying table names.
3. `quack_query(uri, query)`: stateless querying.

In all cases, you need to authenticate.

The client picks plain HTTP automatically for local URIs (`localhost`, `127.0.0.1`, `::1`) and HTTPS otherwise. Either default can be overridden with the `DISABLE_SSL` configuration option.

Stateless Queries with `quack_query`

You can run any SQL against a server without attaching it. To query a local database via HTTP, run:

{:.codebox-client}

```
FROM quack_query(
    'quack:localhost',
    'SELECT 42',
    token = '<MY_QUACK_TOKEN_01234567890ABCDEF>');
```

Remote databases use HTTPS by default. To override on a plain-HTTP remote, run:

{:.codebox-client}

```
FROM quack_query(
    'quack:remote.com',
    'SELECT 42',
    token = '<MY_QUACK_TOKEN_01234567890ABCDEF>',
    disable_ssl = true
);
```

The query executes remotely and the server streams the result back. Errors occurring on the server (parse errors, missing tables, etc.) are shown locally in the DuckDB client.

Attaching a Remote Database

To attach a local database via HTTP, simply run:

{:.codebox-client}

```
ATTACH 'quack:localhost' AS remote_db (
    TOKEN '<MY_QUACK_TOKEN_01234567890ABCDEF>'
);
```

Attaching remote databases uses HTTPS by default. To override on a plain-HTTP remote:

{:.codebox-client}

```
ATTACH 'quack:remote.com' AS remote_db (
    TOKEN '<MY_QUACK_TOKEN_01234567890ABCDEF>',
    DISABLE_SSL true
);
```

Once attached, remote tables look and behave like local ones:

{:.codebox-client}

```
CREATE TABLE remote_db.t AS FROM range(10) r(i);  -- DDL on remote
INSERT INTO remote_db.t VALUES (42);              -- remote writes
```

You can run queries against the remote database:

{:.codebox-client}

```
FROM remote_db.t;              -- scan remote table
FROM remote_db.t WHERE i = 42; -- run filter remotely
BEGIN; ...; COMMIT;            -- transactions are forwarded
DETACH remote_db;              -- detach from the remote database
```

The attached catalog also exposes a `query` table macro for ad-hoc SQL scoped to that attachment:

{:.codebox-client}

```
FROM remote_db.query('SELECT 42');
```

> ATTACH binds the remote's table definitions on the client, so a single table definition that cannot be bound fails the whole attachment. If this happens (for example, with an Iceberg catalog using lazy table resolution), see ATTACH Fails to Bind a Remote Table Definition.

Routing a Session with CONNECT

Once a remote is attached, the CONNECT statement routes the whole session to it, so that subsequent queries execute on the server and their results stream back. This is the recommended way to work with a remote Quack database: you write plain SQL without qualifying every table name with the attached catalog. `DISCONNECT` returns the session to the local database.

{:.codebox-client}

```sql
ATTACH 'quack:server.example.com' AS qk (TOKEN '<MY_QUACK_TOKEN_01234567890ABCDEF>');

CONNECT qk;
SELECT count(*) FROM events; -- executes on the server, results stream back
DISCONNECT;
```

CONNECT is not specific to Quack: it points the session at any remote database that supports remote execution. See the CONNECT and DISCONNECT statements page for details.

Authentication

Clients pass the authentication token to the server in one of two ways: a `quack` secret scoped to the server URI or an explicit TOKEN option on ATTACH / `quack_query`. See Security for the full picture.

We recommend using a secret scoped to the server URI:

{:.codebox-client}

```sql
CREATE SECRET (
    TYPE quack,
    TOKEN '<MY_QUACK_TOKEN_01234567890ABCDEF>',
    SCOPE 'quack:localhost'
);

ATTACH 'quack:localhost' AS remote_db (TYPE quack);
```

Alternatively, you can pass the token directly, which overrides any matching secret:

{:.codebox-client}

```sql
ATTACH 'quack:localhost' AS remote_db (
    TOKEN '<MY_QUACK_TOKEN_01234567890ABCDEF>'
);
```

HTTP Connection Caching

Quack clients send requests through the `httpfs` HTTP client, which caches connections by default (`httpfs_connection_caching = true`). Reusing connections avoids a new TCP and, with SSL, a costly TLS handshake per request. To open a fresh connection for every request (e.g., for debugging), disable caching:

{:.codebox-client}

```sql
SET httpfs_connection_caching = false;
```

Node Identity (`whoami`)

Each Quack node exposes a `whoami()` table macro that surfaces basic identity and runtime info, useful when proxying to a fleet of servers or when correlating logs:

{:.codebox-client}

```sql
FROM remote_db.query('FROM whoami()');
```

```
│  name   │ provider │ hostname │ region  │     uptime      │           ts_now              │
meta                                      │
│ varchar │ varchar  │ varchar  │ varchar │    interval     │   timestamp with time zone    │
json                                      │
│ NULL    │ NULL     │ NULL     │ NULL    │ 00:04:56.832456 │ 2026-10-21 15:59:38.631715+02 │ {"duckdb_
version":"v2.0.0","platform":"osx_arm64"} │
```

Identity fields are populated either by setting `whoami_*` options directly or by calling the `quack_identify` helper:

{:.codebox-client}

```
CALL quack_identify(
    name = 'analytics-1',
    provider = 'ec2',
    region = 'eu-west-1',
    meta = '{"role": "worker"}'
);
```

`meta` is merged with auto-computed `duckdb_version` and `platform` keys; user-supplied keys win on conflict. `whoami_started_at` (an ISO 8601 timestamp) overrides the uptime anchor, otherwise uptime is measured from extension load.

Reference

This page lists the functions, settings, and log types exposed by the quack extension. For a tour of the protocol, start with the Overview.

Function Reference

Server Management

Function	Description
`quack_serve(<listen_uri>, ...)`	Start a server on `listen_uri`, which defaults to `quack:localhost` when omitted. Localhost-only by default. Returns the listen URI, the listen URL, and the authentication token. See the `quack_serve` parameters below.
`quack_stop(listen_uri)`	Stop the server listening on `listen_uri`.
`quack_server_list()`	List the servers running in this instance: listen URI and URL, host, port, number of active connections, and additional info (IPv6, SSL).
`quack_active_connections()`	List the client connections of the servers running in this instance: server, `quack_connection_id`, last query, state, hashed client id, cached rows, and query start time.
`quack_generate_keys(directory = <path>)`	Generate a self-signed certificate (`server.pem`) and private key (`private_key.pem`) for serving over SSL. Defaults to `~/.duckdb/extension_data/quack`; existing key files are not overwritten. Returns the file paths and the certificate fingerprint.
`quack_identify(name, provider, hostname, region, meta)`	Set this node's `whoami` identity fields. Any subset can be supplied.
`whoami()`	Table macro returning identity + runtime info for the current node.

`quack_serve` Parameters

Parameter	Type	Default	Description
`listen_uri`	VARCHAR	`quack:localhost`	Positional. The Quack URI to listen on.
`allow_other_hostname`	BOOLEAN	`false`	Allow binding to a non-local hostname, e.g., `0.0.0.0`.
`token`	VARCHAR	(generated)	Authentication token clients have to present (minimum 4 characters). If omitted, the token of the default `quack` secret is used if one exists, otherwise a random token is generated.
`secret`	VARCHAR	(unset)	Name of an existing `quack` secret whose token the server uses.

Parameter	Type	Default	Description
`create_secret_if_not_exists`	`BOOLEAN`	`false`	If no token is given and no default `quack` secret exists, store the generated token in a persistent default `quack` secret, so that subsequent servers reuse it.
`disable_ssl`	`BOOLEAN`	`true` for local, else `false`	Serve plain HTTP. Local URIs default to plain HTTP, other URIs to HTTPS.
`ssl_cert_file`	`VARCHAR`	(generated)	Certificate file used when serving over SSL. If omitted, a self-signed certificate is generated (see `quack_generate_keys`).
`ssl_key_file`	`VARCHAR`	(generated)	Private key file used when serving over SSL.

Client Queries

Function	Description
`quack_query(uri, query, ...)`	Run `query` on remote `uri`, stream result back. Accepts the named parameters `token`, `disable_ssl`, `ssl_fingerprint`, `client_id` and `heartbeat_timeout`, which behave like the ATTACH options of the same name. Pass `token` to override any matching quack secret on the client side.
`quack_query_by_name(catalog, query)`	Run `query` against an already-attached Quack catalog (used by `<catalog>.query(){:.language-sql .highlight}`). Also accepts the named parameters `use_transaction` and `refresh_catalog` (both `BOOLEAN`).
`quack_connection_id(catalog)`	Return the server-issued `quack_connection_id` of an attached Quack catalog.
`quack_cancel(catalog, connection_id)`	Cancel the running query of the connection `connection_id` on the server of the attached Quack catalog `catalog`. Returns the connection id and whether the cancellation succeeded.

Utility

Function	Description
`quack_uri_parser(uri, ssl)`	Parse a Quack URI into a `STRUCT(host, port, ipv6, ssl, url)` entry.
`quack_check_token(sid, client_token, server_token)`	Default authentication callback, compares the client-supplied token against the server's stored token.
`quack_nop_authorization(sid, query)`	Default authorization callback, always allows.
`quack_clear_cache()`	Clear the cache. This also forces the DuckDB client to refresh its catalog.

ATTACH Options

Option	Type	Default	Description
`TOKEN`	`VARCHAR`	(unset)	Authentication token. Overrides any matching quack secret on the client side.
`DISABLE_SSL`	`BOOLEAN`	`true` for local, else `false`	Force the client transport. Local URIs default to plain HTTP.
`TYPE`	`VARCHAR`	inferred	Pin the secret type used for token resolution (e.g., `quack`).
`SSL_FINGERPRINT`	`VARCHAR`	(unset)	Expected fingerprint of the server's certificate, e.g., the `fingerprint` returned by `quack_generate_keys`. Use this to connect to a server with a self-signed certificate.
`CLIENT_ID`	`VARCHAR`	`quack_default_client_id`	Client identifier sent to the server (minimum 4 characters).
`HEARTBEAT_TIMEOUT`	`UBIGINT`	`quack_default_heartbeat_timeout`	Heartbeat lease timeout in seconds requested from the server.

Settings

All settings are regular DuckDB session / global options. Set with `SET <name> = <value>`{:.language-sql .highlight} or `SET GLOBAL`{:.language-sql .highlight}.

Authentication / Authorization

The auth callbacks are evaluated on a fresh server-side connection every time, so the two settings below are **global-scoped** (`SET GLOBAL`). A plain `SET` on these is forwarded to the global slot automatically. Use `RESET GLOBAL` to restore the default, a plain `RESET` only clears the session view and the auth path will keep reading the stale global value.

Setting	Type	Default	Description
`quack_authentication_function`	`VARCHAR`	`quack_check_token`	Name of a 3-arg scalar function `(sid, client_token, server_token) -> BOOLEAN` used by the server to authenticate clients.
`quack_authorization_function`	`VARCHAR`	`quack_nop_authorization`	Name of a 2-arg scalar function `(sid, query) -> BOOLEAN` (or `-> VARCHAR`, returning the SQL to execute) used by the server to authorize each query.

You can plug in your own auth by creating any scalar function with the expected signature and pointing the setting at it. See Security for examples.

Data Transfer

These settings control how results and inserted data are batched and buffered.

Setting	Type	Default	Description
`quack_target_batch_bytes`	`UBIGINT`	`33554432` (32 MiB)	Target in-memory size of one batch (one wire message payload). Batches are cut when they reach this size.

Setting	Type	Default	Description
quack_prepare_inline_rows	UBIGINT	24576	Rows returned inline in the PREPARE response before the remainder is left to FETCH. Drains whole batches, so it may overshoot to a batch boundary.
quack_fetch_read_ahead	UBIGINT	0	Number of FETCH requests the client keeps in flight ahead of the scan. 0 uses the number of async threads.
quack_rebalance_buffer_bytes	UBIGINT	0	Pending bytes the batch rebalancer buffers before gating non-minimum producers. 0 means automatic, governed by the memory manager.
quack_server_stream_buffer_bytes	UBIGINT	268435456 (256 MiB)	Server-side per-stream byte budget, at most a quarter of memory_limit. FETCH counts serialized payload bytes, INSERT counts decoded chunk allocations. 0 disables the limit.

Server Connections

Setting	Type	Default	Description
quack_server_max_connections	UBIGINT	1024	Maximum number of concurrent connections the server accepts. Beyond this, new connections are refused.
quack_server_keep_alive_timeout	UBIGINT	300	Seconds an idle keep-alive connection is kept open by the server.

Client Identity and Reconnects

Setting	Type	Default	Description
quack_default_client_id	VARCHAR	(random per instance)	Client id used when ATTACH / quack_query omit one. Computed at load time from the QUACK_CLIENT_ID environment variable, or a random per-instance id if the variable is not set. Set to '' to opt out.
quack_default_heartbeat_timeout	UBIGINT	60	Heartbeat lease timeout in seconds requested by clients when ATTACH / quack_query omit heartbeat_timeout.
quack_enable_reconnects	BOOLEAN	false	Enable reconnect support: clients acknowledge results and the server caches the last result until it is acknowledged.
quack_result_ttl	UBIGINT	3600	Seconds an idle cached result is kept before it is dropped (0 = never).
quack_cache_max_rows	UBIGINT	100000	Maximum number of rows the server retains in the result cache (0 = unlimited).

The extension also registers several `quack_debug_*` settings, which are intended for testing the protocol and should not be used in production.

Node Identity

These settings back the `whoami()` macro. `quack_identify(...)` is sugar that updates them.

Setting	Type	Default	Description
`whoami_name`	VARCHAR	(empty)	Human-readable node name.
`whoami_provider`	VARCHAR	(empty)	Deployment provider (`ec2`, `docker`, `local`, ...).
`whoami_hostname`	VARCHAR	(empty)	Network hostname / public address.
`whoami_region`	VARCHAR	(empty)	Deployment region.
`whoami_started_at`	VARCHAR	(empty)	Node start time (ISO 8601 timestamp). Anchors `uptime`.
`whoami_meta`	VARCHAR	`{}`	Provider-specific metadata as JSON.
`quack_loaded_at_us`	BIGINT	Epoch microseconds at extension load	Fallback uptime anchor when `whoami_started_at` is empty.

Logging

Two log types are registered by the extension. Enable them to debug connectivity or measure request timing.

Quack Log

Structured log of every Quack message (both client- and server-side):

```
CALL enable_logging('Quack');
FROM quack_query('quack:localhost', 'SELECT 42');
SELECT * FROM duckdb_logs_parsed('Quack');
```

context_id	scope	connection_id	transaction_id	query_id	thread_id	timestamp	log_type	level	message_type	quack_connection_id	client_query_id	query	server	duration_ms	response_type	error
60	CONNECTION	2	18	18	NULL	2026-05-10 09:06:19.841623+02	Quack	DEBUG	CONNECTION_REQUEST		18	NULL	http://localhost:9494	41	CONNECTION_RESPONSE	NULL
60	CONNECTION	2	18	18	NULL	2026-05-10 09:06:19.842407+02	Quack	DEBUG	PREPARE_REQUEST	091A003553E7E67B6155F3D6BE1FD2E	42		http://localhost:9494	0	PREPARE_RESPONSE	NULL

Fields on each entry:

Field	Description
`message_type`	Request type: `PREPARE_REQUEST`, `FETCH_REQUEST`, etc.
`quack_connection_id`	Server-issued connection id (stable across requests in one `ATTACH`).
`client_query_id`	Monotonic id assigned by the client, correlates client / server logs.
`query`	SQL payload for `PREPARE_REQUEST`s.
`server`	HTTP URL on client-side logs, `NULL` on server-side logs.
`duration_ms`	Round-trip time (client) or handling time (server).
`response_type`	Response type or `ERROR`.
`error`	Error message if the request failed.

To correlate a client request with its server-side handling, join on `(quack_connection_id, client_query_id)`.

HTTP Log

The underlying HTTP transport can be logged separately:

```
CALL enable_logging('HTTP');
FROM quack_query('quack:localhost', 'SELECT 1');
SELECT request.type, request.url, response.status
FROM duckdb_logs_parsed('HTTP');
```

type	url	status
POST	http://localhost:9494/quack	OK_200
POST	http://localhost:9494/quack	OK_200

Requests are POSTs to a `/quack` endpoint.

Persisting Logs for Querying

`duckdb_logs_parsed` reads from DuckDB's in-memory log buffer. For non-trivial sessions you'll want to persist logs:

```
CALL enable_logging(
    'Quack',
    storage = 'file',
    storage_config = {'path': '/tmp/duckdb-rpc-logs'}
);
```

To clear the log between runs, use:

```
CALL truncate_duckdb_logs();
```

To turn logging off, run:

```
CALL disable_logging();
```

Security

This page covers Quack's security posture end to end: what the server exposes, what stays local, the role of a TLS-terminating reverse proxy, and the authentication / authorization callbacks the server runs on every connection and every query.

Exposure Model

A Quack server exposes the full SQL surface of the underlying DuckDB instance, including read and write access to every table the server's session can see. Because of this, the extension ships with conservative defaults that prevent accidental exposure:

- The server requires an **authentication token**, which the client has to supply on every connection. Unless a token is passed explicitly or supplied by a `quack` secret, the server generates a random token at startup.
- The server binds to `localhost` only, non-local hostnames require an explicit `allow_other_hostname = true`.
- The server does **not** use TLS for local URIs, as TLS for localhost communication only adds dependencies for no real benefit. Non-local URIs default to HTTPS with a self-signed certificate (see `quack_serve` parameters).

> **Best practice.** For any deployment beyond local-only, do not expose Quack directly to the internet. We recommend you to put a proven HTTP reverse proxy in front of it and let the proxy terminate TLS.

The Quack client is shipped with these assumptions in mind: for non-local URIs it assumes HTTPS by default, so a properly fronted server "just works" from the client side too. See Securing Quack with a Reverse Proxy for nginx and Caddy recipes (production and local-test).

Authentication and Authorization

For every database call, there are two distinct decisions to be made:

- **Authentication**: *who* is the caller? Establishes identity, usually by having the caller supply a credential (e.g., a token, a password, a client certificate).
- **Authorization**: *may they do this?* Establishes whether an already authenticated caller is allowed to run a particular query, against a particular set of objects.

Quack runs these as two separate hooks: the authentication when a client first connects and the authorization before each query the client wants to issue.

Default Configuration

Both hooks ship with built-in defaults that are suitable for local development and single-user deployments, and each is exposed as an overridable callback for deployments with stricter requirements.

Out of the box, they come with the following configuration:

- **Authentication is token-based.** When you call `quack_serve`, the server uses the token passed via `quack_serve(uri, token = '...')`, otherwise the token of a matching (or the default) `quack` secret, otherwise a newly generated random token, and returns it in the `auth_token` column. Clients have to present this token on every connection, either through a `quack` secret scoped to the server URI or via the explicit `TOKEN` option in the `ATTACH` statement / `quack_query` function call. The default authentication callback compares the client-supplied token against the server's stored token.
- **Authorization is permissive.** The default authorization callback, `quack_nop_authorization`, returns the query text unchanged for every query. No further filtering happens.

Both callbacks can be replaced with user-supplied code, including plain SQL macros. See the examples below.

The Callback Contract

Two settings hold the **name** of the function to call as a hook for authentication / authorization:

Setting	Default	Called when
`quack_authentication_function`	`quack_check_token`	A new client connects (`CONNECTION_REQUEST`).
`quack_authorization_function`	`quack_nop_authorization`	A client issues a query (`PREPARE_REQUEST`).

A request is rejected with `Authentication failed` / `Authorization failed` if the callback returns `false` or `NULL`, or if the callback query fails. Any other result admits the request, so we recommend returning a `BOOLEAN`. The authorization callback may also return a `VARCHAR`: the returned string is then executed instead of the client's query (the default `quack_nop_authorization` returns the query unchanged). Anything resolvable as a function with the matching arity will work: built-in scalar functions, scalar UDFs registered by another extension, or SQL macros. Authentication takes `(VARCHAR, VARCHAR, VARCHAR)`, authorization takes `(VARCHAR, VARCHAR)`.

The callbacks run in a **fresh, transient server-side connection**. That means they can read tables, call other UDFs, and reference extensions, but each invocation starts a new session and cannot rely on session-local state.

Authentication Hook

The server invokes the authentication function by issuing the following SQL statement on every `CONNECTION_REQUEST` call:

```
SELECT <quack_authentication_function>(<session_id>, <client_token>, <server_token>);
```

The arguments are defined as follows:

- `session_id`: Server-generated session id (random 32-char string). Becomes the `quack_connection_id` for that client.
- `client_token`: The token the client sent.
- `server_token`: The token configured on the server (via `quack_serve(token = ...)` or auto-generated).

Overriding Authentication The cleanest way to plug in custom authentication is through using a `MACRO`.

Example: Multi-Token Table Authenticate against a small table of allowed tokens (e.g., one per user):

```
CREATE TABLE quack_tokens (auth_token VARCHAR, user_name VARCHAR);
INSERT INTO quack_tokens VALUES
    ('alice-key-123', 'alice'),
    ('bob-key-456',   'bob');

CREATE MACRO check_token(sid, client_token, server_token) AS (
    EXISTS (SELECT 1 FROM quack_tokens WHERE auth_token = client_token)
);

SET GLOBAL quack_authentication_function = 'check_token';
```

Now any client whose token is in `quack_tokens` is admitted, everyone else is rejected. Adding / removing users is a regular `INSERT` / `DELETE` operation.

Example: Developer Mode (Always Allow) When developing locally in a sandboxed environment, you can consider using "developer mode" authentication, which allows every incoming connection:

```
CREATE MACRO developer_mode_auth(sid, client_token, server_token) AS true;
SET GLOBAL quack_authentication_function = 'developer_mode_auth';
```

Authorization Hook

The server invokes the authorization function by issuing the following SQL statement on every PREPARE_REQUEST call:

```
SELECT <quack_authorization_function>(<connection_id>, <query>);
```

The arguments are defined as follows:

- `connection_id`: The `quack_connection_id` of the calling client (i.e., the same id the authentication hook saw as its `session_id` argument).
- `query`: The full SQL text the client wants to execute.

Overriding Authorization Authorization runs once per PREPARE_REQUEST, with the connection id and the full SQL text. Common shapes:

Example: Read-Only

```
CREATE MACRO read_only(sid, query) AS
    regexp_matches(upper(trim(query)), '^(SELECT|FROM|WITH|EXPLAIN|DESCRIBE|SHOW)\b');

SET GLOBAL quack_authorization_function = 'read_only';
```

For more involved authorization functions, see the Beyond SQL Macros section.

Beyond SQL Macros

SQL macros cover most authentication and authorization cases, but a macro body is restricted to a single expression and cannot execute DML directly: there is no INSERT, UPDATE, or DELETE inside a macro. For policies that need to record every call to a table, maintain in-process state across calls, or otherwise drive imperative logic, register a scalar function via a DuckDB extension instead.

DuckDB extensions can be written in C++ (the primary language) or any language with bindings to DuckDB's C extension API, including Rust, C, and Go. The registered authentication or authorization function must expose the same `(VARCHAR, ...) → BOOLEAN` signature as the SQL macros above. Once the extension is loaded, point `quack_authentication_function` or `quack_authorization_function` at the function name.

> Python UDFs registered through `con.create_function` are scoped to the connection that created them. Quack invokes each callback on a fresh server-side connection, so Python UDFs are not visible at dispatch time and cannot be used as authentication or authorization callbacks. Register the function via a DuckDB extension to make it globally visible.

Example: Read-Only Queriers

A self-contained example: a server that requires per-user tokens and limits each user to read-only queries.

```
CREATE TABLE quack_tokens (auth_token VARCHAR, user_name VARCHAR);
INSERT INTO quack_tokens VALUES ('analytics-team-token', 'analytics');

CREATE MACRO check_token(sid, client_token, server_token) AS (
    EXISTS (SELECT 1 FROM quack_tokens WHERE auth_token = client_token)
);

CREATE MACRO read_only(sid, query) AS (
    regexp_matches(upper(trim(query)), '^(SELECT|FROM|WITH|EXPLAIN)\b')
);

CALL quack_serve('quack:localhost', token = 'analytics-team-token');

SET GLOBAL quack_authentication_function = 'check_token';
SET GLOBAL quack_authorization_function  = 'read_only';
```

A client with the right token now connects (e.g., with `ATTACH 'quack:localhost' AS remote_db (TOKEN 'analytics-team-token')` {:.language-sql .highlight}) and can run `SELECT`s, but `INSERT INTO remote_db.t ...` issued through the standard SQL path will fail at authorization time.

> **Warning.** The prefix regex above is illustrative, not a robust read-only filter: a query like `WITH x AS (SELECT 1) INSERT INTO t SELECT * FROM x` starts with `WITH` yet still mutates data. For production read-only enforcement, attach the database in read-only mode or use an authorization function that inspects the *parsed statement type* rather than the raw query text.

Example: Per-User Access Control List

To implement per-user access control list (ACL), create a custom authentication hook that records `sid → user` pairs so authorization can look up who is asking. Because macros can't write, the recording side has to be a scalar UDF defined e.g. by a custom DuckDB extension. The authorization side can be a macro:

```
-- (populated by the auth UDF when a client connects)
CREATE TABLE quack_sessions (sid VARCHAR PRIMARY KEY, user_name VARCHAR);

-- per-user query allowlist (your own data model)
CREATE TABLE quack_user_acls (user_name VARCHAR, query_kind VARCHAR);

CREATE MACRO acl_check(sid, query) AS (
    EXISTS (
        SELECT 1
        FROM quack_sessions s
        JOIN quack_user_acls a ON a.user_name = s.user_name
        WHERE s.sid = sid
          AND regexp_matches(upper(trim(query)), '^' || a.query_kind || '\b')
    )
);

SET GLOBAL quack_authorization_function = 'acl_check';
```

Setup

Quack Setup

This section contains setup guides for using the Quack Remote Protocol.

Securing Quack with a Reverse Proxy

By default, the Quack server binds to localhost and speaks plain HTTP. See Security for the rationale behind these configurations. For any deployment beyond local-only, the recommended pattern is the same as for any other HTTP-based database / application server: put a proven HTTP reverse proxy in front of it, and let the proxy terminate TLS.

The Quack client cooperates with this: for non-local URIs it assumes HTTPS by default, so a properly fronted server "just works" from the client side too.

This guide walks through three setups, in order of likely usefulness:

1. A **local TLS test setup** with Caddy, so you can exercise the full HTTPS path on your machine.
2. **nginx + Let's Encrypt** in production.
3. **Caddy + Let's Encrypt** in production.

The two production setups are interchangeable. Pick whichever fits your operational stack.

Local Test Setup with Caddy

You can exercise the full HTTPS path on your own machine using Caddy. Caddy issues itself a certificate for `localhost` from a local CA and installs the CA root into your system trust store, so DuckDB's HTTPS client trusts the cert without any extra config.

1. Run Caddy

Save this `Caddyfile`:

```
localhost:8443 {
    reverse_proxy 127.0.0.1:9494 {
        flush_interval -1
    }

    request_body {
        max_size 256MB
    }
}
```

Then start Caddy:

```
brew install caddy # macOS, for other platforms see https://caddyserver.com/docs/install
caddy run --config Caddyfile
```

The first run will prompt for elevation to install Caddy's local CA into your system trust store. After that, certs issued by Caddy for `localhost` are trusted system-wide.

2. Start Quack and Connect Through the Proxy

In one DuckDB session, start the server (this prints an authentication token):

{:.codebox-server}

```
CALL quack_serve('quack:localhost');
```

In the client session, connect through Caddy on `:8443`. Local URIs default to plain HTTP, so you have to **force SSL on** explicitly:

{:.codebox-client}

```
ATTACH 'quack:localhost:8443' AS quack (
    TOKEN '<authentication_token-from-quack_serve>',
    DISABLE_SSL false
);

FROM quack.query('SELECT 42');
```

If the round-trip succeeds, your traffic just went out as TLS to Caddy, got terminated, and was forwarded as plain HTTP to Quack on `:9494`.

Nginx + Let's Encrypt

We expect this to be the most common choice. A minimal site for a Quack server listening on the loopback interface looks as follows:

```
# /etc/nginx/sites-enabled/quack.example.com
server {
    listen 443 ssl http2;
    server_name quack.example.com;

    ssl_certificate     /etc/letsencrypt/live/quack.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/quack.example.com/privkey.pem;

    # Quack RPC bodies can be large: PREPARE carries SQL, APPEND carries
    # inserted DataChunks. The 1 MiB nginx default will fail mid-INSERT.
    client_max_body_size 256M;

    # Long-running queries can sit on the wire between FETCHes for
    # minutes. Raise the timeouts above nginx's default 60s.
    proxy_read_timeout 600s;
    proxy_send_timeout 600s;

    location / {
        proxy_pass http://127.0.0.1:9494;

        # Keep-alive against upstream. Quack relies on persistent
        # connections to keep server-side `quack_connection_id` state alive.
        proxy_http_version 1.1;
        proxy_set_header Connection "";

        # Quack streams results via repeated FETCH responses; buffering
        # through nginx defeats the streaming and inflates memory.
        proxy_buffering off;
    }
}
```

On the Quack side, start the server bound to localhost (the default):

{:.codebox-server}

```
CALL quack_serve('quack:localhost');
```

Issue the certificate with `certbot --nginx -d quack.example.com`. Clients connect over HTTPS automatically. As the default Quack port remains `9494` even with HTTPS, specify the proxy's port `443` explicitly:

{:.codebox-client}

```
ATTACH 'quack:quack.example.com:443' AS quack (
    TOKEN '<authentication_token-from-quack_serve>'
);
```

Caddy + Let's Encrypt

Caddy auto-provisions certificates from Let's Encrypt and needs almost no configuration. A complete public-facing Quack proxy:

```
# /etc/caddy/Caddyfile
quack.example.com {
    reverse_proxy 127.0.0.1:9494 {
        # Equivalent of nginx `proxy_buffering off`. Required so Quack's
        # streamed FETCH responses pass through immediately instead of
        # being buffered in Caddy.
        flush_interval -1
    }

    # Equivalent of nginx `client_max_body_size`. PREPARE / APPEND bodies
    # can be much larger than the default request body cap.
    request_body {
        max_size 256MB
    }
}
```

Caddy handles certificate issuance and renewal automatically. No `certbot` step is required.

On the Quack side, start the server bound to localhost (the default):

{:.codebox-server}

```
CALL quack_serve('quack:localhost');
```

Clients connect over HTTPS automatically. As the default Quack port remains `9494` even with HTTPS, specify the proxy's port `443` explicitly:

{:.codebox-client}

```
ATTACH 'quack:quack.example.com:443' AS quack (
    TOKEN '<authentication_token-from-quack_serve>'
);
```

Deploying Quack

This page collects deployment recipes for running a public-facing Quack server. Today there is one recipe (AWS EC2), we will introduce more over time.

On AWS EC2

The fastest way to get a public-facing Quack server is the one-click AWS CloudFormation template maintained alongside the extension. It provisions a small EC2 instance running DuckDB, the quack extension behind nginx and Let's Encrypt TLS. As its output, it surfaces the per-instance token and connection URI.

One-Click Launch

1. Open the Launch Stack URL.
2. Click **Create stack** in the bottom right.
3. Wait about two minutes for `CREATE_COMPLETE`.
4. Open the stack's **Outputs** tab.
5. Click the `ConnectURL` value to open a shell.duckdb.org session pre-wired to the new instance, or copy `QuackURI` and `Token` to connect from a local DuckDB session (see below).

The stack defaults to a `t3.micro` instance; other sizes are selectable on the Parameters page when launching. Supported regions: `us-east-1`, `us-east-2`, `us-west-1`, `us-west-2`, `eu-west-1`, `eu-central-1`, `ap-northeast-1`, `ap-southeast-1`.

Connecting from a Local DuckDB

Copy `QuackURI` and `Token` from the stack's Outputs tab and substitute below:

```
-- Register the credentials once per session.
CREATE SECRET quack_credentials (
    TYPE quack,
    SCOPE '<uri>',                  -- e.g., 'quack:54.1.2.3.nip.io:443'
    TOKEN '<token>'
);

-- Who did I just launch?
FROM quack_query('<uri>', 'FROM whoami()');

-- Anything else you'd normally run, shipped verbatim to the remote.
FROM quack_query('<uri>', 'SELECT 1 + 1');
```

Sticky Session via ATTACH `ATTACH` keeps server-side state (temp tables, `SET` variables) across calls, which `quack_query` alone does not.

```
ATTACH '<uri>' AS remote (TYPE quack);

-- Temp table lives on the remote.
FROM quack_query_by_name('remote', 'CREATE TEMP TABLE t AS SELECT range AS x FROM range(10)');
FROM quack_query_by_name('remote', 'SELECT sum(x) FROM t');

-- Session settings stick.
FROM quack_query_by_name('remote', 'SET threads = 8');
FROM quack_query_by_name('remote', 'SELECT current_setting(''threads'')');
```

Tearing Down

When you're done with the instance, delete the stack:

```
aws cloudformation delete-stack --stack-name quack-demo --region us-east-1
```

Or use the **Delete** button in the CloudFormation console.

What the Stack Provisions

- An `AWS::EC2::Instance` from a public AMI (per-region map baked into the template) running DuckDB, the quack extension behind nginx and the Let's Encrypt TLS.
- An `AWS::EC2::SecurityGroup` opening ports `80` (ACME challenge) and `443` (HTTPS).
- An `AWS::CloudFormation::WaitCondition` the instance signals once the RPC server is ready.
- CFN `Outputs` carrying the ready URI, per-instance token, and two shareable shell.duckdb.org URLs (`QueryURL`, `ConnectURL`).

For Maintainers

The CloudFormation template, AMI baking process, and per-region AMI publishing live at `duckdb/duckdb-quack-infra`. Updating the AMI or extending the supported region list happens there.

Quack on WebAssembly

CloudFormation Stack

We provide an example template for initializing a CloudFormation stack, based on pre-baked public AMI, based on Ubuntu, which will:

- via DuckDB, install and load Quack, create a randomized token, and do `quack_serve` on the `0.0.0.0:1294` port.
- setup via nginx a proxy between `0.0.0.0:1294` and the incoming `:443` port
- obtain a valid TLS certificate via Let's Encrypt

All together, this allows to expose the local `0.0.0.0:1294` port to the public EC2 instance IP via HTTPS.

The template is reachable at https://duckdb-quack-ami.s3.us-east-1.amazonaws.com/quack.yaml, and is backed by images for (at the moment) eight regions:

- `us-east-1`
- `us-east-2`
- `us-west-1`
- `us-west-2`
- `eu-central-1`
- `eu-west-1`
- `ap-northeast-1`
- `ap-southeast-1`

Only inputs are the type of the instance (default t3.micro) and the name of the stack (needs to be unique on the same org/region).

> Template is maintained at https://github.com/duckdb/duckdb-quack-infra

Deploy via Web UI

1. Visit https://console.aws.amazon.com/cloudformation/home#/stacks/quickcreate?templateURL=https://duckdb-quack-ami.s3.us-east-1.amazonaws.com/quack.yaml&stackName=my-duckdb-quack-demo
2. Log-in, select a name, possibly an instance class, and click **Launch Stack**.

> This will incur billing. Make sure you check the stack page afterward, and delete the non-relevant anymore stacks.

Deploy via aws CLI

To deploy a Quack demo with the name `my-quack-demo` in the `us-east-2` region, run:

```
aws cloudformation create-stack \
  --stack-name my-quack-demo \
  --template-url https://duckdb-quack-ami.s3.us-east-1.amazonaws.com/quack.yaml \
  --region us-east-2
```

Wait approx. 2 minutes for the deployment to complete (this command is blocking):

```
aws cloudformation wait stack-create-complete \
  --stack-name my-quack-demo \
  --region us-east-2
```

Consult the auto-generated outputs:

```
aws cloudformation describe-stacks \
  --stack-name my-quack-demo \
  --region us-east-2 \
  --query 'Stacks[0].Outputs' \
  --output table
```

Now you can visit the relevant link, that will encode connecting to the just created quack server via DuckDB-Wasm.

Destroy

```
aws cloudformation delete-stack \
  --stack-name my-quack-demo \
  --region us-east-2
```

Web UI

To see the status of your CloudFormation stack on the AWS web user interface, visit the CloudFormation console.

Troubleshooting Quack

DuckDB 2.0 ships a polished version of Quack (first released in DuckDB 1.5.3) with substantial performance and stability improvements. If you experience any issues with Quack, try upgrading all your DuckDB clients to the latest version of the `quack` extension:

```
FORCE INSTALL quack;
```

ATTACH Fails to Bind a Remote Table Definition

When a Quack client runs `ATTACH 'quack:<host>'`, it loads the server's table definitions up front and binds them locally. A single definition that cannot be bound fails the entire attachment, even when you only intend to query unrelated, ordinary tables on the server. The attachment does not silently skip the problematic table.

A common cause is an Iceberg catalog attached on the server with the default `TABLE_RESOLUTION 'lazy'`. Lazily resolved tables are listed with a placeholder column of type `UNKNOWN`, which the client receives through `duckdb_tables()` and cannot bind:

```
Catalog Error: Type with name UNKNOWN does not exist!
```

The `UNKNOWN` type here is not the table's real type, and it does not mean the client must install the `iceberg` extension. It means the server has not yet resolved the table's schema.

Resolving Tables Eagerly on the Server

Attach the Iceberg catalog on the server with `TABLE_RESOLUTION 'eager'` so that schemas are resolved while tables are enumerated, which lets Quack import their real definitions:

{:.codebox-server}

```
-- Server: substitute your own warehouse, URI, and secret.
LOAD iceberg;
LOAD quack;

ATTACH 'warehouse' AS ice (
    TYPE iceberg,
    URI 'https://catalog.example.com/api/catalog',
    SECRET iceberg_catalog_secret,
    TABLE_RESOLUTION 'eager'
);

CREATE TABLE local_data AS SELECT 42 AS i;
CALL quack_serve('quack:localhost', token = 'example-token');
```

The client can then attach and query as usual:

{:.codebox-client}

```
-- Client, in a separate DuckDB instance.
LOAD quack;
ATTACH 'quack:localhost' AS remote (TOKEN 'example-token');
FROM remote.local_data;
```

Eager resolution adds metadata-loading work and can increase catalog enumeration time for large catalogs. It does not fix unrelated binding errors or failures to load Iceberg metadata.

Resolving only some tables before attaching is not enough: if any other unresolved table remains, the attachment still fails. Warming every table individually is a workaround, but restarting the server or adding new unresolved tables brings the problem back, so eager server-side resolution is the more reliable fix.

Querying Without Attaching

If you want to keep lazy resolution, use `quack_query()`, which runs SQL directly on the server without importing its catalog:

{:.codebox-client}

```
FROM quack_query(
    'quack:localhost',
    'SELECT count(*) FROM ice.demo.orders',
    token = 'example-token'
);
```

Guides

Guides

The guides section contains compact how-to guides that are focused on achieving a single goal. For API references and examples, see the rest of the documentation.

Note that there are many tools using DuckDB, which are not covered in the official guides. To find a list of these tools, check out the Awesome DuckDB repository.

> **Tip.** For a short introductory tutorial, check out the "Analyzing Railway Traffic in the Netherlands" tutorial.

Data Import and Export

- Data import overview
- File access with the `file:` protocol
- Reading DuckDB databases

CSV Files

- How to load a CSV file into a table
- How to export a table to a CSV file

Parquet Files

- How to load a Parquet file into a table
- How to export a table to a Parquet file
- How to run a query directly on a Parquet file

HTTP(S), S3 and GCP

- Network and cloud storage overview
- How to authenticate to S3 / AWS
- How to load a Parquet file directly from HTTP(S)
- How to load a CSV file directly from HTTP(S)
- How to load a Parquet file directly from S3
- How to export a Parquet file to S3
- How to load a Parquet file from S3 Express One
- How to load a Parquet file directly from GCS
- How to load a Parquet file directly from Cloudflare R2
- How to load a Parquet file directly from Fastly Object Storage
- How to load a Parquet file directly from SeaweedFS
- How to load a Parquet file directly from Tigris
- How to load an Iceberg table directly from S3
- How to connect to Amazon S3 Tables
- How to connect to Amazon SageMaker Lakehouse (AWS Glue)
- How to share a DuckDB database that only stores views
- How to attach to a DuckDB database over HTTPS or S3

JSON Files

- How to load a JSON file into a table
- How to export a table to a JSON file

Excel Files

- How to load an Excel file into a table
- How to export a table to an Excel file

Querying Other Database Systems

- Database integration overview
- How to directly query a MySQL database
- How to directly query a PostgreSQL database
- How to directly query a SQLite database
- How to connect to Amazon RDS / Aurora with IAM authentication

Directly Reading Files

- How to directly read a binary file
- How to directly read a text file

Performance

- Performance guide overview
- My workload is slow (troubleshooting guide)
- How to design the schema for optimal performance
- What is the ideal hardware environment for DuckDB
- How to import data efficiently
- How to use indexes and zonemaps
- How to make join operations efficient
- What performance implications do Parquet files and (compressed) CSV files have
- How to tune workloads
- How to avoid out-of-memory issues
- How to work with huge databases
- Benchmarks

Meta Queries

- How to list all tables
- How to view the schema of the result of a query
- How to quickly get a feel for a dataset using summarize
- How to view the query plan of a query
- How to profile a query
- How to inspect the DuckDB environment (version, platform, extensions)

ODBC

- How to set up an ODBC application (and more!)

Python Client

- How to install the Python client
- How to execute SQL queries
- How to easily query DuckDB in Jupyter Notebooks
- How to easily query DuckDB in marimo Notebooks
- How to use Multiple Python Threads with DuckDB
- How to use fsspec filesystems with DuckDB

Pandas

- How to execute SQL on a Pandas DataFrame
- How to create a table from a Pandas DataFrame
- How to export data to a Pandas DataFrame

NumPy

- How to create a table from NumPy arrays
- How to export data to NumPy arrays

Apache Arrow

- How to execute SQL on Apache Arrow
- How to create a DuckDB table from Apache Arrow
- How to export data to Apache Arrow

Relational API

- How to query Pandas DataFrames with the Relational API

Python Library Integrations

- How to use Ibis to query DuckDB with or without SQL
- How to use DuckDB with Polars DataFrames via Apache Arrow
- How to stream DuckDB query results to PyTorch via Apache Arrow

SQL Features

- Friendly SQL
- As-of join
- Full-text search
- Graph queries
- `query` and `query_table` functions
- Merge statement for SCD Type 2
- Timestamp issues

SQL Editors and IDEs

- How to set up the Beekeeper Studio SQL editor
- How to set up the DBeaver SQL IDE
- How to set up the DBine SQL editor
- How to set up the LibreDB Studio SQL editor
- How to set up the VisuaLeaf SQL editor

Data Viewers

- How to visualize DuckDB databases with Tableau
- How to draw command-line plots with DuckDB and YouPlot

Snippets

- Creating synthetic data
- Dutch railway datasets
- Sharing macros
- Analyzing a Git repository
- Importing duckbox tables
- Copying an in-memory database to a file
- Calculating a database checksum

Troubleshooting

- Command line
- Crashes
- Out of memory errors

Glossary and Offline Docs

- Glossary of terms
- Browsing offline

Data Viewers

Tableau – A Data Visualization Tool

Tableau is a popular commercial data visualization tool. In addition to a large number of built-in connectors, it also provides generic database connectivity via ODBC and JDBC connectors.

Tableau has two main versions: Desktop and Online (Server).

- For Desktop, connecting to a DuckDB database is similar to working in an embedded environment like Python.
- For Online, since DuckDB is in-process, the data needs to be either on the server itself or in a remote data bucket that is accessible from the server.

Database Creation

When using a DuckDB database file the datasets do not actually need to be imported into DuckDB tables; it suffices to create views of the data. For example, this will create a view of the `h2oai` Parquet test file in the current DuckDB code base:

```
CREATE VIEW h2oai AS (
    FROM read_parquet('/Users/username/duckdb/data/parquet-testing/h2oai/h2oai_group_small.parquet')
);
```

Note that you should use full path names to local files so that they can be found from inside Tableau. Also note that you will need to use a version of the driver that is compatible (i.e., from the same release) as the database format used by the DuckDB tool (e.g., Python module, command line) that was used to create the file.

Installing the JDBC Driver

Tableau provides documentation on how to install a JDBC driver for Tableau to use.

> Both Tableau Desktop and Server need to be restarted any time you add or modify drivers.

Driver Links

The link here is for a recent version of the JDBC driver that is compatible with Tableau. If you wish to connect to a database file, you will need to make sure the file was created with a file-compatible version of DuckDB. Also, check that there is only one version of the driver installed as there are multiple filenames in use.

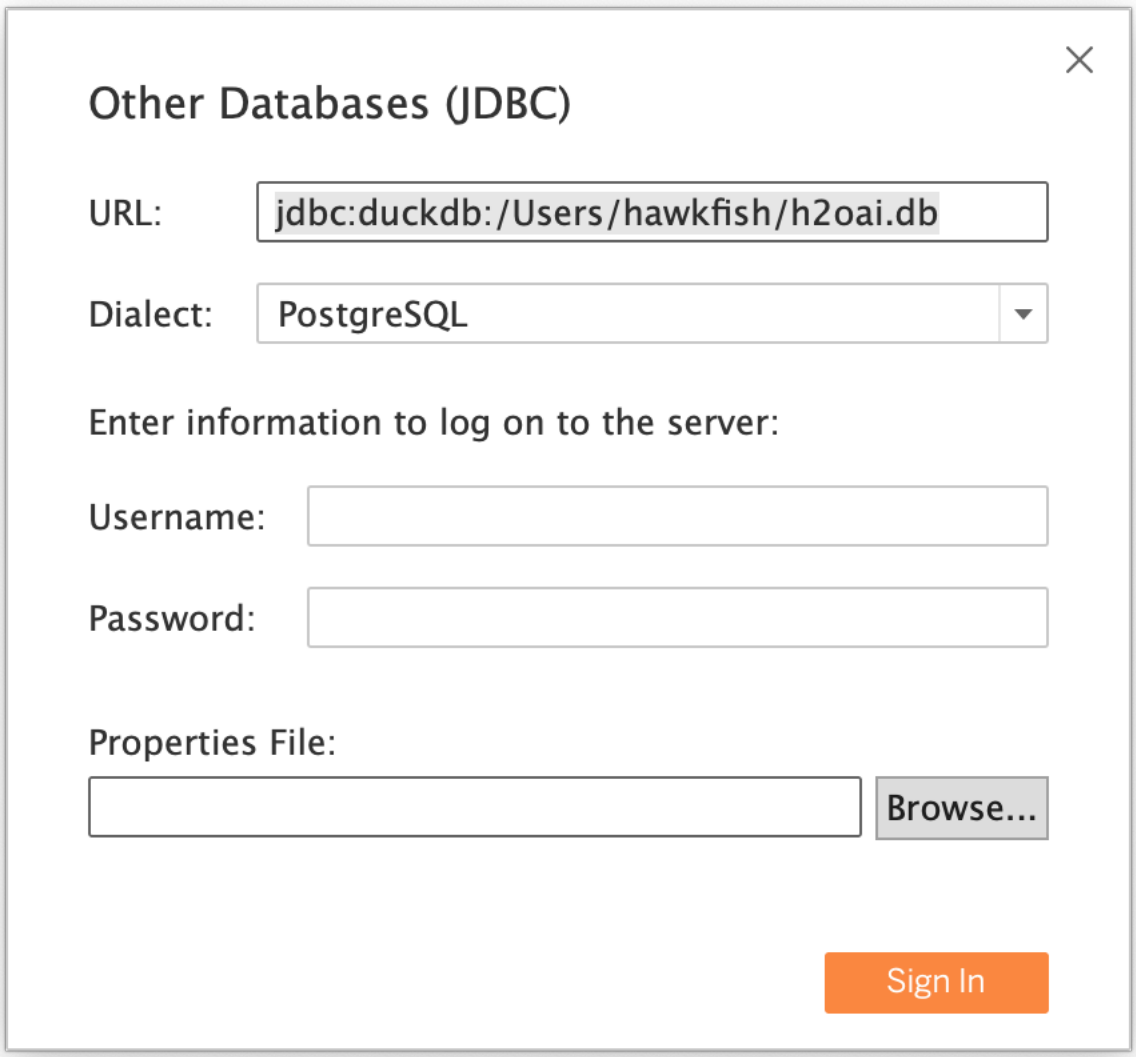
Download the JAR file.

- macOS: Copy it to `~/Library/Tableau/Drivers/`.
- Windows: Copy it to `C:\Program Files\Tableau\Drivers`.
- Linux: Copy it to `/opt/tableau/tableau_driver/jdbc`.

Using the PostgreSQL Dialect

If you just want to do something simple, you can try connecting directly to the JDBC driver and using the Tableau-provided PostgreSQL dialect.

1. Create a DuckDB file containing your views and/or data.
2. Launch Tableau.
3. Under Connect > To a Server > More…, click on "Other Databases (JDBC)". This brings up the connection dialog box. For the URL, enter `jdbc:duckdb:/User/username/path/to/database.db`. For the Dialect, choose PostgreSQL. The rest of the fields can be ignored:



However, functionality will be missing such as `median` and `percentile` aggregate functions. To make the data source connection more compatible with the PostgreSQL dialect, please use the DuckDB taco connector as described below.

Installing the Tableau DuckDB Connector

While it is possible to use the Tableau-provided PostgreSQL dialect to communicate with the DuckDB JDBC driver, we strongly recommend using the DuckDB Taco connector. This connector has been fully tested against the Tableau dialect generator and is more compatible than the provided PostgreSQL dialect.

The documentation on how to install and use the connector is in its repository, but essentially you will need the `duckdb_jdbc.taco` file. (Despite what the Tableau documentation says, the real security risk is in the JDBC driver code, not the small amount of JavaScript in the Taco file.)

Server (Online)

On Linux, copy the Taco file to `/opt/tableau/connectors`. On Windows, copy the Taco file to `C:\Program Files\Tableau\Connectors`. Then issue these commands to disable signature validation:

```
tsm configuration set -k native_api.disable_verify_connector_plugin_signature -v true
```

```
tsm pending-changes apply
```

The last command will restart the server with the new settings.

macOS

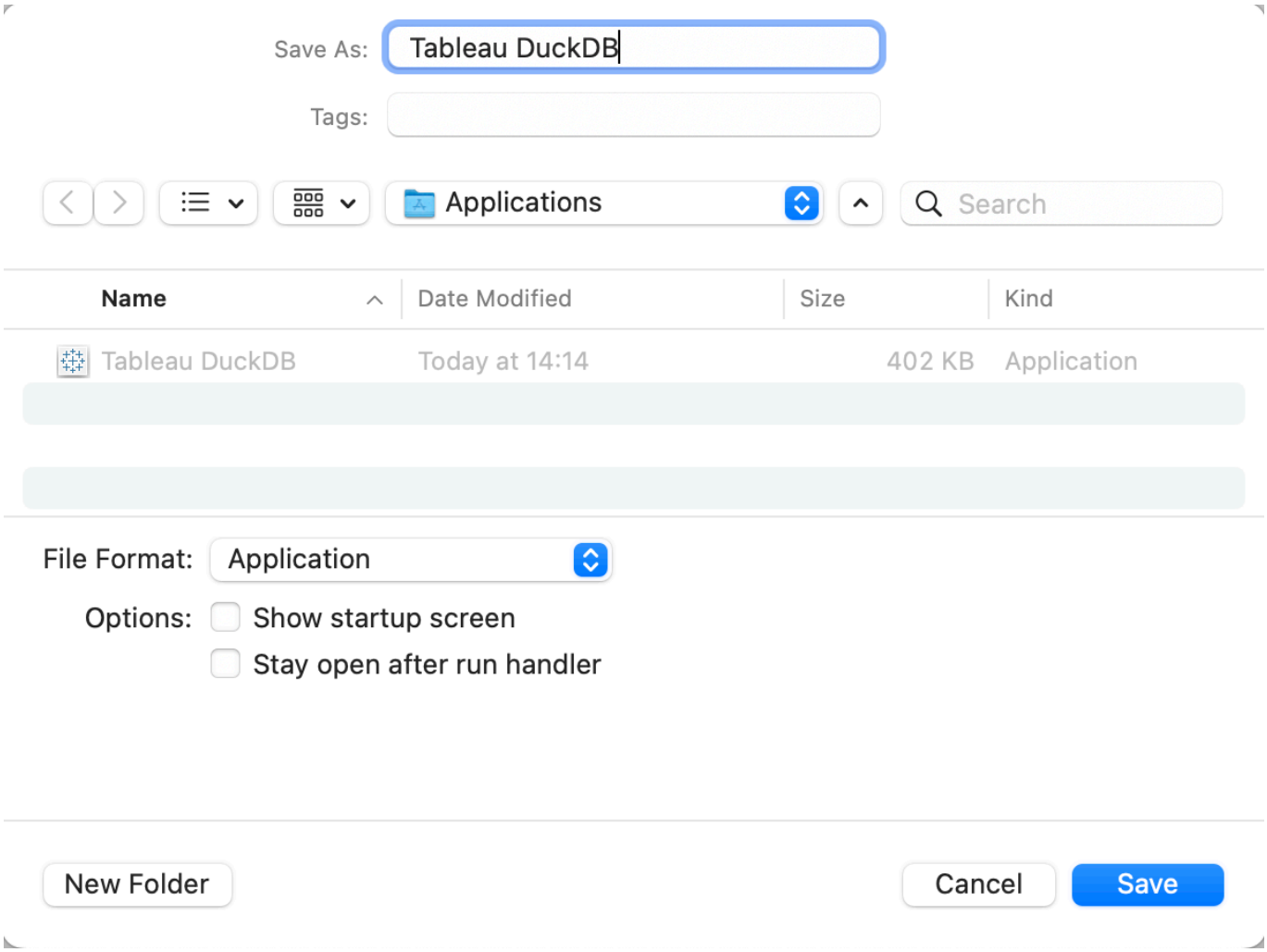
Copy the Taco file to the `/Users/[User]/Documents/My Tableau Repository/Connectors` folder. Then launch Tableau Desktop from the Terminal with the command line argument to disable signature validation:

```
/Applications/Tableau\ Desktop\ <year>.<quarter>.app/Contents/MacOS/Tableau
-DDisableVerifyConnectorPluginSignature=true
```

You can also package this up with AppleScript by using the following script:

```
do shell script "\"/Applications/Tableau Desktop 2023.2.app/Contents/MacOS/Tableau\"
-DDisableVerifyConnectorPluginSignature=true"
quit
```

Create this file with the Script Editor (located in `/Applications/Utilities`) and save it as a packaged application:



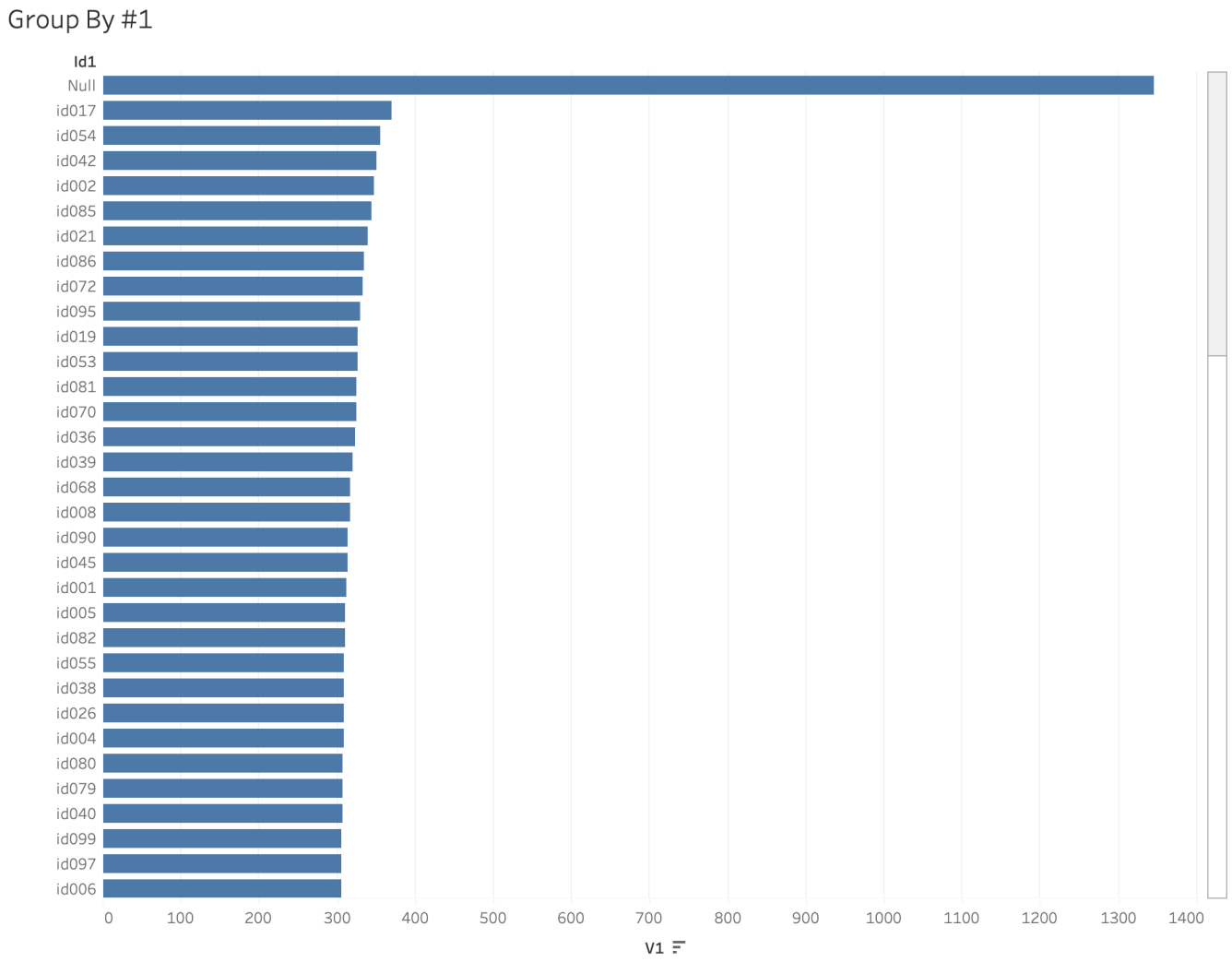
You can then double-click it to launch Tableau. You will need to change the application name in the script when you get upgrades.

Windows Desktop

Copy the Taco file to the `C:\Users\[Windows User]\Documents\My Tableau Repository\Connectors` directory. Then launch Tableau Desktop from a shell with the `-DDisableVerifyConnectorPluginSignature=true` argument to disable signature validation.

Output

Once loaded, you can run queries against your data! Here is the result of the first H2O.ai benchmark query from the Parquet test file:



CLI Charting with YouPlot

DuckDB can be used with CLI graphing tools to quickly pipe input to stdout to graph your data in one line.

YouPlot is a Ruby-based CLI tool for drawing visually pleasing plots on the terminal. It can accept input from other programs by piping data from `stdin`. It takes tab-separated (or delimiter of your choice) data and can easily generate various types of plots including bar, line, histogram and scatter.

With DuckDB, you can write to the console (`stdout`) by using the `TO '/dev/stdout'` command. And you can also write comma-separated values by using `WITH (FORMAT csv, HEADER)`.

Installing YouPlot

Installation instructions for YouPlot can be found on the main YouPlot repository. If you're on a Mac, you can use:

```
brew install youplot
```

Run `uplot --help` to ensure you've installed it successfully!

Piping DuckDB Queries to stdout

By combining the `COPY...TO` function with a CSV output file, you can read data from any format DuckDB supports and pipe it to YouPlot. Follow these three steps:

1. First, read all data from `input.json`:

   ```
   duckdb -s "SELECT * FROM read_json('input.json')"
   ```

2. To prepare the data for YouPlot, write a simple aggregate:

   ```
   duckdb -s "SELECT date, sum(purchases) AS total_purchases FROM read_json('input.json') GROUP BY 1
   ORDER BY 2 DESC LIMIT 10"
   ```

3. Finally, wrap the `SELECT` in the `COPY ... TO` function with an output location of `/dev/stdout`.

   The syntax looks like this:

   ```
   COPY (<query>) TO '/dev/stdout' WITH (FORMAT csv, HEADER);
   ```

   The full DuckDB command below outputs the query in CSV format with a header:

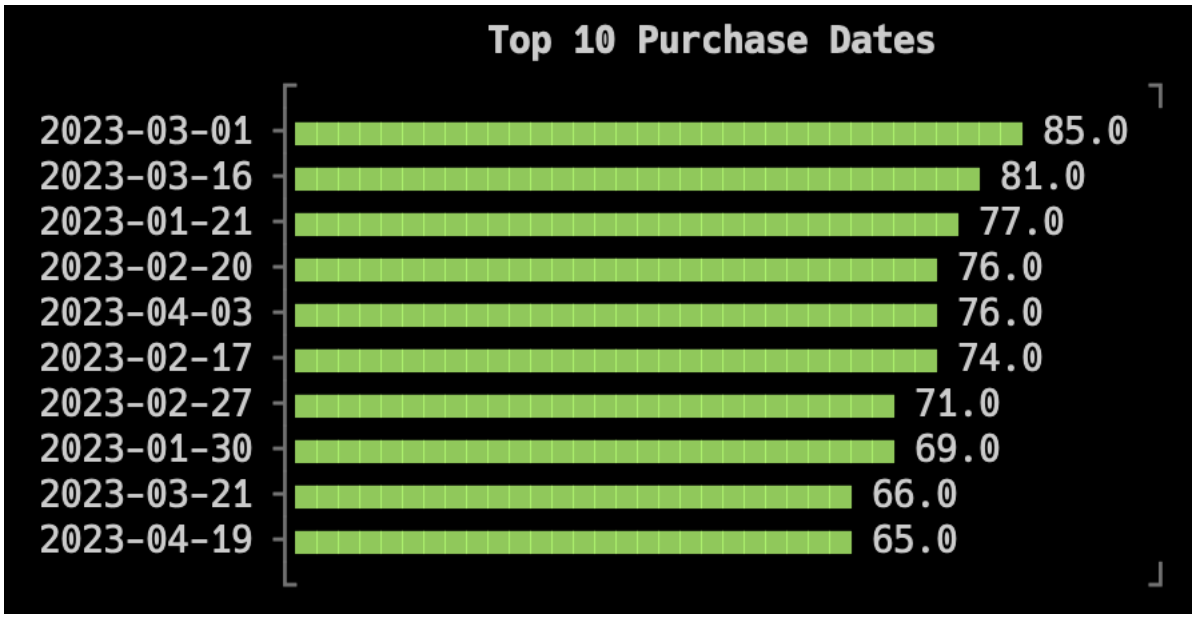
   ```
   duckdb -s "COPY (SELECT date, sum(purchases) AS total_purchases FROM read_json('input.json') GROUP BY
   1 ORDER BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT csv, HEADER)"
   ```

Connecting DuckDB to YouPlot

Finally, the data can now be piped to YouPlot! Let's assume we have an `input.json` file with dates and number of purchases made by somebody on that date. Using the query above, we'll pipe the data to the `uplot` command to draw a plot of the Top 10 Purchase Dates.

```
duckdb -s "COPY (SELECT date, sum(purchases) AS total_purchases FROM read_json('input.json') GROUP BY 1
ORDER BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT csv, HEADER)" \
     | uplot bar -d, -H -t "Top 10 Purchase Dates"
```

This tells `uplot` to draw a bar plot, use a comma-separated delimiter (`-d,`), that the data has a header (`-H`), and give the plot a title (`-t`).

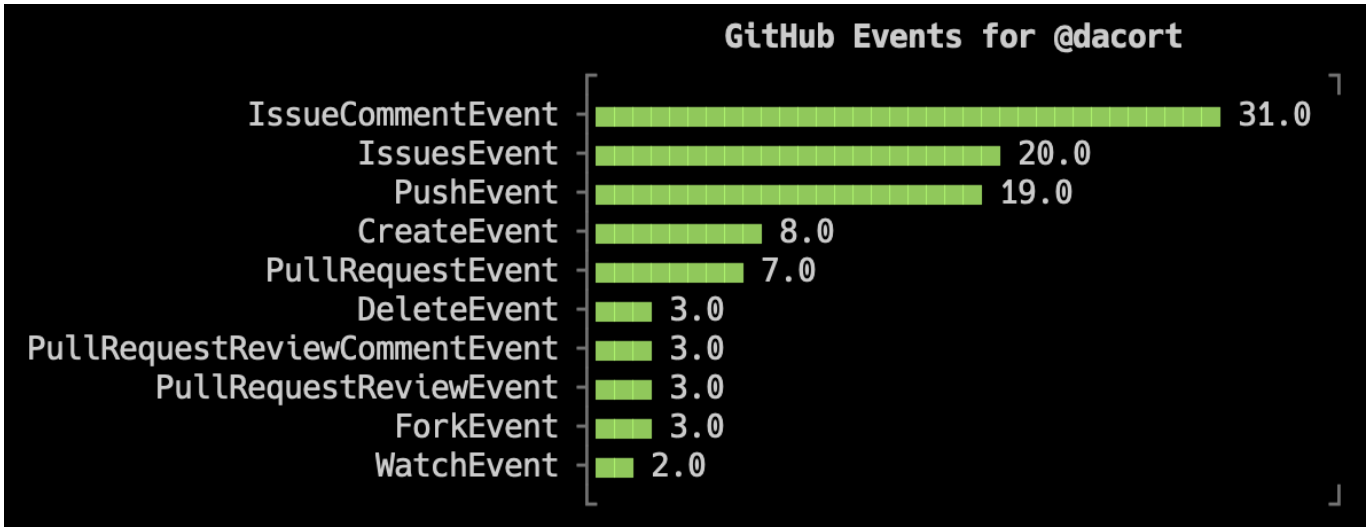


Additional Example: stdin + stdout

You might be piping data through `jq` or downloading a JSON file from somewhere. You can also tell DuckDB to read data from another process by changing the filename to `/dev/stdin`.

Let's combine this with a quick `curl` from GitHub to see what a certain user has been up to lately.

```
curl -sL "https://api.github.com/users/dacort/events?per_page=100" \
     | duckdb -s "COPY (SELECT type, count(*) AS event_count FROM read_json('/dev/stdin') GROUP BY 1 ORDER
BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT csv, HEADER)" \
     | uplot bar -d, -H -t "GitHub Events for @dacort"
```



Database Integration

Database Integration

MySQL Import

To run a query directly on a running MySQL database, the `mysql` extension is required.

Installation and Loading

The extension can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL mysql;
```

To load the `mysql` extension for usage, use the `LOAD` SQL command:

```
LOAD mysql;
```

Usage

After the `mysql` extension is installed, you can attach to a MySQL database using the following command:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql);
USE mysql_db;
```

The string used by `ATTACH` is a PostgreSQL-style connection string (*not* a MySQL connection string!). It is a list of connection arguments provided in `{key}={value}` format. Below is a list of valid arguments. Any options not provided are replaced by their default values.

Setting	Default
`database`	`NULL`
`host`	`localhost`
`password`	
`port`	`0`
`socket`	`NULL`
`user`	current user

You can directly read and write the MySQL database:

```
CREATE TABLE tbl (id INTEGER, name VARCHAR);
INSERT INTO tbl VALUES (42, 'DuckDB');
```

For a list of supported operations, see the MySQL extension documentation.

To run entire queries on the MySQL server instead of scanning its tables from DuckDB, see the `CONNECT` statement.

PostgreSQL Import

To run a query directly on a running PostgreSQL database, the `postgres` extension is required.

Installation and Loading

The extension can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL postgres;
```

To load the `postgres` extension for usage, use the `LOAD` SQL command:

```
LOAD postgres;
```

Usage

After the `postgres` extension is installed, you can attach a PostgreSQL database using the ATTACH command. This allows you to query all tables stored within the PostgreSQL database as if it was a regular database.

The first parameter of `ATTACH` is the PostgreSQL connection string, a list of connection arguments provided in `{key}={value}` format. Below is a list of valid arguments.

Name	Description	Default
`host`	Name of host to connect to	`localhost`
`hostaddr`	Host IP address	`localhost`
`port`	Port number	`5432`
`user`	PostgreSQL user name	[OS user name]
`password`	PostgreSQL password	
`dbname`	Database name	[user]
`passfile`	Name of file passwords are stored in	`~/.pgpass`

```
-- Attach the PostgreSQL database using the given connection string
ATTACH 'host=localhost port=5432 dbname=mydb' AS test (TYPE postgres);
-- The table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM test.tbl_name;
-- Switch the active database to "test"
USE test;
-- List all tables in the file
SHOW TABLES;
```

For more information see the PostgreSQL extension documentation.

To run entire queries on the PostgreSQL server instead of scanning its tables from DuckDB, see the CONNECT statement.

SQLite Import

To run a query directly on a SQLite file, the `sqlite` extension is required.

Installation and Loading

The extension can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL sqlite;
```

To load the `sqlite` extension for usage, use the `LOAD` SQL command:

```
LOAD sqlite;
```

Usage

After the SQLite extension is installed, tables can be queried from SQLite using the `sqlite_scan` function:

```
-- Scan the table "tbl_name" from the SQLite file "test.db"
SELECT * FROM sqlite_scan('test.db', 'tbl_name');
```

Alternatively, the entire file can be attached using the ATTACH command. This allows you to query all tables stored within a SQLite database file as if they were a regular database.

```
-- Attach the SQLite file "test.db"
ATTACH 'test.db' AS test (TYPE sqlite);
-- The table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM test.tbl_name;
-- Switch the active database to "test"
USE test;
-- List all tables in the file
SHOW TABLES;
```

For more information see the SQLite extension documentation.

Amazon RDS with IAM Authentication

Managed PostgreSQL databases on Amazon RDS and Aurora can authenticate with short-lived IAM tokens instead of a static password. DuckDB supports this through the `postgres` and `aws` extensions: the `aws` extension generates the token from an AWS credential chain, and the `postgres` extension uses it as the connection password, refreshing it automatically.

Installation and Loading

```
INSTALL postgres;
INSTALL aws;
LOAD postgres;
LOAD aws;
```

Creating the Secrets

First, create a secret of type `rds`. It uses the AWS credential chain to mint the IAM token:

```
CREATE SECRET aws_rds_secret (
    TYPE rds,
    PROVIDER credential_chain,
    CHAIN 'env;sso',
    REGION '<eu-west-1>',
    RDS_USER '<postgres>',
    RDS_HOST '<instance>.<identifier>.<region>.rds.amazonaws.com',
    RDS_PORT '5432'
);
```

Then create a secret of type `postgres` that references it via `AWS_RDS_SECRET`. Because no password is set, the token generated by the `rds` secret is used (and refreshed) automatically:

```
CREATE SECRET pg_rds_secret (
    TYPE postgres,
    HOST '<instance>.<identifier>.<region>.rds.amazonaws.com',
    PORT '5432',
    USER '<postgres>',
    DATABASE '<postgres>',
    SSLMODE require,
    AWS_RDS_SECRET aws_rds_secret
);
```

Connecting and Querying

Attach the database using the `postgres` secret, then query it as a regular database:

```
ATTACH '' AS rds_db (TYPE postgres, SECRET pg_rds_secret);
SELECT * FROM rds_db.public.<tbl_name>;
```

See Also

- Postgres extension secrets — AWS RDS IAM Authentication — the full reference, including verifying the setup with `psql` first.
- AWS extension — the credential chain options available to the `rds` secret.

File Formats

File Formats

CSV Import

To read data from a CSV file, use the `read_csv` function in the `FROM` clause of a query:

```sql
SELECT * FROM read_csv('input.csv');
```

Alternatively, you can omit the `read_csv` function and let DuckDB infer it from the extension:

```sql
SELECT * FROM 'input.csv';
```

To create a new table using the result from a query, use `CREATE TABLE ... AS SELECT` statement:

```sql
CREATE TABLE new_tbl AS
    SELECT * FROM read_csv('input.csv');
```

We can use DuckDB's optional FROM-first syntax to omit `SELECT *`:

```sql
CREATE TABLE new_tbl AS
    FROM read_csv('input.csv');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```sql
INSERT INTO tbl
    SELECT * FROM read_csv('input.csv');
```

Alternatively, the `COPY` statement can also be used to load data from a CSV file into an existing table:

```sql
COPY tbl FROM 'input.csv';
```

For additional options, see the CSV import reference and the `COPY` statement documentation.

CSV Export

To export the data from a table to a CSV file, use the `COPY` statement:

```sql
COPY tbl TO 'output.csv' (HEADER, DELIMITER ',');
```

The result of queries can also be directly exported to a CSV file:

```sql
COPY (SELECT * FROM tbl) TO 'output.csv' (HEADER, DELIMITER ',');
```

For additional options, see the `COPY` statement documentation.

Directly Reading Files

DuckDB allows directly reading files via the `read_text` and `read_blob` functions. These functions accept a filename, a list of filenames, or a glob pattern. They output the content of each file as a `VARCHAR` or `BLOB`, respectively, along with metadata such as the file size and last modified time.

`read_text`

The `read_text` table function reads from the selected source(s) to a VARCHAR. Each file results in a single row with the `content` field holding the entire content of the respective file.

```
SELECT size, parse_path(filename), content
FROM read_text('test/sql/table_function/files/*.txt');
```

size	parse_path(filename)	content
12	[test, sql, table_function, files, one.txt]	Hello World!
2	[test, sql, table_function, files, three.txt]	42
10	[test, sql, table_function, files, two.txt]	Foo Bar\nFöö Bär

DuckDB first validates the file content as valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8, DuckDB throws an error suggesting to use `read_blob` instead.

`read_text` also supports reading from pipes (e.g., `/dev/stdin`).

> The maximum allowed file size for `read_text` is 3.9 GiB.

`read_blob`

The `read_blob` table function reads from the selected source(s) to a BLOB:

```
SELECT size, content, filename
FROM read_blob('test/sql/table_function/files/*');
```

size	content	filename
178	PK\x03\x04\x0A\x00\x00\x00\x00\x00\xACi=X\x14t\xCE\xC7\x0A…	test/sql/table_function/files/four.blob
12	Hello World!	test/sql/table_function/files/one.txt
2	42	test/sql/table_function/files/three.txt
10	F\xC3\xB6\xC3\xB6 B\xC3\xA4r	test/sql/table_function/files/two.txt

> The maximum allowed file size for `read_blob` is 3.9 GiB.

Schema

The schemas of the tables returned by `read_text` and `read_blob` are identical:

```
DESCRIBE FROM read_text('README.md');
```

column_name	column_type	null	key	default	extra
filename	VARCHAR	YES	NULL	NULL	NULL
content	VARCHAR	YES	NULL	NULL	NULL
size	BIGINT	YES	NULL	NULL	NULL
last_modified	TIMESTAMP WITH TIME ZONE	YES	NULL	NULL	NULL

Hive Partitioning

Data can be read from Hive partitioned datasets.

```
SELECT *
FROM read_blob('data/parquet-testing/hive-partitioning/simple/**/*.parquet')
WHERE part IN ('a', 'b') AND date >= '2012-01-01';
```

filename	content	size	last_modified	date	part
…/part=a/date=2012-01-01/test.parquet	PAR1\x15\x00\x15\x14\x15\x18…	266	2024-11-12 02:23:20+00	2012-01-01	a
…/part=b/date=2013-01-01/test.parquet	PAR1\x15\x00\x15\x14\x15\x18…	266	2024-11-12 02:23:20+00	2013-01-01	b

Handling Missing Metadata

When the underlying filesystem cannot provide this data (e.g., HTTPFS may not always return a valid timestamp), the cell is set to `NULL` instead.

Support for Projection Pushdown

These table functions also use projection pushdown to avoid computing properties unnecessarily. For example, you can glob a directory of large files to get file sizes in the `size` column. As long as you omit the `content` column, DuckDB won't read the file data.

Directly Read DuckDB Databases

DuckDB allows directly reading DuckDB files through the `read_duckdb` function:

```
read_duckdb(<'path_to_database'>, table_name = <'table_to_read'>);
```

Using this function is equivalent to performing the following steps:

- Attaching to the database using a read-only connection.
- Querying the table specified through the `table_name` argument.
- Closing the connection to the database.

Examples

Reading a Specific Table

To read the `region` table from the TPC-H dataset, run:

```
SELECT r_regionkey, r_name
FROM read_duckdb('https://blobs.duckdb.org/data/tpch-sf10.db', table_name = 'region');
```

r_regionkey int32	r_name varchar
0	AFRICA
1	AMERICA
2	ASIA
3	EUROPE
4	MIDDLE EAST

Reading from Multiple Databases

You can use globbing to read from multiple databases. To illustrate this, let's create two databases:

```
duckdb my-1.duckdb \
    -c "CREATE TABLE numbers AS SELECT 42 AS x;" \
    -c "CREATE TABLE letters AS SELECT 'm' AS a;"

duckdb my-2.duckdb \
    -c "CREATE TABLE numbers AS SELECT 43 AS x;"
```

Then, in DuckDB, you can run:

```
SELECT x FROM read_duckdb('my-*.duckdb', table_name = 'numbers');
```

```
┌─────────┐
│    x    │
│  int32  │
├─────────┤
│      42 │
│      43 │
└─────────┘
```

Reading from Databases with a Single Table

If all databases in `read_duckdb`'s argument have a single table, the `table_name` argument is optional:

```
FROM read_duckdb('my-2.duckdb');
```

```
┌─────────┐
│    x    │
│  int32  │
├─────────┤
│      43 │
└─────────┘
```

If the extension is `.db` or `.duckdb`, you can also omit the `read_duckdb` call (similarly to how you can omit `read_csv` and `read_parquet`):

```
FROM 'my-2.duckdb';
```

Limitations

`read_duckdb` currently only supports reading from tables. Reading from views is not yet supported.

Excel Import

DuckDB supports reading Excel `.xlsx` files. However, `.xls` files are not supported.

Importing Excel Sheets

Use the `read_xlsx` function in the FROM clause of a query:

```
SELECT * FROM read_xlsx('test_excel.xlsx');
```

Alternatively, you can omit the `read_xlsx` function and let DuckDB infer it from the extension:

```
SELECT * FROM 'test_excel.xlsx';
```

However, if you want to be able to pass options to control the import behavior, you should use the `read_xlsx` function.

One such option is the `sheet` parameter, which allows specifying the name of the Excel worksheet:

```
SELECT * FROM read_xlsx('test_excel.xlsx', sheet = 'Sheet1');
```

By default, the first sheet is loaded if no sheet is specified.

Importing a Specific Range

To select a specific range of cells, use the `range` parameter with a string in the format `A1:B2`, where `A1` is the top-left cell and `B2` is the bottom-right cell:

```
SELECT * FROM read_xlsx('test_excel.xlsx', range = 'A1:B2');
```

For example, to skip the first 5 rows:

```
SELECT * FROM read_xlsx('test_excel.xlsx', range = 'A5:Z');
```

To skip the first 5 columns:

```
SELECT * FROM read_xlsx('test_excel.xlsx', range = 'E:Z');
```

If no range parameter is provided, DuckDB automatically infers the range as the rectangular region of cells between the first row of consecutive non-empty cells and the first empty row spanning the same columns.

By default, if no range is provided, DuckDB will stop reading the Excel file when encountering an empty row. But when a range is provided, the default is to read until the end of the range. This behavior can be controlled with the `stop_at_empty` parameter:

```
-- Read the first 100 rows, or until the first empty row, whichever comes first
SELECT * FROM read_xlsx('test_excel.xlsx', range = '1:100', stop_at_empty = true);

-- Always read the whole sheet, even if it contains empty rows
SELECT * FROM read_xlsx('test_excel.xlsx', stop_at_empty = false);
```

Creating a New Table

To create a new table using the result from a query, use `CREATE TABLE ... AS` from a `SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT * FROM read_xlsx('test_excel.xlsx', sheet = 'Sheet1');
```

Loading to an Existing Table

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT * FROM read_xlsx('test_excel.xlsx', sheet = 'Sheet1');
```

Alternatively, you can use the COPY statement with the `XLSX` format option to import an Excel file into an existing table:

```
COPY tbl FROM 'test_excel.xlsx' (FORMAT xlsx, SHEET 'Sheet1');
```

When using the COPY statement to load an Excel file into an existing table, the types of the columns in the target table will be used to coerce the types of the cells in the Excel sheet.

Importing a Sheet with/without a Header

To treat the first row as containing the names of the resulting columns, use the `header` parameter:

```
SELECT * FROM read_xlsx('test_excel.xlsx', header = true);
```

By default, the first row is treated as a header if all the cells in the first row (within the inferred or supplied range) are non-empty strings. To disable this behavior, set `header` to `false`.

Detecting Types

When not importing into an existing table, DuckDB will attempt to infer the types of the columns in the Excel sheet based on their contents and/or "number format".

- `TIMESTAMP`, `TIME`, `DATE` and `BOOLEAN` types are inferred when possible based on the "number format" applied to the cell.
- Text cells containing `TRUE` and `FALSE` are inferred as `BOOLEAN`.
- Empty cells are considered to be of type `DOUBLE` by default.
- Otherwise cells are inferred as `VARCHAR` or `DOUBLE` based on their contents.

You can adjust this behavior in several ways.

To treat all empty cells as `VARCHAR` instead of `DOUBLE`, set `empty_as_varchar` to `true`:

```
SELECT * FROM read_xlsx('test_excel.xlsx', empty_as_varchar = true);
```

To disable type inference completely and treat all cells as `VARCHAR`, set `all_varchar` to `true`:

```
SELECT * FROM read_xlsx('test_excel.xlsx', all_varchar = true);
```

Additionally, if the `ignore_errors` parameter is set to `true`, DuckDB will silently replace cells that can't be cast to the corresponding inferred column type with `NULL`s.

```
SELECT * FROM read_xlsx('test_excel.xlsx', ignore_errors = true);
```

See Also

DuckDB can also export Excel files. For additional details on Excel support, see the excel extension page.

Excel Export

DuckDB supports exporting data to Excel `.xlsx` files via the `excel` extension. Please note that `.xls` files are not supported.

To install and load the extension, run:

```
INSTALL excel;
LOAD excel;
```

Exporting Excel Sheets

To export a table to an Excel file, use the COPY statement with the `FORMAT xlsx` option:

```
COPY tbl TO 'output.xlsx' WITH (FORMAT xlsx);
```

The result of a query can also be directly exported to an Excel file:

```
COPY (SELECT * FROM tbl) TO 'output.xlsx' WITH (FORMAT xlsx);
```

Or:

```
COPY (SELECT * FROM tbl) TO 'output.xlsx';
```

To write the column names as the first row in the Excel file, use the `HEADER` option:

```
COPY tbl TO 'output.xlsx' WITH (FORMAT xlsx, HEADER true);
```

To name the worksheet in the resulting Excel file, use the `SHEET` option:

```
COPY tbl TO 'output.xlsx' WITH (FORMAT xlsx, SHEET 'Sheet1');
```

Type Conversions

Because Excel only really supports storing numbers or strings – the equivalent of `VARCHAR` and `DOUBLE`, the following type conversions are automatically applied when writing XLSX files:

- Numeric types are cast to `DOUBLE`.
- Temporal types (`TIMESTAMP`, `DATE`, `TIME`, etc.) are converted to Excel "serial" numbers, that is the number of days since 1900-01-01 for dates and the fraction of a day for times. These are then styled with a "number format" so that they appear as dates or times when opened in Excel.
- `TIMESTAMP_TZ` and `TIME_TZ` are cast to UTC `TIMESTAMP` and `TIME` respectively, with the timezone information being lost.
- `BOOLEAN`s are converted to `1` and `0`, with a "number format" applied to make them appear as `TRUE` and `FALSE` in Excel.
- All other types are cast to `VARCHAR` and then written as text cells.

But you can of course also explicitly cast columns to a different type before exporting them to Excel:

```
COPY (SELECT CAST(a AS VARCHAR), b FROM tbl) TO 'output.xlsx' WITH (FORMAT xlsx);
```

See Also

DuckDB can also import Excel files. For additional details on Excel support, see the `excel` extension page.

JSON Import

To read data from a JSON file, use the `read_json` function in the `FROM` clause of a query:

```
SELECT *
FROM read_json('input.json');
```

To create a new table using the result from a query, use `CREATE TABLE AS` from a `SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT *
    FROM read_json('input.json');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT *
    FROM read_json('input.json');
```

Alternatively, the `COPY` statement can also be used to load data from a JSON file into an existing table:

```
COPY tbl FROM 'input.json';
```

For additional options, see the JSON Loading reference and the `COPY` statement documentation.

JSON Export

To export the data from a table to a JSON file, use the `COPY` statement:

```
COPY tbl TO 'output.json';
```

The result of queries can also be directly exported to a JSON file:

```
COPY (SELECT * FROM range(3) tbl(n)) TO 'output.json';
```

```
{"n":0}
{"n":1}
{"n":2}
```

The JSON export writes JSON lines by default, standardized as Newline-delimited JSON. The ARRAY option can be used to write a single JSON array object instead.

```
COPY (SELECT * FROM range(3) tbl(n)) TO 'output.json' (ARRAY);
```

```
[
        {"n":0},
        {"n":1},
        {"n":2}
]
```

For additional options, see the COPY statement documentation.

Parquet Import

To read data from a Parquet file, use the `read_parquet` function in the `FROM` clause of a query:

```
SELECT * FROM read_parquet('input.parquet');
```

Alternatively, you can omit the `read_parquet` function and let DuckDB infer it from the extension:

```
SELECT * FROM 'input.parquet';
```

To create a new table using the result from a query, use the `CREATE TABLE ... AS SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT * FROM read_parquet('input.parquet');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT * FROM read_parquet('input.parquet');
```

Alternatively, use the COPY statement to load data from a Parquet file into an existing table:

```
COPY tbl FROM 'input.parquet' (FORMAT parquet);
```

Adjusting the Schema on the Fly

You can load a Parquet file into a slightly different schema (e.g., different number of columns, more relaxed types) using the following trick.

Suppose you have a Parquet file with two columns, `c1` and `c2`:

```
COPY (FROM (VALUES (42, 43)) t(c1, c2))
TO 'f.parquet';
```

To add another column `c3` that is not present in the file, run:

```
FROM (VALUES (NULL::VARCHAR, NULL, NULL::INTEGER)) t(c1, c2, c3)
WHERE false
UNION ALL BY NAME
FROM 'f.parquet';
```

The first `FROM` clause generates an empty table with *three* columns where `c1` is a `VARCHAR` and `c3` is an `INTEGER`. Then, use `UNION ALL BY NAME` to union the Parquet file. The result is:

c1 varchar	c2 int32	c3 int32
42	43	NULL

For additional options, see the Parquet loading reference.

Parquet Export

To export the data from a table to a Parquet file, use the COPY statement:

```
COPY tbl TO 'output.parquet' (FORMAT parquet);
```

The result of queries can also be directly exported to a Parquet file:

```
COPY (SELECT * FROM tbl) TO 'output.parquet' (FORMAT parquet);
```

The flags for setting compression, row group size, etc. are listed in the Reading and Writing Parquet files page.

Querying Parquet Files

To run a query directly on a Parquet file, use the `read_parquet` function in the `FROM` clause of a query.

```
SELECT * FROM read_parquet('input.parquet');
```

The Parquet file will be processed in parallel. Filters will be automatically pushed down into the Parquet scan, and only the relevant columns will be read automatically.

For more information see the blog post "Querying Parquet with Precision using DuckDB".

File Access with the file: Protocol

DuckDB supports using the `file:` protocol. It currently supports the following formats:

- `file:/some/path` (host omitted completely)
- `file:///some/path` (empty host)
- `file://localhost/some/path` (`localhost` as host)

Note that the following formats are *not* supported because they are non-standard:

- `file:some/relative/path` (relative path)
- `file://some/path` (double-slash path)

Additionally, the `file:` protocol currently does not support remote (non-localhost) hosts.

Meta Queries

Describe

Describing a Table

To view the schema of a table, use the DESCRIBE statement (or its alias DESC) followed by the table name.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j VARCHAR);
DESCRIBE tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	NO	PRI	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

> **Deprecated.** Describing a table with `SHOW tbl` is deprecated and emits a warning. Use `DESCRIBE tbl` instead. The `show_behavior` setting controls whether SHOW resolves a bare identifier to a table or a setting.

Describing a Query

To view the schema of the result of a query, prepend DESCRIBE to a query.

```
DESCRIBE SELECT * FROM tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	NO	PRI	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

Using DESCRIBE in a Subquery

DESCRIBE can be used as a subquery. This allows creating a table from the description, for example:

```
CREATE TABLE tbl_description AS SELECT * FROM (DESCRIBE tbl);
```

Describing Remote Tables

It is possible to describe remote tables via the `httpfs` extension using the `DESCRIBE TABLE` statement. For example:

```
DESCRIBE TABLE 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';
```

column_name	column_type	null	key	default	extra
season_num	BIGINT	YES	NULL	NULL	NULL
episode_num	BIGINT	YES	NULL	NULL	NULL
aired_date	DATE	YES	NULL	NULL	NULL
cnt_kirk_hookups	BIGINT	YES	NULL	NULL	NULL
cnt_downed_redshirts	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_planet	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_enterprise	BIGINT	YES	NULL	NULL	NULL
cnt_vulcan_nerve_pinch	BIGINT	YES	NULL	NULL	NULL
cnt_warp_speed_orders	BIGINT	YES	NULL	NULL	NULL
highest_warp_speed_issued	BIGINT	YES	NULL	NULL	NULL
bool_hand_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_photon_torpedos_fired	BIGINT	YES	NULL	NULL	NULL
cnt_transporter_pax	BIGINT	YES	NULL	NULL	NULL
cnt_damn_it_jim_quote	BIGINT	YES	NULL	NULL	NULL
cnt_im_givin_her_all_shes_got_quote	BIGINT	YES	NULL	NULL	NULL
cnt_highly_illogical_quote	BIGINT	YES	NULL	NULL	NULL
bool_enterprise_saved_the_day	BIGINT	YES	NULL	NULL	NULL

EXPLAIN: Inspect Query Plans

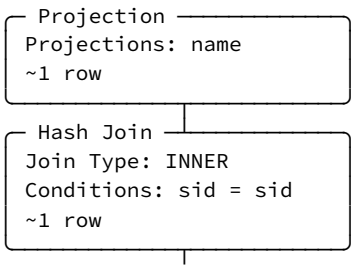
```sql
EXPLAIN SELECT * FROM tbl;
```

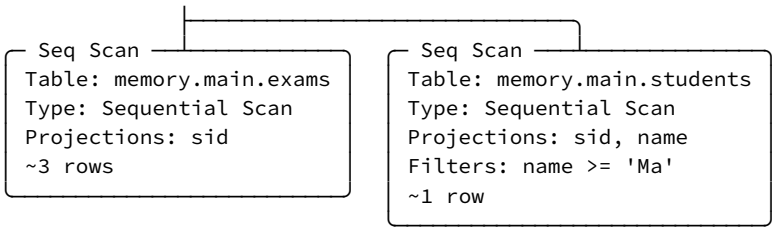
The `EXPLAIN` statement displays the physical plan, i.e., the query plan that will get executed, and is enabled by prepending the query with `EXPLAIN`. The physical plan is a tree of operators that are executed in a specific order to produce the result of the query. To generate an efficient physical plan, the query optimizer transforms the existing physical plan into a better physical plan.

To demonstrate, see the below example:

```sql
CREATE TABLE students (name VARCHAR, sid INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);

EXPLAIN
    SELECT name
    FROM students
    JOIN exams USING (sid)
    WHERE name LIKE 'Ma%';
```





Note that the query is not actually executed – therefore, we can only see the estimated cardinality for each operator (shown with a ~ prefix, e.g., `~3 rows`), which is calculated by using the statistics of the base tables and applying heuristics for each operator.

Table scan operators display the fully qualified table name including catalog and schema (e.g., `memory.main.exams`).

Additional Explain Settings

The `EXPLAIN` statement supports additional settings that can be used to control the output. The following settings are available:

The default setting. Only shows the physical plan.

```
SET explain_output = 'physical_only';
```

Shows only the optimized plan.

```
SET explain_output = 'optimized_only';
```

Shows both the physical and optimized plans.

```
SET explain_output = 'all';
```

See Also

For more information, see the "Profiling" page.

EXPLAIN ANALYZE: Profile Queries

Prepending a query with `EXPLAIN ANALYZE` both pretty-prints the query plan, and executes it, providing run-time performance numbers and the actual cardinality (number of rows produced) for every operator.

```
EXPLAIN ANALYZE SELECT * FROM tbl;
```

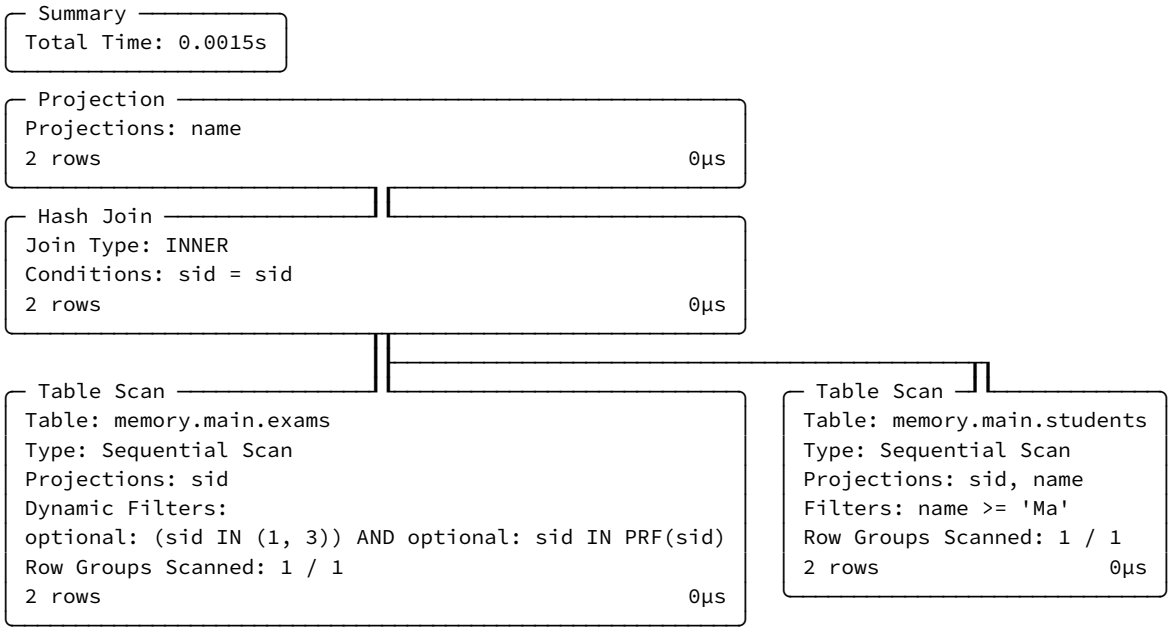
Note that the **cumulative** wall-clock time that is spent on every operator is shown. When multiple threads are processing the query in parallel, the total processing time of the query may be lower than the sum of all the times spent on the individual operators.

For multi-file reads (e.g., reading multiple Parquet files), the output includes the file names being read.

Below is an example of running `EXPLAIN ANALYZE` on a query:

```
CREATE TABLE students (name VARCHAR, sid INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);

EXPLAIN ANALYZE
    SELECT name
    FROM students
    JOIN exams USING (sid)
    WHERE name LIKE 'Ma%';
```



Each operator shows the number of rows it produced and the time spent in it. Scan operators also show the number of row groups scanned and any *dynamic filters* that were pushed into them at runtime, e.g., by the hash join.

See Also

For more information, see the "Profiling" page.

List Tables

The `SHOW TABLES` command can be used to obtain a list of all tables within the selected schema.

```
CREATE TABLE tbl (i INTEGER);
SHOW TABLES;
```

name
tbl

`SHOW` or `SHOW ALL TABLES` can be used to obtain a list of all tables within **all** attached databases and schemas.

```
CREATE TABLE tbl (i INTEGER);
CREATE SCHEMA s1;
CREATE TABLE s1.tbl (v VARCHAR);
SHOW ALL TABLES;
```

database	schema	name	column_names	column_types	temporary
memory	main	tbl	[i]	[INTEGER]	false
memory	s1	tbl	[v]	[VARCHAR]	false

`SHOW TABLES FROM db` can be used to list all tables in a given database or schema.

```
ATTACH 'db.duckdb';
CREATE TABLE db.main_tbl (u VARCHAR);
CREATE SCHEMA db.s1;
CREATE TABLE db.s1.schema_tbl (v VARCHAR);
SHOW TABLES FROM db;
```

name
main_tbl
schema_tbl

Or a specific schema.

```
SHOW TABLES FROM db.s1;
```

name
schema_tbl

To view the schema of an individual table, use the DESCRIBE command.

See Also

The SQL-standard `information_schema` views are also defined. Moreover, DuckDB defines `sqlite_master` and many PostgreSQL system catalog tables for compatibility with SQLite and PostgreSQL respectively.

Summarize

The `SUMMARIZE` command can be used to easily compute a number of aggregates over a table or a query. The `SUMMARIZE` command launches a query that computes a number of aggregates over all columns (`min`, `max`, `approx_unique`, `avg`, `std`, `q25`, `q50`, `q75`, `count`), and return these along the column name, column type, and the percentage of `NULL` values in the column. Note that the quantiles and percentiles are **approximate values**.

Usage

To summarize the contents of a table, use `SUMMARIZE` followed by the table name.

```
SUMMARIZE tbl;
```

To summarize a query, prepend `SUMMARIZE` to a query.

```
SUMMARIZE SELECT * FROM tbl;
```

Example

Below is an example of `SUMMARIZE` on the `lineitem` table of TPC-H `SF1` table, generated using the `tpch` extension.

```
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 1);

SUMMARIZE lineitem;
```

column_name	column_type	min	max	approx_unique	avg	std	q25	q50	q75	count	null_percentage
l_orderkey	INTEGER	1	6000000	1508227	3000279.60420	1732187.87348	1500	3000	4500	60012150	0.00
l_partkey	INTEGER	1	200000	202598	100017.98932	57735.69082	50013	99992	150039	60012150	0.00
l_suppkey	INTEGER	1	10000	10061	5000.60260	2886.96199	2514	4999	7500	60012150	0.00
l_linenumber	INTEGER	1	7	7	3.00057	2.00312	1	3	4	60012150	0.00
l_quantity	DECIMAL(15,2)	1.00	50.00	50	25.50796	14.42762	13	26	38	60012150	0.00
l_extendedprice	DECIMAL(15,2)	901.00	104949.50	923139	38255.13	23054.43	17556	36724	55159	60012150	0.00
l_discount	DECIMAL(15,2)	0.00	0.10	11	0.04999	0.03163	0	0	0	60012150	0.00
l_tax	DECIMAL(15,2)	0.00	0.08	9	0.04001	0.02821	0	0	0	60012150	0.00
l_returnflag	VARCHAR	A	R	3	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_linestatus	VARCHAR	F	O	2	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_shipdate	DATE	1992-01-02	1998-12-01	2516	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_commitdate	DATE	1992-01-31	1998-10-31	2460	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_receiptdate	DATE	1992-01-04	1998-12-31	2549	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_shipinstruct	VARCHAR	COLLECT COD	TAKE BACK RETURN	4	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_shipmode	VARCHAR	AIR	TRUCK	7	NULL	NULL	NULL	NULL	NULL	60012150	0.00
l_comment	VARCHAR	Tiresias	zzle? furiously iro	3558599	NULL	NULL	NULL	NULL	NULL	60012150	0.00

Using SUMMARIZE in a Subquery

`SUMMARIZE` can be used as a subquery. This allows creating a table from the summary, for example:

```
CREATE TABLE tbl_summary AS SELECT * FROM (SUMMARIZE tbl);
```

Summarizing Remote Tables

It is possible to summarize remote tables via the `httpfs` extension using the `SUMMARIZE TABLE` statement. For example:

```
SUMMARIZE TABLE 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';
```

DuckDB Environment

DuckDB provides a number of functions and PRAGMA options to retrieve information on the running DuckDB instance and its environment.

Version

The `version()` function returns the version number of DuckDB.

```
SELECT version() AS version;
```

version
v2.0.0

Using a PRAGMA:

```
PRAGMA version;
```

library_version	source_id	codename
v2.0.0	d54cbbe3df	Cyanoptera

Platform

The platform information consists of the operating system, system architecture, and, optionally, the compiler. The platform is used when installing extensions. To retrieve the platform, use the following PRAGMA:

```
PRAGMA platform;
```

On macOS, running on Apple Silicon architecture, the result is:

platform
osx_arm64

On Windows, running on an AMD64 architecture, the platform is `windows_amd64`. On Ubuntu Linux, running on the ARM64 architecture, the platform is `linux_arm64`.

Extensions

To get a list of DuckDB extensions and their status (e.g., `loaded`, `installed`), use the `duckdb_extensions()` function:

```
SELECT *
FROM duckdb_extensions();
```

Meta Table Functions

DuckDB has the following built-in table functions to obtain metadata about available catalog objects:

- `duckdb_columns()`: columns
- `duckdb_constraints()`: constraints
- `duckdb_databases()`: lists the databases that are accessible from within the current DuckDB process
- `duckdb_dependencies()`: dependencies between objects
- `duckdb_extensions()`: extensions
- `duckdb_functions()`: functions
- `duckdb_indexes()`: secondary indexes
- `duckdb_keywords()`: DuckDB's keywords and reserved words
- `duckdb_optimizers()`: the available optimization rules in the DuckDB instance
- `duckdb_schemas()`: schemas
- `duckdb_sequences()`: sequences
- `duckdb_settings()`: settings
- `duckdb_tables()`: base tables
- `duckdb_temporary_files()`: the temporary files DuckDB has written to disk, to offload data from memory
- `duckdb_types()`: data types
- `duckdb_views()`: views

Network and Cloud Storage

Network and Cloud Storage

HTTP Parquet Import

To load a Parquet file over HTTP(S), the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After the `httpfs` extension is set up, Parquet files can be read over `http(s)`:

```
SELECT * FROM read_parquet('https://<domain>/path/to/file.parquet');
```

For example:

```
SELECT * FROM read_parquet('https://duckdb.org/data/prices.parquet');
```

Moreover, the `read_parquet` function itself can also be omitted thanks to DuckDB's replacement scan mechanism:

```
SELECT * FROM 'https://duckdb.org/data/holdings.parquet';
```

HTTP CSV Import

To load a CSV file directly over HTTP(S), the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After the `httpfs` extension is set up, CSV files can be read over `http(s)` the same way as a local file, using `read_csv`:

```
SELECT * FROM read_csv('https://<domain>/path/to/file.csv');
```

Moreover, the `read_csv` function itself can also be omitted thanks to DuckDB's replacement scan mechanism:

```
SELECT * FROM 'https://<domain>/path/to/file.csv';
```

Example

The following query reads an openly licensed (CC BY 4.0) real-world CSV directly over HTTP and aggregates it, with no local download step:

```
SELECT
    area,
    round(avg(sunbed_price), 0) AS avg_sunbed_price_eur,
    count(*) AS clubs
FROM 'http://blobs.duckdb.org/data/beach-clubs.csv'
WHERE sunbed_price IS NOT NULL
GROUP BY area
ORDER BY avg_sunbed_price_eur DESC;
```

area varchar	avg_sunbed_price_eur double	clubs int64
Marbella centre (El Cable)	120.0	1
Puerto Banús	103.0	4
Golden Mile	70.0	1
El Rosario (Marbella East)	30.0	1
Elviria (Marbella East)	30.0	1
Marbella centre	20.0	1
Casablanca (Golden Mile)	15.0	1

Because `read_csv` streams the file rather than materializing it first, this works the same way for CSVs far larger than fit in memory, and pairs naturally with `read_csv`'s usual options (`columns`, `types`, `delim`, etc.) for files that need explicit schema hints.

S3 Parquet Import

Prerequisites

To load a Parquet file from S3, the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

Credentials and Configuration

After loading the `httpfs` extension, set up the credentials and S3 region to read data:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your-region>.amazonaws.com'`{:.language-sql .highlight}.

To fetch credentials automatically instead — or to use a named profile, SSO, or an assumed role — see the `aws` extension. The default looks like:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Querying

After the `httpfs` extension is set up and the S3 configuration is set correctly, Parquet files can be read from S3 using the following command:

```
SELECT * FROM read_parquet('s3://<bucket>/<file>');
```

Google Cloud Storage (GCS) and Cloudflare R2

DuckDB can also handle Google Cloud Storage (GCS) and Cloudflare R2 via the S3 API. See the relevant guides for details.

S3 Parquet Export

To write a Parquet file to S3, the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After loading the `httpfs` extension, set up the credentials to write data. Note that the `region` parameter should match the region of the bucket you want to access.

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your-region>.amazonaws.com'`{:.language-sql .highlight}.

To fetch credentials automatically instead — or to use a named profile, SSO, or an assumed role — see the aws extension. The default looks like:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

After the `httpfs` extension is set up and the S3 credentials are correctly configured, Parquet files can be written to S3 using the following command:

```
COPY <table_name> TO 's3://<s3-bucket>/<filename>.parquet';
```

Similarly, Google Cloud Storage (GCS) is supported through the Interoperability API. You need to create HMAC keys and provide the credentials as follows:

```
CREATE SECRET (
    TYPE gcs,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>'
);
```

After setting up the GCS credentials, you can export using:

```
COPY <table_name> TO 'gs://<gcs_bucket>/<filename>.parquet';
```

S3 Iceberg Import

Prerequisites

Loading an Iceberg file from S3 requires both the `httpfs` and `iceberg` extensions. Install them using the `INSTALL` SQL command. You only need to install extensions once.

```
INSTALL httpfs;
INSTALL iceberg;
```

To load the extensions, use the `LOAD` command:

```
LOAD httpfs;
LOAD iceberg;
```

Credentials

After loading the extensions, set up the credentials and S3 region to read data. You may either use an access key and secret, or a token.

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

To fetch credentials automatically instead — or to use a named profile, SSO, or an assumed role — see the aws extension. The default looks like:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Loading Iceberg Tables from S3

After the extensions are set up and the S3 credentials are correctly configured, Iceberg tables can be read from S3 using the following command:

```
SELECT *
FROM iceberg_scan('s3://<bucket>/<iceberg_table_folder>/metadata/<id>.metadata.json');
```

If you pass the table folder instead of the metadata file, DuckDB looks for a `version-hint.text` file in the table's `metadata` directory. If there is no such file, you'll get an error like this:

```
Invalid Configuration Error: Failed to read iceberg table. No version was provided and no version-hint
could be found, globbing the filesystem to locate the latest version is disabled by default as this is
considered unsafe and could result in reading uncommitted data. To enable this use 'SET unsafe_enable_
version_guessing = true;'
```

In this case, link directly to the metadata file or see Selecting Metadata Versions.

S3 Express One

In late 2023, AWS announced the S3 Express One Zone, a high-speed variant of traditional S3 buckets. DuckDB can read S3 Express One buckets using the `httpfs` extension.

Credentials and Configuration

The configuration of S3 Express One buckets is similar to regular S3 buckets with one exception: you must specify the endpoint according to the following pattern:

```
s3express-<availability_zone>.<region>.amazonaws.com
```

where the `<availability_zone>`{:.language-sql .highlight} (e.g., `use-az5`) can be obtained from the S3 Express One bucket's configuration page and the `<region>`{:.language-sql .highlight} is the AWS region (e.g., `us-east-1`).

For example, to allow DuckDB to use an S3 Express One bucket, configure the Secrets manager as follows:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>',
    ENDPOINT 's3express-<use1-az5>.<us-east-1>.amazonaws.com'
);
```

Instance Location

For best performance, ensure the EC2 instance is in the same availability zone as the S3 Express One bucket you are querying. To determine the mapping between zone names and zone IDs, use the `aws ec2 describe-availability-zones` command.

- Zone name to zone ID mapping:

  ```
  aws ec2 describe-availability-zones --output json \
      | jq -r '.AvailabilityZones[] | select(.ZoneName == "us-east-1f") | .ZoneId'
  ```

  ```
  use1-az5
  ```

- Zone ID to zone name mapping:

  ```
  aws ec2 describe-availability-zones --output json \
      | jq -r '.AvailabilityZones[] | select(.ZoneId == "use1-az5") | .ZoneName'
  ```

  ```
  us-east-1f
  ```

Querying

You can query the S3 Express One bucket like any other S3 bucket:

```
SELECT *
FROM 's3://express-bucket-name--use1-az5--x-s3/my-file.parquet';
```

Performance

The following experiments were run on a `c7gd.12xlarge` instance using the LDBC SF300 Comments `creationDate` Parquet file (also used in the microbenchmarks of the performance guide).

Experiment	File size	Runtime
Loading only from Parquet	4.1 GB	3.5 s
Creating local table from Parquet	4.1 GB	5.1 s

The "loading only" variant is running the load as part of an `EXPLAIN ANALYZE` statement to measure the runtime without actually creating a local table, while the "creating local table" variant uses `CREATE TABLE ... AS SELECT` to create a persistent table on the local disk.

Google Cloud Storage Import

Prerequisites

The Google Cloud Storage (GCS) can be used via the `httpfs` extension. This can be installed with the `INSTALL httpfs` SQL command. This only needs to be run once.

Credentials and Configuration

You need to create HMAC keys and declare them:

```
CREATE SECRET (
    TYPE gcs,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>'
);
```

Querying

After setting up the GCS credentials, you can query the GCS data using:

```
SELECT *
FROM read_parquet('gs://<gcs_bucket>/<file.parquet>');
```

Attaching to a Database

You can attach to a database file in read-only mode:

```
LOAD httpfs;
ATTACH 'gs://<gcs_bucket>/<file.duckdb>' AS <duckdb_database> (READ_ONLY);
```

> Databases in Google Cloud Storage can only be attached in read-only mode.

Cloudflare R2 Import

Prerequisites

For Cloudflare R2, the S3 Compatibility API allows you to use DuckDB's S3 support to read and write from R2 buckets.

This requires the `httpfs` extension, which can be installed using the `INSTALL` SQL command. This only needs to be run once.

Credentials and Configuration

You will need to generate an S3 auth token and create an R2 secret in DuckDB:

```
CREATE SECRET (
    TYPE r2,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    ACCOUNT_ID '<your-32-character-hexadecimal-account-ID>'
);
```

Querying

After setting up the R2 credentials, you can query the R2 data using DuckDB's built-in methods, such as `read_csv` or `read_parquet`:

```
SELECT * FROM read_parquet('r2://<r2-bucket-name>/<file>');
```

Attach to a DuckDB Database over HTTPS or S3

You can establish a read-only connection to a DuckDB instance via HTTPS or the S3 API.

Prerequisites

This guide requires the `httpfs` extension, which can be installed using the `INSTALL httpfs` SQL command. This only needs to be run once.

Attaching to a Database over HTTPS

To connect to a DuckDB database via HTTPS, use the ATTACH statement as follows:

```
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db;
```

Then, the database can be queried using:

```
SELECT count(*) AS num_stations
FROM stations_db.stations;
```

num_stations
578

Attaching to a Database over the S3 API

To connect to a DuckDB database via the S3 API, configure the authentication for your bucket (if required). Then, use the ATTACH statement as follows:

```
ATTACH 's3://duckdb-blobs/databases/stations.duckdb' AS stations_db;
```

The database can be queried using:

```
SELECT count(*) AS num_stations
FROM stations_db.stations;
```

num_stations
578

> Connecting to S3-compatible APIs such as the Google Cloud Storage (`gs://`) is also supported.

Limitations

- Only read-only connections are allowed, writing the database via the HTTPS protocol or the S3 API is not possible.

Share a DuckDB Database That Only Stores Views

A DuckDB database file does not have to contain any data: it can store only view definitions that point at remote data such as Parquet files on object storage. The resulting file is only a few hundred kilobytes, yet it behaves like a shared catalog: anyone who attaches it sees a set of named, ready-to-query relations, while the underlying data stays in its original location and format.

This is useful for publishing a curated "semantic layer" over a data lake, where the view definitions encode the joins, filters, and column names that consumers should use, without copying or moving the data itself.

Prerequisites

This guide requires the `httpfs` extension, which can be installed using the `INSTALL httpfs` SQL command. This only needs to be run once.

Building the Catalog

Create a persistent database file and define views that reference the remote data. Use fully qualified locations (remote URLs or absolute paths) so that the views resolve regardless of the client's working directory:

```
ATTACH 'catalog.duckdb' AS catalog;
USE catalog;

CREATE VIEW services AS
    SELECT *
    FROM read_parquet('s3://my-bucket/rail/services/*.parquet');

CREATE VIEW stations AS
    SELECT *
    FROM read_parquet('s3://my-bucket/rail/stations.parquet');
```

Because the file stores only view definitions, it stays small no matter how large the referenced datasets are.

Hosting the Catalog

Upload the `catalog.duckdb` file to any location that DuckDB can read over the network, for example an `s3://` bucket or an `https://` endpoint.

Querying the Catalog

Consumers attach the catalog read-only and query the views as if they were local tables. Each query reads the current data from the referenced remote files:

```
ATTACH 'https://example.com/catalog.duckdb' AS catalog (READ_ONLY);

SELECT count(*) AS num_stations
FROM catalog.stations;
```

Consumers need the `httpfs` extension and, for `s3://` sources, credentials that grant access to the underlying data.

Limitations

- Consumers must be able to access both the catalog file and every data source that its views reference.
- Connections over HTTPS and the S3 API are read-only, so the catalog cannot be modified in place.
- Query performance depends on the network and the layout of the referenced files. The tips for querying remote files apply.

Fastly Object Storage Import

Prerequisites

For Fastly Object Storage, the S3 Compatibility API allows you to use DuckDB's S3 support to read and write from Fastly buckets.

This requires the `httpfs` extension, which can be installed using the `INSTALL` SQL command. This only needs to be run once.

Credentials and Configuration

You will need to generate an S3 auth token and create an S3 secret in DuckDB:

```
CREATE SECRET my_secret (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    URL_STYLE 'path',
    REGION '<us-east>',
    ENDPOINT '<us-east>.object.fastlystorage.app' -- see note below
);
```

- The `ENDPOINT` needs to point to the Fastly endpoint for the region you want to use (e.g., `eu-central.object.fastlystorage.app`).
- `REGION` must use the same region mentioned in `ENDPOINT`.
- `URL_STYLE` needs to use `path`.

Querying

After setting up the Fastly Object Storage credentials, you can query the data there using DuckDB's built-in methods, such as `read_csv` or `read_parquet`:

```
SELECT * FROM 's3://<fastly-bucket-name>/(file).csv';
SELECT * FROM read_parquet('s3://<fastly-bucket-name>/<file>.parquet');
```

SeaweedFS Import

Prerequisites

For SeaweedFS, the S3-compatible gateway allows you to use DuckDB's S3 support to read and write from SeaweedFS buckets.

This requires the `httpfs` extension, which can be installed using the `INSTALL` SQL command. This only needs to be run once.

To try SeaweedFS locally, create a file `s3config.json` with the S3 credentials:

```
{
  "identities": [
    {
      "name": "analyst",
      "credentials": [
        {
          "accessKey": "your_access_key_id",
          "secretKey": "your_secret_access_key"
        }
      ],
      "actions": ["Admin", "Read", "Write", "List", "Tagging"]
    }
  ]
}
```

Then start the whole SeaweedFS stack in one container and create a bucket:

```
docker run -d --name seaweedfs -p 8333:8333 \
    -v "$(pwd)/s3config.json:/etc/seaweedfs/s3config.json" \
    chrislusf/seaweedfs:latest \
    mini -dir=/data -s3.config=/etc/seaweedfs/s3config.json

echo "s3.bucket.create -name <your-bucket>" | \
    docker exec -i seaweedfs weed shell -master=localhost:9333
```

The S3 endpoint listens on port 8333.

Credentials and Configuration

Create an S3 secret with the credentials from the SeaweedFS S3 configuration:

```
CREATE SECRET my_secret (
    TYPE s3,
    KEY_ID '<your_access_key_id>',
    SECRET '<your_secret_access_key>',
    ENDPOINT '<seaweedfs-host>:8333',
    URL_STYLE 'path',
    USE_SSL false
);
```

- SeaweedFS serves path-style requests, so set `URL_STYLE` to `path`. No wildcard DNS is needed.
- SeaweedFS does not require a region, so `REGION` can be omitted.
- Set `USE_SSL false` for a plain-HTTP endpoint such as the local setup above; omit it when the gateway runs behind TLS.

Querying

After setting up the SeaweedFS credentials, you can query the data using DuckDB's built-in methods, such as `read_csv` or `read_parquet`:

```
SELECT * FROM 's3://<your-bucket>/<file>.csv';
SELECT * FROM read_parquet('s3://<your-bucket>/<file>.parquet');
```

Writing works the same way, including glob reads over the result:

```
COPY (SELECT 42 AS answer) TO 's3://<your-bucket>/sample/answer.parquet';
SELECT * FROM read_parquet('s3://<your-bucket>/sample/*.parquet');
```

SeaweedFS also supports Iceberg: table buckets store the table data as Parquet files, and the gateway's built-in Iceberg REST Catalog serves the table metadata. See the SeaweedFS catalog example to query Iceberg tables through it.

Tigris Import

Prerequisites

For Tigris, the S3-compatible API allows you to use DuckDB's S3 support to read and write from Tigris buckets.

This requires the `httpfs` extension, which can be installed using the `INSTALL` SQL command. This only needs to be run once.

Credentials and Configuration

You will need to generate an access key pair and create an S3 secret in DuckDB:

```
CREATE SECRET my_secret (
    TYPE s3,
    KEY_ID '<tid_xxxxxxxxxxxx>',
    SECRET '<tsec_xxxxxxxxxxxxxxxxxxxxxxxx>',
    REGION 'auto',
    ENDPOINT 'fly.storage.tigris.dev'
);
```

- A single endpoint (`fly.storage.tigris.dev`) serves all regions; requests are routed to the Tigris region nearest the caller. `REGION` is required for request signing but is not used for routing — set it to `auto`.
- `URL_STYLE` does not need to be set. Tigris uses virtual-hosted-style URLs, which is DuckDB's default for `TYPE s3`.

> **Tip.** When DuckDB runs on a Fly.io Machine, requests to `fly.storage.tigris.dev` stay on Fly's internal network and are served from the same region as the Machine when possible.

Querying

After setting up the Tigris credentials, you can query the data using DuckDB's built-in methods, such as `read_csv` or `read_parquet`:

```
SELECT * FROM 's3://<tigris-bucket-name>/<file>.csv';
SELECT * FROM read_parquet('s3://<tigris-bucket-name>/<file>.parquet');
```

ODBC

ODBC 101: A Duck Themed Guide to ODBC

What is ODBC?

ODBC which stands for Open Database Connectivity, is a standard that allows different programs to talk to different databases including, of course, DuckDB. This makes it easier to build programs that work with many different databases, which saves time as developers don't have to write custom code to connect to each database. Instead, they can use the standardized ODBC interface, which reduces development time and costs, and programs are easier to maintain. However, ODBC can be slower than other methods of connecting to a database, such as using a native driver, as it adds an extra layer of abstraction between the application and the database. Furthermore, because DuckDB is column-based and ODBC is row-based, there can be some inefficiencies when using ODBC with DuckDB.

> There are links throughout this page to the official Microsoft ODBC documentation, which is a great resource for learning more about ODBC.

General Concepts

- Handles
- Connecting
- Error Handling and Diagnostics
- Buffers and Binding

Handles

A handle is a pointer to a specific ODBC object which is used to interact with the database. There are several different types of handles, each with a different purpose, these are the environment handle, the connection handle, the statement handle, and the descriptor handle. Handles are allocated using the `SQLAllocHandle` which takes as input the type of handle to allocate, and a pointer to the handle, the driver then creates a new handle of the specified type, which it returns to the application.

The DuckDB ODBC driver has the following handle types.

Environment

Handle name	Environment
Type name	`SQL_HANDLE_ENV`
:--	:--------
Description	Manages the environment settings for ODBC operations, and provides a global context in which to access data.
Use case	Initializing ODBC, managing driver behavior, resource allocation.
Additional information	Must be allocated once per application upon starting, and freed at the end.

Connection

Handle name	Connection
Type name	SQL_HANDLE_DBC
:--	:--------
Description	Represents a connection to a data source. Used to establish, manage, and terminate connections. Defines both the driver and the data source to use within the driver.
Use case	Establishing a connection to a database, managing the connection state.
Additional information	Multiple connection handles can be created as needed, allowing simultaneous connections to multiple data sources. *Note:* Allocating a connection handle does not establish a connection, but must be allocated first, and then used once the connection has been established.

Statement

Handle name	Statement
Type name	SQL_HANDLE_STMT
:--	:--------
Description	Handles the execution of SQL statements, as well as the returned result sets.
Use case	Executing SQL queries, fetching result sets, managing statement options.
Additional information	To facilitate the execution of concurrent queries, multiple handles can be allocated per connection.

Descriptor

Handle name	Descriptor
Type name	SQL_HANDLE_DESC
:--	:--------
Description	Describes the attributes of a data structure or parameter, and allows the application to specify the structure of data to be bound/retrieved.
Use case	Describing table structures, result sets, binding columns to application buffers.
Additional information	Used in situations where data structures need to be explicitly defined, for example during parameter binding or result set fetching. They are automatically allocated when a statement is allocated, but can also be allocated explicitly.

Connecting

The first step is to connect to the data source so that the application can perform database operations. First the application must allocate an environment handle, and then a connection handle. The connection handle is then used to connect to the data source. There are two functions which can be used to connect to a data source, `SQLDriverConnect` and `SQLConnect`. The former is used to connect to a data source using a connection string, while the latter is used to connect to a data source using a DSN.

Connection String A connection string is a string which contains the information needed to connect to a data source. It is formatted as a semicolon separated list of key-value pairs, e.g., `DSN=DuckDB;access_mode=READ_ONLY`, where the keys can be the DSN, `Database` or any of the DuckDB configuration options.

DSN A DSN (*Data Source Name*) is a string that identifies a database. It can be a file path, URL, or a database name. For example, `C:\Users\me\duckdb.db` and `DuckDB` are both valid DSNs. More information on DSNs can be found on the "Choosing a Data Source or Driver" page of the SQL Server documentation.

Error Handling and Diagnostics

All functions in ODBC return a code which represents the success or failure of the function. This allows for easy error handling, as the application can simply check the return code of each function call to determine if it was successful. When unsuccessful, the application can then use the `SQLGetDiagRec` function to retrieve the error information. The following table defines the return codes:

Return code	Description
`SQL_SUCCESS`	The function completed successfully.
`SQL_SUCCESS_WITH_INFO`	The function completed successfully, but additional information is available, including a warning.
`SQL_ERROR`	The function failed.
`SQL_INVALID_HANDLE`	The handle provided was invalid, indicating a programming error, i.e., when a handle is not allocated before it is used, or is the wrong type.
`SQL_NO_DATA`	The function completed successfully, but no more data is available.
`SQL_NEED_DATA`	More data is needed, such as when a parameter data is sent at execution time, or additional connection information is required.
`SQL_STILL_EXECUTING`	A function that was asynchronously executed is still executing.

Buffers and Binding

A buffer is a block of memory used to store data. Buffers are used to store data retrieved from the database, or to send data to the database. Buffers are allocated by the application, and then bound to a column in a result set, or a parameter in a query, using the `SQLBindCol` and `SQLBindParameter` functions. When the application fetches a row from the result set, or executes a query, the data is stored in the buffer. When the application sends a query to the database, the data in the buffer is sent to the database.

Setting up an Application

The following is a step-by-step guide to setting up an application that uses ODBC to connect to a database, execute a query, and fetch the results in C++.

> To install the driver as well as anything else you will need follow these instructions.

1. Include the SQL Header Files

The first step is to include the SQL header files:

```
#include <sql.h>
#include <sqlext.h>
```

These files contain the definitions of the ODBC functions, as well as the data types used by ODBC. To use these header files you have to have the `unixodbc` package installed:

On macOS:

```
brew install unixodbc
```

On Ubuntu and Debian:

```
sudo apt-get install -y unixodbc-dev
```

On Fedora, CentOS, and Red Hat:

```
sudo yum install -y unixODBC-devel
```

Remember to include the header file location in your `CFLAGS`.

For `MAKEFILE`:

```
CFLAGS=-I/usr/local/include
# or
CFLAGS=-I/opt/homebrew/Cellar/unixodbc/2.3.11/include
```

For `CMAKE`:

```
include_directories(/usr/local/include)
# or
include_directories(/opt/homebrew/Cellar/unixodbc/2.3.11/include)
```

You also have to link the library in your `CMAKE` or `MAKEFILE`. For `CMAKE`:

```
target_link_libraries(ODBC_application /path/to/duckdb_odbc/libduckdb_odbc.dylib)
```

For `MAKEFILE`:

```
LDLIBS=-L/path/to/duckdb_odbc/libduckdb_odbc.dylib
```

2. Define the ODBC Handles and Connect to the Database

2.a. Connecting with SQLConnect Then set up the ODBC handles, allocate them, and connect to the database. First the environment handle is allocated, then the environment is set to ODBC version 3, then the connection handle is allocated, and finally the connection is made to the database. The following code snippet shows how to do this:

```
SQLHANDLE env;
SQLHANDLE dbc;

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);

SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

std::string dsn = "DSN=duckdbmemory";
SQLConnect(dbc, (SQLCHAR*)dsn.c_str(), SQL_NTS, NULL, 0, NULL, 0);

std::cout << "Connected!" << std::endl;
```

2.b. Connecting with SQLDriverConnect Alternatively, you can connect to the ODBC driver using `SQLDriverConnect`. `SQLDriverConnect` accepts a connection string in which you can configure the database using any of the available DuckDB configuration options.

```
SQLHANDLE env;
SQLHANDLE dbc;

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);

SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

SQLCHAR str[1024];
SQLSMALLINT strl;
std::string dsn = "DSN=DuckDB;access_mode=READ_ONLY";
SQLDriverConnect(dbc, nullptr, (SQLCHAR*)dsn.c_str(), SQL_NTS, str, sizeof(str), &strl, SQL_DRIVER_
COMPLETE);

std::cout << "Connected!" << std::endl;
```

3. Adding a Query

Now that the application is set up, we can add a query to it. First, we need to allocate a statement handle:

```
SQLHANDLE stmt;
SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
```

Then we can execute a query:

```
SQLExecDirect(stmt, (SQLCHAR*)"SELECT * FROM integers", SQL_NTS);
```

4. Fetching Results

Now that we have executed a query, we can fetch the results. First, we need to bind the columns in the result set to buffers:

```
SQLLEN int_val;
SQLLEN null_val;
SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
```

Then we can fetch the results:

```
SQLFetch(stmt);
```

5. Process the Results

Now that we have the results, we can do whatever we want with them. For example, we can print them:

```
std::cout << "Value: " << int_val << std::endl;
```

You can also execute additional queries and perform other database operations such as inserting, updating, or deleting data.

6. Free the Handles and Disconnecting

Finally, we need to free the handles and disconnect from the database. First, we need to free the statement handle:

```
SQLFreeHandle(SQL_HANDLE_STMT, stmt);
```

Then we need to disconnect from the database:

```
SQLDisconnect(dbc);
```

And finally, we need to free the connection handle and the environment handle:

```
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
SQLFreeHandle(SQL_HANDLE_ENV, env);
```

Freeing the connection and environment handles can only be done after the connection to the database has been closed. Trying to free them before disconnecting from the database will result in an error.

Sample Application

The following is a sample application that includes a cpp file that connects to the database, executes a query, fetches the results, and prints them. It also disconnects from the database and frees the handles, and includes a function to check the return value of ODBC functions. It also includes a `CMakeLists.txt` file that can be used to build the application.

Sample `.cpp` File

```cpp
#include <iostream>
#include <sql.h>
#include <sqlext.h>

void check_ret(SQLRETURN ret, std::string msg) {
    if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO) {
        std::cout << ret << ": " << msg << " failed" << std::endl;
        exit(1);
    }
    if (ret == SQL_SUCCESS_WITH_INFO) {
        std::cout << ret << ": " << msg << " succeeded with info" << std::endl;
    }
}

int main() {
    SQLHANDLE env;
    SQLHANDLE dbc;
    SQLRETURN ret;

    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    check_ret(ret, "SQLAllocHandle(env)");

    ret = SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
    check_ret(ret, "SQLSetEnvAttr");

    ret = SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
    check_ret(ret, "SQLAllocHandle(dbc)");

    std::string dsn = "DSN=duckdbmemory";
    ret = SQLConnect(dbc, (SQLCHAR*)dsn.c_str(), SQL_NTS, NULL, 0, NULL, 0);
    check_ret(ret, "SQLConnect");

    std::cout << "Connected!" << std::endl;

    SQLHANDLE stmt;
    ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
    check_ret(ret, "SQLAllocHandle(stmt)");

    ret = SQLExecDirect(stmt, (SQLCHAR*)"SELECT * FROM integers", SQL_NTS);
    check_ret(ret, "SQLExecDirect(SELECT * FROM integers)");

    SQLLEN int_val;
    SQLLEN null_val;
    ret = SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
    check_ret(ret, "SQLBindCol");

    ret = SQLFetch(stmt);
    check_ret(ret, "SQLFetch");

    std::cout << "Value: " << int_val << std::endl;
```

```
    ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);
    check_ret(ret, "SQLFreeHandle(stmt)");

    ret = SQLDisconnect(dbc);
    check_ret(ret, "SQLDisconnect");

    ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
    check_ret(ret, "SQLFreeHandle(dbc)");

    ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
    check_ret(ret, "SQLFreeHandle(env)");
}
```

Sample `CMakeLists.txt` File

```
cmake_minimum_required(VERSION 3.25)
project(ODBC_Tester_App)

set(CMAKE_CXX_STANDARD 17)
include_directories(/opt/homebrew/Cellar/unixodbc/2.3.11/include)

add_executable(ODBC_Tester_App main.cpp)
target_link_libraries(ODBC_Tester_App /duckdb_odbc/libduckdb_odbc.dylib)
```

Performance

Performance Guide

DuckDB aims to automatically achieve high performance by using well-chosen default configurations and having a forgiving architecture. Of course, there are still opportunities for tuning the system for specific workloads. The Performance Guide's pages contain guidelines and tips for achieving good performance when loading and processing data with DuckDB.

The guides include several microbenchmarks. You may find details about these on the Benchmarks page.

Environment

The environment where DuckDB is run has an obvious impact on performance. This page focuses on the effects of the hardware configuration and the operating system used.

Hardware Configuration

CPU

DuckDB's officially supported architectures are AMD64 (x86_64) and ARM64 (AArch64) CPU architectures. DuckDB works efficiently on both of these architectures.

> DuckDB can be compiled to other architecture such as LoongArch and RISC-V. However, there are no performance guarantees for these platforms.

Memory

> **Best practice.** Aim for 1-4 GB memory per thread.

Minimum Required Memory As a rule of thumb, DuckDB requires a *minimum* of 125 MB of memory per thread. For example, if you use 8 threads, you need at least 1 GB of memory. If you are working in a memory-constrained environment, consider limiting the number of threads, e.g., by issuing:

```
SET threads = 4;
```

Memory for Ideal Performance The amount of memory required for ideal performance depends on several factors, including the dataset size and the queries to execute. Maybe surprisingly, the *queries* have a larger effect on the memory requirement. Workloads containing large joins over many-to-many tables yield large intermediate datasets and thus require more memory for their evaluation to fully fit into the memory. As an approximation, aggregation-heavy workloads require 1-2 GB memory per thread and join-heavy workloads require 3-4 GB memory per thread.

Larger-than-Memory Workloads DuckDB can process larger-than-memory workloads by spilling to disk. This is possible thanks to *out-of-core* support for grouping, joining, sorting and windowing operators. Note that larger-than-memory workloads can be processed both in persistent mode and in in-memory mode as DuckDB still spills to disk in both modes.

Local Disk

Disk type. DuckDB's disk-based mode is designed to work best with SSD and NVMe disks. While HDDs are supported, they will result in low performance, especially for write operations.

Disk-based vs. in-memory storage. Counter-intuitively, using a disk-based DuckDB instance can be faster than an in-memory instance. This is because the default settings prescribe compression for disk-based storage but they do turn compression off for in-memory storage. Read more in the "How to Tune Workloads" page.

File systems. On Linux, DuckDB performs best with the XFS file system but it also performs well with other file systems such as ext4. On Windows, we recommend using NTFS and avoiding FAT32.

> DuckDB databases have built-in checksums, so integrity checks from the file system are not required to prevent data corruption.

Network-Attached Disks

Special care needs to be taken when using network-attached disks:

- If you are writing to disk, it is important that the disks are reliable. As a general rule of thumb, this is true for locally attached disks, and block storage in the cloud.
- If your workload is larger than memory and/or fast data loading is important, you need fast disks, preferably SSD or NVMe with a fast connection.

With these in mind, here are two common architectures and the related considerations when you are using DuckDB's native database format:

Block storage in the cloud. DuckDB runs well on network-backed cloud disks such as AWS EBS for both read-only and read-write workloads.

Network-attached storage. Network-attached storage can serve DuckDB for read-only workloads. However, *it is recommended to avoid using DuckDB's native database format in read-write mode on network-attached storage (NAS).* These setups include NFS, network drives such as SMB and Samba. Based on user reports, running read-write workloads on network-attached storage can result in slow and unpredictable performance, as well as spurious errors caused by the underlying file system. Instead of using DuckDB's native database format in this setting, consider using the DuckLake lakehouse format. DuckLake keeps the data in immutable Parquet files on object storage and the transactional metadata in a separate catalog database, which avoids the concurrent read-write access patterns on a single large file that network-attached storage handles poorly.

Operating System

We recommend using the latest stable version of operating systems: macOS, Windows, and Linux are all well-tested and DuckDB can run on them with high performance.

Linux

DuckDB runs on all mainstream Linux distributions released in the last ≈5 years. If you don't have a particular preference, we recommend using Ubuntu Linux LTS due to its stability and the fact that most of DuckDB's Linux test suite jobs run on Ubuntu workers.

glibc vs. musl libc We distribute DuckDB builds with both glibc and musl libc. However, note that DuckDB binaries built with musl libc have significantly lower performance. In practice, this can lead to a slowdown of more than 5× on compute-intensive workloads. Therefore, it's recommended to use a Linux distribution with glibc for performance-oriented workloads when running DuckDB.

Memory Allocator

If you have a many-core CPU running on a system where DuckDB ships with `jemalloc` as the default memory allocator, consider enabling the allocator's background threads.

Data Import

Recommended Import Methods

When importing data from other systems to DuckDB, there are several considerations to take into account. We recommend importing using the following order:

1. For systems which are supported by a DuckDB scanner extension, it's preferable to use the scanner. DuckDB currently offers scanners for MySQL, PostgreSQL and SQLite, as well as a generic ODBC scanner.
2. If there is a bulk export feature in the data source system, export the data to Parquet or CSV format, then load it using DuckDB's Parquet or CSV loader.
3. If the approaches above are not applicable, consider using the DuckDB appender, currently available in the C, C++, Go, Java, Julia, Node.js, Python and Rust APIs.

Methods to Avoid

If possible, avoid looping row-by-row (tuple-at-a-time) in favor of bulk operations. Performing row-by-row inserts (even with prepared statements) is detrimental to performance and will result in slow load times.

> **Best practice.** Unless your data is small (<100k rows), avoid using inserts in loops.

Schema

Types

It is important to use the correct type for encoding columns (e.g., `BIGINT`, `DATE`, `DATETIME`). While it is always possible to use string types (VARCHAR, etc.) to encode more specific values, this is not recommended. Strings use more space and are slower to process in operations such as filtering, join, and aggregation. When loading CSV files, you may leverage the CSV reader's auto-detection mechanism to determine the correct types for CSV inputs.

If you run in a memory-constrained environment, using smaller data types (e.g., `TINYINT`) can reduce the amount of memory and disk space required to complete a query. DuckDB's bitpacking compression means small values stored in larger data types will not take up larger sizes on disk, but they will take up more memory during processing – hence, using the most restrictive types possible when creating columns is necessary to reduce memory consumption.

Microbenchmark: Using Timestamps

We illustrate the difference in aggregation speed using the `creationDate` column of the LDBC Comment table on scale factor 300. This table has approx. 554 million unordered timestamp values. We run a simple aggregation query that returns the average day-of-the month from the timestamps in two configurations.

First, we use a `DATETIME` to encode the values and run the query using the `extract` datetime function:

```
SELECT avg(extract('day' FROM creationDate)) FROM Comment;
```

Second, we use the VARCHAR type and use string operations:

```
SELECT avg(CAST(creationDate[9:10] AS INTEGER)) FROM Comment;
```

The results of the microbenchmark are as follows:

Column type	Storage size	Query time
`DATETIME`	3.3 GB	0.9 s

Column type	Storage size	Query time
VARCHAR	5.2 GB	3.9 s

The results show that using the DATETIME value yields smaller storage sizes and faster processing.

Microbenchmark: Joining on Strings

We illustrate the difference caused by joining on different types by computing a self-join on the LDBC Comment table at scale factor 100. The table has 64-bit integer identifiers used as the `id` attribute of each row. We perform the following join operation:

```
SELECT count(*) AS count
FROM Comment c1
JOIN Comment c2 ON c1.ParentCommentId = c2.id;
```

In the first experiment, we use the correct (most restrictive) types, i.e., both the `id` and the `ParentCommentId` columns are defined as BIGINT. In the second experiment, we define all columns with the VARCHAR type. While the results of the queries are the same for both experiments, their runtimes vary significantly. The results below show that joining on BIGINT columns is approx. 1.8× faster than performing the same join on VARCHAR-typed columns encoding the same value.

Join column payload type	Join column schema type	Example value	Query time
BIGINT	BIGINT	70368755640078	1.2 s
BIGINT	VARCHAR	'70368755640078'	2.1 s

> **Best practice.** Avoid representing numeric values as strings, especially if you intend to perform e.g., join operations on them.

Encoding Repeated Strings as a Dimension Table

When a wide string column has only a few distinct values that repeat across many rows, you can replace it with a small integer key that references a separate *dimension table*. This is the same idea as a star schema, and it keeps the large *fact table* narrow, which speeds up grouping and joining. (For a fixed, known set of values, the ENUM type achieves a similar effect automatically.)

First, build a dimension table that assigns an integer key to each distinct value. Sorting the values keeps the keys stable and improves compression:

```
CREATE OR REPLACE TABLE dim_table AS
    SELECT
        field_name,
        (row_number() OVER (ORDER BY field_name))::INTEGER AS field_pk
    FROM (SELECT DISTINCT field_name FROM fact_table);
```

Then run the aggregation using the integer keys and join the strings back in only at the end, once the result set is small:

```
WITH rollup AS (
    SELECT field_pk, count(*) AS count
    FROM fact_table
    NATURAL JOIN dim_table
    GROUP BY ALL
)
SELECT rollup.* EXCLUDE (field_pk), field_name
FROM rollup
NATURAL JOIN dim_table;
```

> **Best practice.** DuckDB already applies dictionary encoding to string columns in storage, so this technique mainly pays off for query-heavy workloads that repeatedly group or join on a low-cardinality string column.

Constraints

DuckDB allows defining constraints such as `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY`. These constraints can be beneficial for ensuring data integrity but they have a negative effect on load performance as they necessitate building indexes and performing checks. Moreover, they *very rarely improve the performance of queries* as DuckDB does not rely on these indexes for join and aggregation operators (see indexing for more details).

> **Best practice.** Do not define constraints unless your goal is to ensure data integrity.

Microbenchmark: The Effect of Primary Keys

We illustrate the effect of using primary keys with the LDBC Comment table at scale factor 300. This table has approx. 554 million entries. In the first experiment, we create the schema *with* a primary key, then load the data. In the second experiment, we create the schema *without* a primary key, then load the data. In the third experiment, we create the schema *without* a primary key, load the data and then add the primary key constraint. In all cases, we take the data from `.csv.gz` files, and measure the time required to perform the loading.

Operation	Execution time
Load with primary key	461.6 s
Load without primary key	121.0 s
Load without primary key then add primary key	242.0 s

For this dataset, primary keys will only have a (small) positive effect on highly selective queries such as when filtering on a single identifier. Defining primary keys (or indexes) will not have an effect on join and aggregation operators.

> **Best practice.** For best bulk load performance, avoid primary key constraints. If they are required, define them after the bulk loading step.

Indexing

DuckDB has two types of indexes: zonemaps and ART indexes.

Zonemaps

DuckDB automatically creates zonemaps (also known as min-max indexes) for the columns of all general-purpose data types. Operations like predicate pushdown into scan operators and computing aggregations use zonemaps. If a filter criterion (like `WHERE column1 = 123`) is in use, DuckDB can skip any row group whose min-max range does not contain that filter value (e.g., it can omit a block with a min-max range of 1000 to 2000 when comparing for `= 123` or `< 400`).

The Effect of Ordering on Zonemaps

The more ordered the data within a column, the more valuable the zonemap indexes will be. For example, a column could contain a random number on every row in the worst case. Then, DuckDB will likely be unable to skip any row groups. If you query specific columns with selective filters, it is best to pre-order data by those columns when inserting it. Even an imperfect ordering will still be helpful. The best case of ordered data commonly arises with `DATETIME` columns.

Microbenchmark: The Effect of Ordering

For an example, let's repeat the microbenchmark for timestamps with an ordered timestamp column using an ascending order vs. an unordered one.

Column type	Ordered	Storage size	Query time
`DATETIME`	yes	1.3 GB	0.6 s
`DATETIME`	no	3.3 GB	0.9 s

The results show that simply keeping the column order allows for improved compression, yielding a 2.5× smaller storage size. It also allows the computation to be 1.5× faster.

Ordered Integers

Another practical way to exploit ordering is to use the `INTEGER` type with automatic increments rather than `UUID` for columns queried using selective filters. In a scenario where a table contains out-of-order `UUID`s, DuckDB has to scan many row groups to find a specific `UUID` value. An ordered `INTEGER` column allows skipping all row groups except those containing the value.

ART Indexes

DuckDB allows defining Adaptive Radix Tree (ART) indexes in two ways. First, such an index is created implicitly for columns with `PRIMARY KEY`, `FOREIGN KEY`, and `UNIQUE` constraints. Second, explicitly running the `CREATE INDEX` statement creates an ART index on the target column(s).

The tradeoffs of having an ART index on a column are as follows:

1. ART indexes enable constraint checking during changes (inserts, updates, and deletes).
2. Changes on indexed tables perform worse than their non-indexed counterparts. That is because of index maintenance for these operations.
3. For some use cases, *single-column ART indexes* improve the performance of highly selective queries using the indexed column.

An ART index does not affect the performance of join, aggregation, and sorting queries.

ART Index Scans

ART index scans probe a single-column ART index for the requested data instead of scanning a table sequentially. Probing can improve the performance of some queries. DuckDB will try to use an index scan for equality and `IN(...)` conditions. It also pushes dynamic filters, e.g., from hash joins, into the scan, allowing dynamic index scans on these filters.

Indexes are only eligible for index scans if they index a single column without expressions. E.g., the following index is eligible for index scans:

```
CREATE INDEX idx ON tbl (col1);
```

E.g., the following two indexes are **NOT** eligible for index scans:

```
CREATE INDEX idx_multi_column ON tbl (col1, col2);
CREATE INDEX idx_expr ON tbl (col1 + 1);
```

The default threshold for index scans is `MAX(2048, 0.001 * table_cardinality)`. You can configure this threshold via `index_scan_percentage` and `index_scan_max_count`, or disable them by setting these values to zero. When in doubt, use `EXPLAIN ANALYZE` to verify that your query plan uses the index scan.

Indexes and Memory

DuckDB registers index memory through its buffer manager. However, these index buffers are not yet buffer-managed. That means DuckDB does not yet destroy any index buffers if it has to evict memory. Thus, indexes can take up a significant portion of DuckDB's available memory, potentially affecting the performance of memory-intensive queries. Re-attaching (DETACH + ATTACH) the database containing indexes can mitigate this effect, as we deserialize index memory lazily. Disabling index scans and re-attaching after changes can further decrease the impact of indexes on DuckDB's available memory.

Indexes and Opening Databases

Indexes are serialized to disk and deserialized lazily, i.e., when reopening the database. Operations using the index will only load the required parts of the index. Therefore, having an index will not cause any slowdowns when opening an existing database.

> **Best practice.** We recommend following these guidelines:
>
> - Only use primary keys, foreign keys, or unique constraints, if these are necessary for enforcing constraints on your data.
> - Do not define explicit indexes unless you have highly selective queries and enough memory available.
> - If you define an ART index, do so after bulk loading the data to the table. Adding an index prior to loading, either explicitly or via primary/foreign keys, is detrimental to load performance.

Join Operations

How to Force a Join Order

DuckDB has a cost-based query optimizer, which uses statistics in the base tables (stored in a DuckDB database or Parquet files) to estimate the cardinality of operations.

Turn off the Join Order Optimizer

To turn off the join order optimizer, set the following PRAGMAs:

```
SET disabled_optimizers = 'join_order,build_side_probe_side';
```

This disables both the join order optimizer and left/right swapping for joins. This way, DuckDB builds a left-deep join tree following the order of `JOIN` clauses.

```
SELECT ...
FROM ...
JOIN ...  -- this join is performed first
JOIN ...; -- this join is performed second
```

Once the query in question has been executed, turn back the optimizers with the following command:

```
SET disabled_optimizers = '';
```

Create Temporary Tables

To force a particular join order, you can break up the query into multiple queries, with each creating a temporary table:

```
CREATE OR REPLACE TEMPORARY TABLE t1 AS
    ...;

-- join on the result of the first query, t1
CREATE OR REPLACE TEMPORARY TABLE t2 AS
    SELECT * FROM t1 ...;

-- compute the final result using t2
SELECT * FROM t1 ...
```

To clean up, drop the interim tables:

```
DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;
```

Configuring the Join Order Optimizer

DuckDB's join order optimizer computes the optimal join order exactly using a dynamic programming algorithm. As the number of tables in a join grows, exact enumeration becomes expensive, so for large joins the optimizer switches to an approximate (greedy) algorithm. The `approximate_join_order_threshold` setting controls when this happens: joins with at least this many tables are ordered approximately, while joins with fewer tables are ordered exactly. The default value is 12.

For example, to use the exact algorithm for joins of up to 15 tables, run:

```
SET approximate_join_order_threshold = 16;
```

Raising the threshold can lead to better join orders for queries with many joins at the cost of longer optimization times. Lowering it reduces optimization time for large joins.

> Even below the threshold, the optimizer falls back to the approximate algorithm if exact enumeration exceeds its internal budget.

File Formats

Handling Parquet Files

DuckDB has advanced support for Parquet files, which includes directly querying Parquet files. When deciding on whether to query these files directly or to first load them to the database, you need to consider several factors.

Reasons for Querying Parquet Files

Availability of basic statistics: Parquet files use a columnar storage format and contain basic statistics such as zonemaps. Thanks to these features, DuckDB can leverage optimizations such as projection and filter pushdown on Parquet files. Therefore, workloads that combine projection, filtering, and aggregation tend to perform quite well when run on Parquet files.

Storage considerations: Loading the data from Parquet files will require approximately the same amount of space for the DuckDB database file. Therefore, if the available disk space is constrained, it is worth running the queries directly on Parquet files.

Reasons against Querying Parquet Files

Lack of advanced statistics: The DuckDB database format has the hyperloglog statistics that Parquet files do not have. These improve the accuracy of cardinality estimates, and are especially important if the queries contain a large number of join operators.

Tip. If you find that DuckDB produces a suboptimal join order on Parquet files, try loading the Parquet files to DuckDB tables. The improved statistics likely help obtain a better join order.

Repeated queries: If you plan to run multiple queries on the same dataset, it is worth loading the data into DuckDB. The queries will always be somewhat faster, which over time amortizes the initial load time.

High decompression times: Some Parquet files are compressed using heavyweight compression algorithms such as gzip. In these cases, querying the Parquet files will necessitate an expensive decompression time every time the file is accessed. Meanwhile, lightweight compression methods like Snappy, LZ4, and zstd, are faster to decompress. You may use the `parquet_metadata` function to find out the compression algorithm used.

Microbenchmark: Running TPC-H on a DuckDB Database vs. Parquet The queries on the TPC-H benchmark run approximately 1.1-5.0× slower on Parquet files than on a DuckDB database.

> **Best practice.** If you have the storage space available, and have a join-heavy workload and/or plan to run many queries on the same dataset, load the Parquet files into the database first. The compression algorithm and the row group sizes in the Parquet files have a large effect on performance: study these using the `parquet_metadata` function.

The Effect of Row Group Sizes

DuckDB works best on Parquet files with row groups of 100K-1M rows each. The reason for this is that DuckDB can only parallelize over row groups – so if a Parquet file has a single giant row group it can only be processed by a single thread. You can use the `parquet_metadata` function to figure out how many row groups a Parquet file has. When writing Parquet files, use the `row_group_size` option.

Microbenchmark: Running Aggregation Query at Different Row Group Sizes We run a simple aggregation query over Parquet files using different row group sizes, selected between 960 and 1,966,080. The results are as follows.

Row group size	Execution time
960	8.77 s
1920	8.95 s
3840	4.33 s
7680	2.35 s
15360	1.58 s
30720	1.17 s
61440	0.94 s
122880	0.87 s
245760	0.93 s
491520	0.95 s
983040	0.97 s
1966080	0.88 s

The results show that row group sizes <5,000 have a strongly detrimental effect, making runtimes more than 5-10× larger than ideally-sized row groups, while row group sizes between 5,000 and 20,000 are still 1.5-2.5× off from best performance. Above row group size of 100,000, the differences are small: the gap is about 10% between the best and the worst runtime.

Parquet File Sizes

DuckDB can also parallelize across multiple Parquet files. It is advisable to have at least as many total row groups across all files as there are CPU threads. For example, with a machine having 10 threads, both 10 files with 1 row group or 1 file with 10 row groups will achieve full parallelism. It is also beneficial to keep the size of individual Parquet files moderate.

> **Best practice.** The ideal range is between 100 MB and 10 GB per individual Parquet file.

Hive Partitioning for Filter Pushdown

When querying many files with filter conditions, performance can be improved by using a Hive-format folder structure to partition the data along the columns used in the filter condition. DuckDB will only need to read the folders and files that meet the filter criteria. This can be especially helpful when querying remote files.

More Tips on Reading and Writing Parquet Files

For tips on reading and writing Parquet files, see the Parquet Tips page.

Loading CSV Files

CSV files are often distributed in compressed format such as GZIP archives (`.csv.gz`). DuckDB can decompress these files on the fly. In fact, this is typically faster than decompressing the files first and loading them due to reduced IO.

Schema	Load time
Load from GZIP-compressed CSV files (`.csv.gz`)	107.1 s
Decompressing (using parallel `gunzip`) and loading from decompressed CSV files	121.3 s

Loading Many Small CSV Files

The CSV reader runs the CSV sniffer on all files. For many small files, this may cause an unnecessarily high overhead. A potential optimization to speed this up is to turn the sniffer off. Assuming that all files have the same CSV dialect and column names/types, get the sniffer options as follows:

```
.mode line
SELECT Prompt FROM sniff_csv('part-0001.csv');
```

```
Prompt = FROM read_csv('file_path.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n',
skip=0, header=true, columns={'hello': 'BIGINT', 'world': 'VARCHAR'});
```

Then, you can adjust `read_csv` command, by e.g., applying filename expansion (globbing), and run with the rest of the options detected by the sniffer:

```
FROM read_csv('part-*.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n', skip=0,
header=true, columns={'hello': 'BIGINT', 'world': 'VARCHAR'});
```

Reducing Storage Size with DuckLake

Storing data in DuckLake can significantly reduce storage size compared to CSV files or even DuckDB's native format, because DuckLake writes the data as Parquet files whose compression settings you control. Before copying data into a DuckLake catalog, set the Parquet compression options on the attached catalog:

```
ATTACH 'ducklake:metadata.ducklake' AS d;
CALL d.set_option('parquet_compression', 'zstd');
CALL d.set_option('parquet_version', 2);
CALL d.set_option('parquet_compression_level', 22);
COPY FROM DATABASE services TO d;
```

Using a heavyweight `zstd` compression level trades a longer write time for smaller files, so it is most beneficial for data that is written once and read many times.

Tuning Workloads

DuckDB aims to perform well out of the box, so most workloads do not require any tuning. When tuning is worthwhile, there is no single configuration or magic number that is optimal for every workload: the right settings depend on what you are optimizing for and on the characteristics of your data, hardware, and environment. Rather than copying a fixed value, identify which of the concerns below match your workload, apply the corresponding change, and measure its effect.

If your priority is…	Consider	Covered in
Loading or exporting data larger than memory	Disabling `preserve_insertion_order`	The `preserve_insertion_order` option
Using all available cores	Row group sizing and the `threads` setting	Parallelism
Processing data larger than memory	Configuring the spill-to-disk directory	Larger-than-memory workloads
Diagnosing a slow query	Inspecting the query plan	Profiling
Running the same query many times with different parameters	Prepared statements	Prepared statements
Reading files over a slow or unreliable network	Tuning asynchronous I/O and minimizing IO	Querying remote files
Running many small queries	Reusing a single connection	Best practices for using connections

The rest of this page covers each of these areas in detail. See the configuration reference for the full list of available settings.

The `preserve_insertion_order` Option

When importing or exporting datasets (from/to the Parquet or CSV formats), which are much larger than the available memory, an out of memory error may occur:

```
Out of Memory Error: failed to allocate data of size ... (.../... used)
```

In these cases, consider setting the `preserve_insertion_order` configuration option to `false`:

```
SET preserve_insertion_order = false;
```

This allows the system to re-order any results that do not contain `ORDER BY` clauses, potentially reducing memory usage. For details on which operations preserve row order and which do not, see the Order Preservation page.

Parallelism (Multi-Core Processing)

The Effect of Row Groups on Parallelism

DuckDB parallelizes the workload based on *row groups,* i.e., groups of rows that are stored together at the storage level. The default row group size in DuckDB's database format is 122,880 rows. Parallelism starts at the level of row groups, therefore, for a query to run on *k* threads, it needs to scan at least *k* * 122,880 rows.

The row group size can be specified as an option of the ATTACH statement:

```
ATTACH '/tmp/somefile.db' AS db (ROW_GROUP_SIZE 16384);
```

The performance considerations when choosing `ROW_GROUP_SIZE` for Parquet files apply verbatim to DuckDB's own database format.

Too Many Threads

Note that in certain cases DuckDB may launch *too many threads* (e.g., due to HyperThreading), which can lead to slowdowns. In these cases, it's worth manually limiting the number of threads using `SET threads = X{:.language-sql .highlight}`.

Larger-than-Memory Workloads (Out-of-Core Processing)

A key strength of DuckDB is support for larger-than-memory workloads, i.e., it is able to process datasets that are larger than the available system memory (also known as *out-of-core processing*). It can also run queries where the intermediate results cannot fit into memory. This section explains the prerequisites, scope, and known limitations of larger-than-memory processing in DuckDB.

Spilling to Disk

Larger-than-memory workloads are supported by spilling to disk. With the default configuration, DuckDB creates the `<database_file_name>.tmp`{:.language-sql .highlight} temporary directory (in persistent mode) or the `.tmp`{:.language-sql .highlight} directory (in in-memory mode). This directory can be changed using the `temp_directory` configuration option, e.g.:

```
SET temp_directory = '/path/to/temp_dir.tmp/';
```

Blocking Operators

Some operators cannot output a single row until the last row of their input has been seen. These are called *blocking operators* as they require their entire input to be buffered, and are the most memory-intensive operators in relational database systems. The main blocking operators are the following:

- *grouping:* `GROUP BY`
- *joining:* `JOIN`
- *sorting:* `ORDER BY`
- *windowing:* `OVER ... (PARTITION BY ... ORDER BY ...)`

DuckDB supports larger-than-memory processing for all of these operators.

Limitations

DuckDB strives to always complete workloads even if they are larger-than-memory. That said, there are some limitations at the moment:

- If multiple blocking operators appear in the same query, DuckDB may still throw an out-of-memory exception due to the complex interplay of these operators.
- Some aggregate functions, such as `string_agg()`, do not support offloading to disk.
- Aggregate functions that use sorting are holistic, i.e., they need all inputs before the aggregation can start. As DuckDB cannot yet offload some complex intermediate aggregate states to disk, these functions can cause an out-of-memory exception when run on large datasets.

Profiling

If your queries are not performing as well as expected, it's worth studying their query plans:

- Use `EXPLAIN` to print the physical query plan without running the query.
- Use `EXPLAIN ANALYZE` to run and profile the query. This will show the CPU time that each step in the query takes. Note that due to multi-threading, adding up the individual times will be larger than the total query processing time.

Query plans can point to the root of performance issues. A few general directions:

- Avoid nested loop joins in favor of hash joins.
- A scan that does not include a filter pushdown for a filter condition that is later applied performs unnecessary IO. Try rewriting the query to apply a pushdown.
- Bad join orders where the cardinality of an operator explodes to billions of tuples should be avoided at all costs.

Prepared Statements

Prepared statements can improve performance when running the same query many times, but with different parameters. When a statement is prepared, it completes several of the initial portions of the query execution process (parsing, planning, etc.) and caches their output. When it is executed, those steps can be skipped, improving performance. This is beneficial mostly for repeatedly running small queries (with a runtime of < 100ms) with different sets of parameters.

Note that it is not a primary design goal for DuckDB to quickly execute many small queries concurrently. Rather, it is optimized for running larger, less frequent queries.

Querying Remote Files

Starting with version 2.0, DuckDB uses asynchronous I/O when reading Parquet files, uncompressed, seekable UTF-8 CSV files and DuckDB database files (e.g., attached over HTTPS or S3), as well as when Parquet and CSV files are read through data lake formats such as DuckLake. For these formats, DuckDB keeps multiple requests in flight automatically, so there is no need to raise the number of threads. If the defaults do not saturate your network bandwidth, tune the `async_threads` and `read_ahead_depth` settings instead.

Other formats, such as JSON files and compressed CSV files, still use synchronous I/O: each DuckDB thread can make at most one HTTP request at a time. If a query on such files must make many small requests over the network, increasing DuckDB's `threads` setting to larger than the total number of CPU cores (approx. 2-5 times CPU cores) can improve parallelism and performance.

Avoid Reading Unnecessary Data

The main bottleneck in workloads reading remote files is likely to be the IO. This means that minimizing the unnecessarily read data can be highly beneficial.

Some basic SQL tricks can help with this:

- Avoid `SELECT *`. Instead, only select columns that are actually used. DuckDB will try to only download the data it actually needs.
- Apply filters on remote Parquet files when possible. DuckDB can use these filters to reduce the amount of data that is scanned.
- Either sort or partition data by columns that are regularly used for filters: this increases the effectiveness of the filters in reducing IO.

To inspect how much remote data is transferred for a query, `EXPLAIN ANALYZE` can be used to print out the total number of requests and total data transferred for queries on remote files.

Caching

Starting with version 1.3.0, DuckDB supports caching remote data. To inspect the content of the external file cache, run:

```
FROM duckdb_external_file_cache();
```

The external file cache follows the `Cache-Control` headers of HTTP responses: data from a response marked `no-store`, `no-cache` or `private`, or from a response that is already stale (e.g., `max-age=0`), is not reused. Object stores often send such headers on authenticated reads. For example, Google Cloud Storage returns `Cache-Control: private, max-age=0` for every read of a non-public object, so by default, these files are read from the object store again on every query.

To reuse cached data regardless of these headers, set `enable_external_file_cache` explicitly, even though it is enabled by default:

```
SET enable_external_file_cache = true;
```

Cached data is still validated against the remote file as configured by the `validate_external_file_cache` setting.

> **Warning.** The external file cache is shared by all connections of a database instance. Only set this option if every connection may read the cached files, e.g., because all connections use the same credentials.

Best Practices for Using Connections

DuckDB will perform best when reusing the same database connection many times. Disconnecting and reconnecting on every query will incur some overhead, which can reduce performance when running many small queries. DuckDB also caches some data and metadata in memory, and that cache is lost when the last open connection is closed. Frequently, a single connection will work best, but a connection pool may also be used.

Using multiple connections can parallelize some operations, although it is typically not necessary. DuckDB does attempt to parallelize as much as possible within each individual query, but it is not possible to parallelize in all cases. Making multiple connections can process more operations concurrently. This can be more helpful if DuckDB is not CPU limited, but instead bottlenecked by another resource like network transfer speed.

Persistent vs. In-Memory Tables

DuckDB supports lightweight compression techniques. By default, compression is only applied on persistent (on-disk) databases and not on in-memory tables.

In some cases, this can result in counter-intuitive performance results where queries are faster on on-disk tables compared to in-memory ones. Let's take Q1 of the TPC-H workload for example on the SF30 dataset:

```
CALL dbgen(sf = 30);
.timer on
PRAGMA tpch(1);
```

We run this script using three DuckDB prompts:

Database setup	DuckDB prompt	Execution time
In-memory DB (uncompressed)	`duckdb`	4.22 s
In-memory DB (compressed)	`duckdb -cmd "ATTACH ':memory:' AS db (COMPRESS); USE db;"`	0.55 s
Persistent DB (compressed)	`duckdb tpch-sf30.db`	0.56 s

We can observe that the compressed databases are about 8× faster compared to the uncompressed in-memory database.

My Workload Is Slow

If you find that your workload in DuckDB is slow, we recommend performing the following checks. More detailed instructions are linked for each point.

1. Do you have enough memory? DuckDB works best if you have 1-4 GB memory per thread.
2. Is your system maybe overcommitting memory, forcing the operating system to swap? Try *lowering* the amount of memory available from the default 80% of the total RAM using `SET memory_limit = '...';`{:.language-sql .highlight}. While this sounds counter-intuitive, it can sometimes improve query performance, especially in memory-constrained environments where other processes are likely using more than 20% of the total system memory.
3. Are you using a fast disk? Network-attached disks (such as cloud block storage) cause write-intensive and larger than memory workloads to slow down. For running such workloads in cloud environments, it is recommended to use instance-attached storage (NVMe SSDs).
4. Are you using indexes or constraints (primary key, unique, etc.)? If possible, try disabling them, which boosts load and update performance.
5. Are you using the correct types? For example, use `TIMESTAMP` to encode datetime values.
6. Are you reading from Parquet files? If so, do they have row group sizes between 100k and 1M and file sizes between 100 MB to 10 GB?
7. Does the query plan look right? Study it with `EXPLAIN`.
8. Is the workload running in parallel? Use `htop` or the operating system's task manager to observe this.
9. Is DuckDB using too many threads? Try limiting the amount of threads.

Are you aware of other common issues? If so, please click the *Report content issue* link below and describe them along with their workarounds.

Out-of-Memory Issues

Configuration Options

If you are experiencing out-of-memory issues with DuckDB, try tweaking the following configuration options:

- Reduce the number of threads using the `SET threads = ...`{:.language-sql .highlight} command.
- Setting the memory limit lower than the default 80% can help prevent out of memory errors. While this configuration sounds counter-intuitive, it helps because some of DuckDB's operations circumvent the database's buffer manager and thus they can reserve more memory than allowed by the memory limit. If this happens (e.g., DuckDB is killed by the operating system or an OOM reaper process), set the memory limit to just 50-60% of the total system memory by using the `SET memory_limit = '...'`{:.language-sql .highlight} statement.
- If your query reads a large amount of data from a file or writes a large amount of data, try setting the `preserve_insertion_order` option to `false`: `SET preserve_insertion_order = false`{:.language-sql .highlight}.

In short, try the settings:

```
SET threads = <lower_than_the_number_of_available_threads>;
SET memory_limit = <lower_than_80%_of_system_memory>;
SET preserve_insertion_order = false;
```

Schema and Indexes

If you run in a memory-constrained environment, using smaller data types (e.g., `TINYINT`) can reduce the amount of memory and disk space required to complete a query. DuckDB's bitpacking compression means small values stored in larger data types will not take up larger sizes on disk, but they will take up more memory during processing – hence, using the most restrictive types possible when creating columns is necessary to reduce memory consumption.

Indexes can significantly increase memory consumption. Additionally, they are currently not buffer-managed. Avoid using indexes if possible. If your workload requires indexing large tables, consider reducing the memory limit to accommodate this.

Benchmarks

For several of the recommendations in our performance guide, we use microbenchmarks to back up our claims. For these benchmarks, we use datasets from the TPC-H benchmark and the LDBC Social Network Benchmark's BI workload.

Datasets

We use the LDBC BI SF300 dataset's Comment table (20 GB `.tar.zst` archive, 21 GB when decompressed into `.csv.gz` files), while others use the same table's `creationDate` column (4 GB `.parquet` file).

The TPC datasets used in the benchmark are generated with the DuckDB tpch extension.

A Note on Benchmarks

Running fair benchmarks is difficult, especially when performing system-to-system comparison. When running benchmarks on DuckDB, please make sure you are using the latest version (preferably the preview build). If in doubt about your benchmark results, feel free to contact us at `gabor@duckdb.org`.

Disclaimer on Benchmarks

Note that the benchmark results presented in this guide do not constitute official TPC or LDBC benchmark results. Instead, they merely use the datasets of and some queries provided by the TPC-H and the LDBC BI benchmark frameworks, and omit other parts of the workloads such as updates.

Working with Huge Databases

This page contains information for working with huge DuckDB database files. While most DuckDB databases are well below 1 TB, in our 2024 user survey, 1% of respondents used DuckDB files of 2 TB or more (corresponding to roughly 10 TB of CSV files).

DuckDB's native database format supports huge database files without any practical restrictions, however, there are a few things to keep in mind when working with huge database files.

1. Object storage systems have lower limits on file sizes than block-based storage systems. For example, AWS S3 limits the file size to 5 TB.
2. Checkpointing a DuckDB database can be slow. For example, checkpointing after adding a few rows to a table in the TPC-H SF1000 database takes approximately 5 seconds.
3. On block-based storage, the file system has a significant effect on performance when working with large files. On Linux, DuckDB performs best with XFS on large files.

For storing large amounts of data, consider using the DuckLake lakehouse format. Because DuckLake spreads the data across many immutable Parquet files instead of a single database file, it sidesteps the object-storage file-size limits and the slow whole-database checkpoints described above, while still providing transactional guarantees.

Python

Installing the Python Client

Installing via Pip

The Python client requires Python 3.11 or later. On Python 3.10, `pip` does not report an error: it installs a 1.x release instead, and with `--pre` a 1.x development build.

The latest release of the Python client can be installed using `pip`.

```
pip install duckdb
```

The pre-release Python client (known as the "preview" or "nightly" build) can be installed using `--pre`.

```
pip install duckdb --upgrade --pre
```

Installing from Source

Follow the Python build guide.

Executing SQL in Python

SQL queries can be executed using the `duckdb.sql` function.

```
import duckdb

duckdb.sql("SELECT 42").show()
```

By default this will create a relation object. The result can be converted to various formats using the result conversion functions. For example, the `fetchall` method can be used to convert the result to Python objects.

```
results = duckdb.sql("SELECT 42").fetchall()
print(results)
```

```
[(42,)]
```

Several other result objects exist. For example, you can use `df` to convert the result to a Pandas DataFrame.

```
results = duckdb.sql("SELECT 42").df()
print(results)
```

```
   42
0  42
```

By default, a global in-memory connection will be used. Any data stored in it will be lost after shutting down the program. A connection to a persistent database can be created using the `connect` function.

After connecting, SQL queries can be executed using the `sql` command.

```
con = duckdb.connect("file.db")
con.sql("CREATE TABLE integers (i INTEGER)")
con.sql("INSERT INTO integers VALUES (42)")
con.sql("SELECT * FROM integers").show()
```

A statement that changes data or the schema, such as `CREATE TABLE` or `INSERT`, runs when `sql` is called, and the relation it returns holds the statement's result. A `SELECT` query that only reads runs each time its relation is read, for example by `show()` or `fetchall()`.

Query Parameters

To parameterize a query, pass the values separately from the SQL string using the `execute` method. Use `?` for positional parameters, supplying the values as a list:

```
con.execute("SELECT * FROM integers WHERE i = ?", [42]).fetchall()
```

Named parameters use the `$name` syntax, with the values supplied as a dictionary:

```
con.execute("SELECT * FROM integers WHERE i = $value", {"value": 42}).fetchall()
```

`con.sql` and `duckdb.sql` take the values through their `params` keyword argument:

```
con.sql(
    "SELECT * FROM integers WHERE i = $value",
    params={"value": 42},
).fetchall()
```

The `params` list or dictionary is copied when `sql` is called, together with any list, tuple or dictionary inside it. Other values are converted each time the relation is read, so a `bytearray` changed in place afterwards is read changed.

To run the same parameterized statement for many sets of values, for example when inserting multiple rows, use `executemany`:

```
con.executemany("INSERT INTO integers VALUES (?)", [[1], [2], [3]])
```

Afterwards, `con.rowcount` holds the number of inserted rows, here 3.

Jupyter Notebooks

DuckDB's Python client can be used directly in Jupyter notebooks with no additional configuration, as the first section shows. Additional libraries can simplify SQL query development; the rest of this guide uses the JupySQL package. See other guides in the Python section for how to use DuckDB and Python together.

Querying in a Python Cell

In a plain Python cell, `duckdb.sql` runs a query and finds a pandas DataFrame by its variable name:

```
import duckdb
import pandas as pd

sales = pd.DataFrame({"item": ["jeans", "hammer"], "price": [20, 42]})
duckdb.sql("SELECT * FROM sales WHERE price > 30")
```

The last expression displays a table in the notebook. Add `.df()` to keep the result as a pandas DataFrame.

Library Installation

This JupySQL workflow is also available as a Google Colab notebook.

Four additional libraries improve the DuckDB experience in Jupyter notebooks.

1. jupysql: Convert a Jupyter code cell into a SQL cell
2. Pandas: Clean table visualizations and compatibility with other analysis
3. matplotlib: Plotting with Python
4. duckdb-engine (DuckDB SQLAlchemy driver): Used by SQLAlchemy to connect to DuckDB (optional)

Run these `pip install` commands from the command line if Jupyter Notebook is not yet installed. Otherwise, see Google Colab link above for an in-notebook example:

```
pip install duckdb
```

Install Jupyter Notebook:

```
pip install notebook
```

Or JupyterLab:

```
pip install jupyterlab
```

Install supporting libraries:

```
pip install jupysql pandas matplotlib duckdb-engine
```

Library Import and Configuration

Open a Jupyter Notebook and import the relevant libraries.

Set configurations on jupysql to directly output data to Pandas and to simplify the output that is printed to the notebook.

```
%config SqlMagic.autopandas = True
%config SqlMagic.feedback = 0
%config SqlMagic.displaycon = False
```

Connecting to DuckDB Natively

To connect to DuckDB, run:

```
import duckdb
import pandas as pd

%load_ext sql
conn = duckdb.connect(config={"python_scan_all_frames": True})
%sql conn --alias duckdb
```

> **Warning.** With JupySQL 0.11.1, `%sqlplot boxplot` fails on a native connection of DuckDB's Python client 2.0 with `Aggregate Function with name percentile_disc does not exist!`. To plot a boxplot, connect via SQLAlchemy.

> **Warning.** Variables are not recognized within a native DuckDB connection.

Connecting to DuckDB via SQLAlchemy

Alternatively, you can connect to DuckDB via SQLAlchemy using `duckdb_engine`. See the performance and feature differences.

```
import duckdb
import pandas as pd
# No need to import duckdb_engine
#  jupysql will auto-detect the driver needed based on the connection string!

# Import jupysql Jupyter extension to create SQL cells
%load_ext sql
```

Either connect to a new in-memory DuckDB or a file-backed database:

```
%sql duckdb:///:memory:
```

```
%sql duckdb:///path/to/file.db
```

> The SQLAlchemy connection cannot share the default connection of `duckdb.sql`: `duckdb:///:default:` fails with `Default connection fetching is only allowed without additional options.`

A failed statement aborts the open transaction of the SQLAlchemy connection. Later statements then fail with `Current transaction is aborted (please ROLLBACK)` until you run `%sql ROLLBACK`.

Querying DuckDB

Single line SQL queries can be run using `%sql` at the start of a line. Query results will be displayed as a Pandas DataFrame.

```
%sql SELECT 'Off and flying!' AS a_duckdb_column;
```

An entire Jupyter cell can be used as a SQL cell by placing `%%sql` at the start of the cell. Query results will be displayed as a Pandas DataFrame.

```
%%sql
SELECT
    schema_name,
    function_name
FROM duckdb_functions()
ORDER BY ALL DESC
LIMIT 5;
```

To store the query results in a Python variable, use `<<` as an assignment operator. This can be used with both the `%sql` and `%%sql` Jupyter magics.

```
%sql res << SELECT 'Off and flying!' AS a_duckdb_column;
```

If the `%config SqlMagic.autopandas = True` option is set, the variable is a Pandas dataframe, otherwise, it is a `ResultSet` that can be converted to Pandas with the `DataFrame()` function.

Querying Pandas Dataframes

DuckDB is able to find and query any dataframe stored as a variable in the Jupyter notebook.

```
input_df = pd.DataFrame.from_dict({"i": [1, 2, 3],
                                   "j": ["one", "two", "three"]})
```

The dataframe being queried can be specified just like any other table in the FROM clause.

> **Warning.** The query finds `input_df` only with the `python_scan_all_frames` option. Set it in `duckdb.connect` for a native connection, as above. With the SQLAlchemy connection, run `%sql SET python_scan_all_frames=true` first.

```
%sql output_df << SELECT sum(i) AS total_i FROM input_df;
```

Visualizing DuckDB Data

The most common way to plot datasets in Python is to load them using Pandas and then use matplotlib or seaborn for plotting. This approach requires loading all data into memory which is highly inefficient. The plotting module in JupySQL runs computations in the SQL engine. This delegates memory management to the engine and ensures that intermediate computations do not keep eating up memory, efficiently plotting massive datasets.

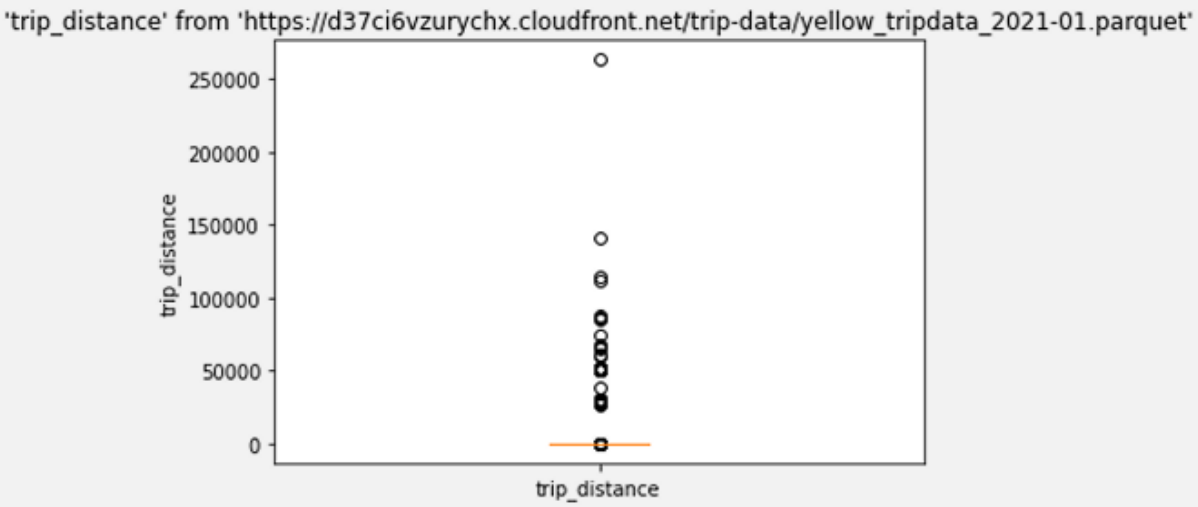
Boxplot & Histogram

To create a boxplot, call `%sqlplot boxplot`, passing the name of the table and the column to plot. In this case, the name of the table is the path of the locally stored Parquet file.

```
from urllib.request import urlretrieve

_ = urlretrieve(
    "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2021-01.parquet",
    "yellow_tripdata_2021-01.parquet",
)
```

```
%sqlplot boxplot --table yellow_tripdata_2021-01.parquet --column trip_distance
```



Install and Load DuckDB httpfs Extension

DuckDB's httpfs extension allows Parquet and CSV files to be queried remotely over http. These examples query a Parquet file that contains historical taxi data from NYC. Using the Parquet format allows DuckDB to only pull the rows and columns into memory that are needed rather than downloading the entire file. DuckDB can be used to process local Parquet files as well, which may be desirable if querying the entire Parquet file, or running multiple queries that require large subsets of the file.

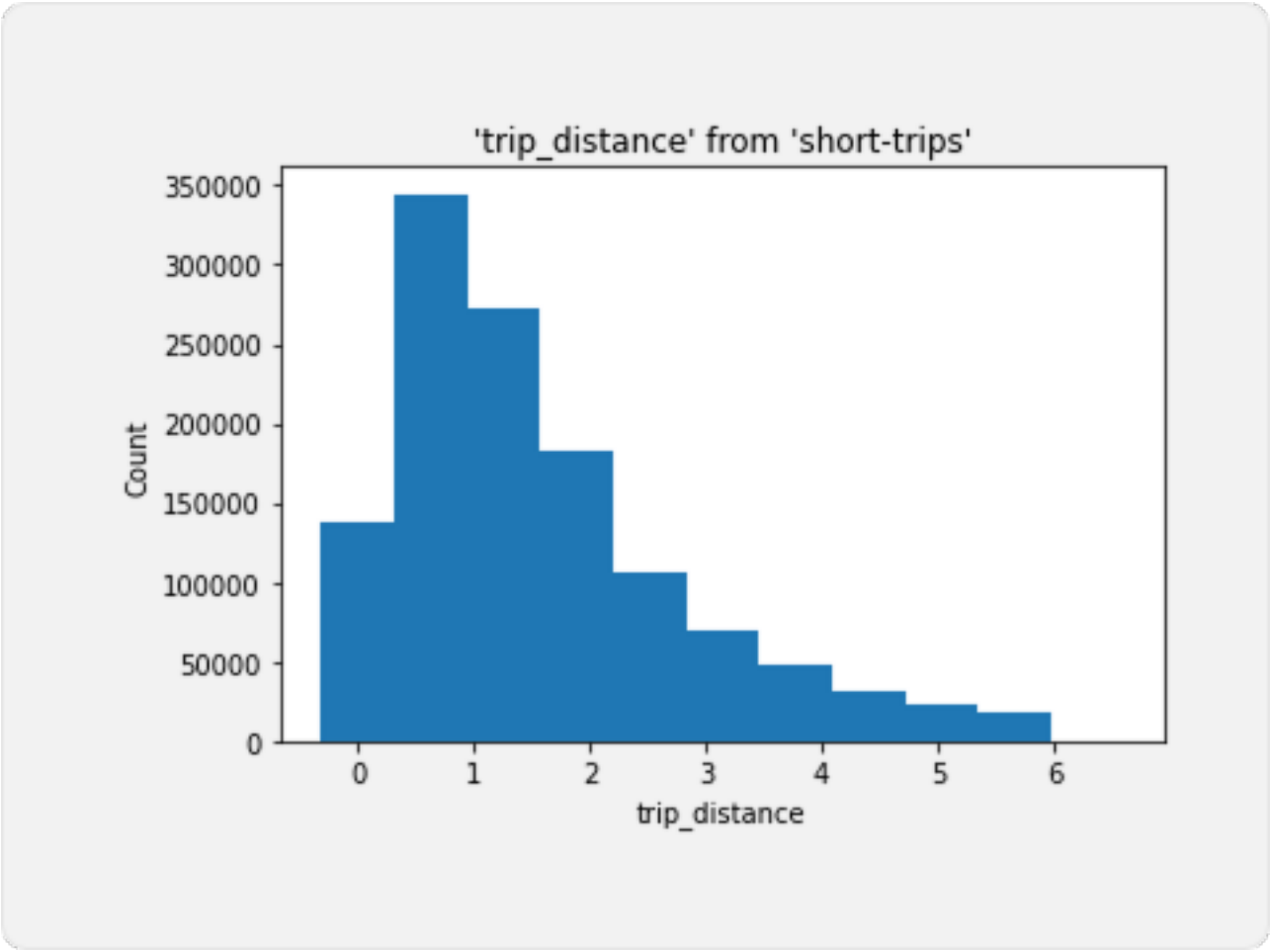
```
%%sql
INSTALL httpfs;
LOAD httpfs;
```

Now, create a query that filters by the 90th percentile. Note the use of the `--save` and `--no-execute` options. This tells JupySQL to store the query, but skips execution. It will be referenced in the next plotting call.

```
%%sql --save short_trips --no-execute
SELECT *
FROM 'https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2021-01.parquet'
WHERE trip_distance < 6.3
```

To create a histogram, call `%sqlplot histogram` and pass the name of the table, the column to plot and the number of bins. This uses `--with short_trips` so JupySQL uses the query defined previously and therefore only plots a subset of the data.

```
%sqlplot histogram --table short_trips --column trip_distance --bins 10 --with short_trips
```



Summary

You now have the ability to alternate between SQL and Pandas in a simple and highly performant way! You can plot massive datasets directly through the engine (avoiding both the download of the entire file and loading all of it into Pandas in memory). Dataframes can be read as tables in SQL, and SQL results can be output into Dataframes. Happy analyzing!

An alternative to `jupysql` is `magic_duckdb`.

marimo Notebooks

marimo is an open-source reactive notebook for Python and SQL that's tightly integrated with DuckDB's Python client, letting you mix and match Python and SQL in a single git-versionable notebook. Unlike traditional notebooks, when you run a cell or interact with a UI element, marimo automatically (or lazily) runs affected cells, keeping code and outputs consistent. Its integration with DuckDB makes it well-suited to interactively working with data, and its representation as a Python file makes it simple to run notebooks as scripts.

Installation

To get started, install marimo from your terminal. Its `sql` extra also installs DuckDB, Polars and PyArrow:

```
pip install "marimo[sql]"
# or
uv add "marimo[sql]"
```

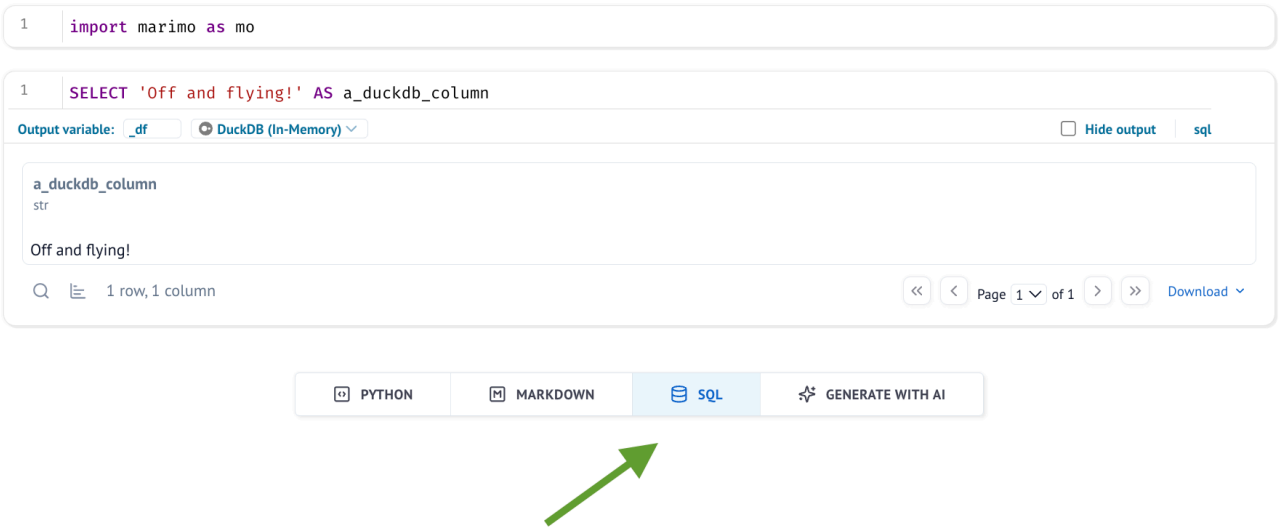
Run a tutorial:

```
marimo tutorial sql
```

SQL in marimo

Create a notebook from your terminal with `marimo edit notebook.py`. Create SQL cells in one of three ways:

1. Right-click the **+** button and pick **SQL cell**
2. Convert any empty cell to SQL via the cell menu
3. Hit the SQL button at the bottom of your notebook



In marimo, SQL cells give the appearance of writing SQL while being serialized as standard Python code using the `mo.sql()` function, which keeps your notebook as pure Python code without requiring special syntax or magic commands.

```
df = mo.sql(f"SELECT 'Off and flying!' AS a_duckdb_column")
```

This is because marimo stores notebooks as pure Python, for many reasons, such as git-friendly diffs and running notebooks as Python scripts.

The SQL statement itself is an f-string, letting you interpolate Python values into the query with `{}` (shown later). In particular, this means your SQL queries can depend on the values of UI elements or other Python values, all part of marimo's dataflow graph.

> **Warning.** Heads up! If you have user-generated content going into the SQL queries, be sure to sanitize your inputs to prevent SQL injection.

Connecting a Custom DuckDB Connection

To connect to a custom DuckDB connection instead of using the default global connection, create a cell and create a DuckDB connection as a Python variable:

```
import duckdb

# Create a DuckDB connection
conn = duckdb.connect("path/to/my/duckdb.db")
```

marimo automatically discovers the connection and lets you select it in the SQL cell's connection dropdown.

Database, Schema and Table Auto-Discovery

marimo introspects connections and displays the database, schemas, tables and columns in the Data Sources panel. This panel lets you quickly navigate your schemas to pull tables and columns into your SQL queries.

Reference a Local Dataframe

Reference a local dataframe in your SQL cell by using the name of the Python variable that holds the dataframe. If you have a database connection with a table of the same name, the database table will be used instead.

```
import polars as pl
df = pl.DataFrame({"value": [1, 2, 3]})
```

```
SELECT * FROM df WHERE value > 2
```

Reference the Output of a SQL Cell

Defining a non-private (non-underscored) output variable in the SQL cell allows you to reference the resulting dataframe in other Python and SQL cells.

Reactive SQL Cells

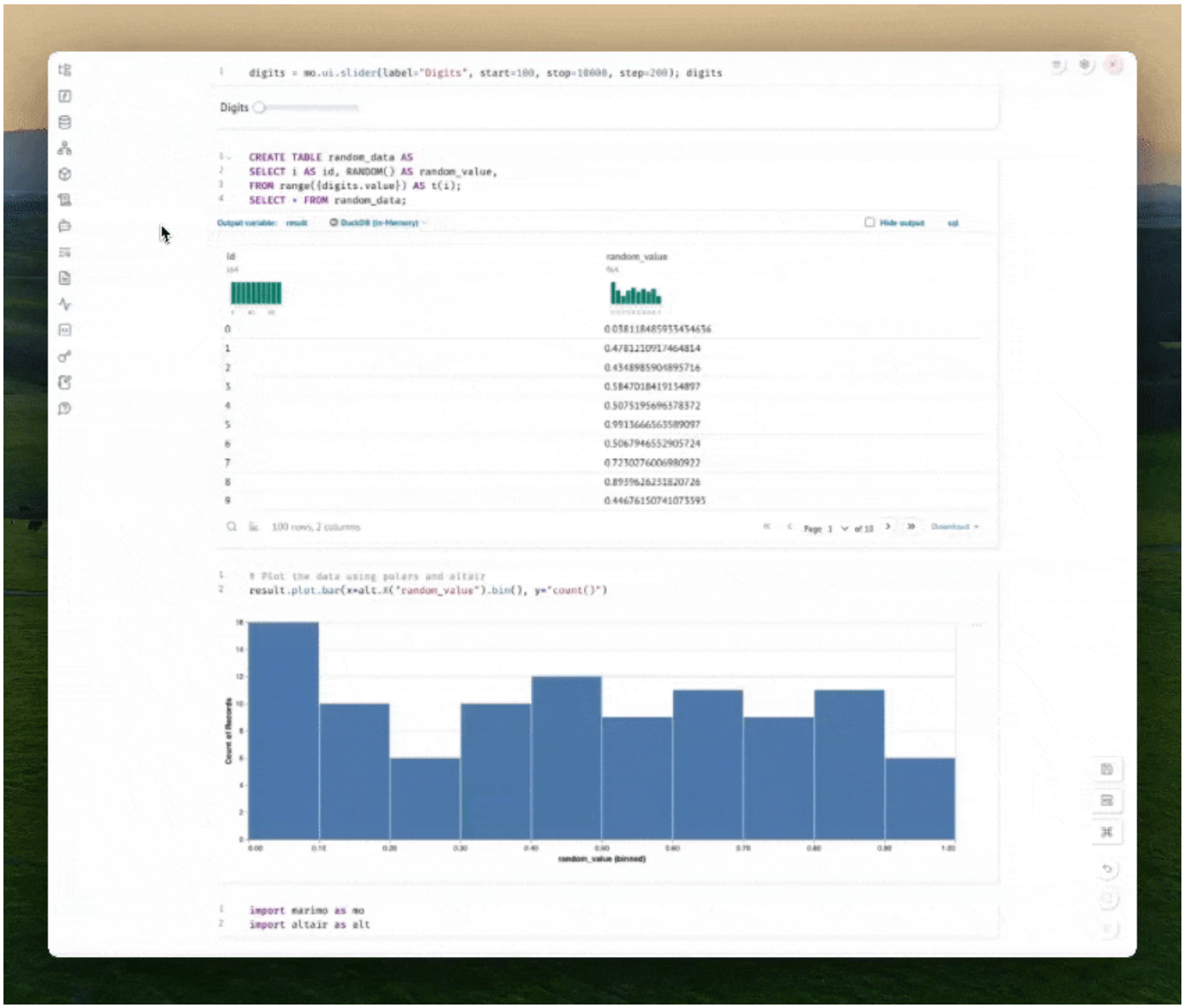
marimo allows you to create reactive SQL cells that automatically update when their dependencies change. **Working with expensive queries or large datasets?** You can configure marimo's runtime to be “lazy”. By doing so, dependent cells are only marked as stale, letting the user choose when they should be re-run.

```
digits = mo.ui.slider(label="Digits", start=100, stop=10000, step=200)
digits
```

```
CREATE TABLE random_data AS
    SELECT i AS id, random() AS random_value,
    FROM range({digits.value}) AS t(i);

SELECT * FROM random_data;
```

Interacting with UI elements, like a slider, makes your data more tangible.



DuckDB-Powered OLAP Analytics in marimo

marimo provides several features that work well with DuckDB for analytical workflows:

- Seamless integration between Python and SQL
- Reactive execution that automatically updates dependent cells when queries change
- Interactive UI elements that can be used to parameterize SQL queries
- Ability to export notebooks as standalone applications or Python scripts, or even run entirely in the browser with WebAssembly.

Next Steps

- Read the marimo docs.
- Try the SQL tutorial: `marimo tutorial sql`{:.language-sql .highlight}.
- The code for this guide is available on GitHub. Run it with `marimo edit <github_url>`{:.language-sql .highlight}.

SQL on Pandas

Pandas DataFrames stored in local variables can be queried as if they are regular tables within DuckDB.

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# query the Pandas DataFrame "my_df"
# Note: duckdb.sql connects to the default in-memory database connection
results = duckdb.sql("SELECT * FROM my_df").df()
```

The seamless integration of Pandas DataFrames into DuckDB SQL queries is allowed by replacement scans, which replace instances of accessing the `my_df` table (which does not exist in DuckDB) with a table function that reads the `my_df` dataframe.

Import from Pandas

`CREATE TABLE ... AS` and `INSERT INTO` can be used to create a table from any query. We can then create tables or insert into existing tables by referring to the Pandas DataFrame in the query. There is no need to register the DataFrames manually – DuckDB can find them in the Python process by name thanks to replacement scans.

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# create the table "my_table" from the DataFrame "my_df"
# Note: duckdb.sql connects to the default in-memory database connection
duckdb.sql("CREATE TABLE my_table AS SELECT * FROM my_df")

# insert into the table "my_table" from the DataFrame "my_df"
duckdb.sql("INSERT INTO my_table SELECT * FROM my_df")
```

If the order of columns is different or not all columns are present in the DataFrame, use `INSERT INTO ... BY NAME`:

```
duckdb.sql("INSERT INTO my_table BY NAME SELECT * FROM my_df")
```

Alternatively, the `append` method appends a DataFrame to an existing table directly, without writing a SQL query. Pass `by_name=True` to match columns by name rather than by position:

```
con = duckdb.connect()
con.sql("CREATE TABLE my_table (a INTEGER)")
con.append("my_table", my_df)
```

See Also

DuckDB also supports exporting to Pandas.

Export to Pandas

The result of a query can be converted to a Pandas DataFrame using the `df()` function.

```
import duckdb

# read the result of an arbitrary SQL query to a Pandas DataFrame
results = duckdb.sql("SELECT 42").df()
results
```

```
   42
0  42
```

See Also

DuckDB also supports importing from Pandas.

Import from Numpy

It is possible to query Numpy arrays from DuckDB. There is no need to register the arrays manually – DuckDB can find them in the Python process by name thanks to replacement scans. For example:

```
import duckdb
import numpy as np

columns = {
    "id": np.array([1, 2, 3]),
    "value": np.array([9.0, 8.0, 7.0]),
}
print(duckdb.sql("SELECT * FROM columns").fetchall())
```

```
[(1, 9.0), (2, 8.0), (3, 7.0)]
```

Each array supplies one column. DuckDB also accepts a two-dimensional array, treating each outer array as a column. Transpose an array whose outer dimension represents rows before querying it.

A masked array is read with its mask: masked elements become `NULL`.

```
my_masked_arr = np.ma.masked_array([1.0, 2.0, 3.0], mask=[False, True, False])

duckdb.sql("SELECT * FROM my_masked_arr")
```

```
┌─────────┐
│ column0 │
│ double  │
├─────────┤
│     1.0 │
│    NULL │
│     3.0 │
└─────────┘
```

See Also

DuckDB also supports exporting to Numpy.

Export to Numpy

The result of a query can be converted to Numpy arrays using the `fetchnumpy()` function, which returns a dictionary that maps each column name to a Numpy array. For example:

```
import duckdb
import numpy as np

my_arr = duckdb.sql("SELECT unnest([1, 2, 3]) AS x, 5.0 AS y").fetchnumpy()
my_arr
```

```
{'x': array([1, 2, 3], dtype=int32), 'y': array([5., 5., 5.])}
```

Then, the array can be processed using Numpy functions, e.g.:

```
np.sum(my_arr['x'])
```

```
np.int64(6)
```

See Also

DuckDB also supports importing from Numpy.

SQL on Apache Arrow

DuckDB can query multiple different types of Apache Arrow objects.

> DuckDB pushes the filters of a query that it can translate, such as comparisons, into the scan of an Arrow dataset. Arrow tables, record batches, scanners and RecordBatchReaders are read in full and filtered afterwards. A RecordBatchReader can be scanned only once; to query its data more than once, read it into a table with `read_all()` before the first query. Arrow tables are scanned on all threads, so an aggregate that depends on row order, such as `first(i)` without ORDER BY, can return a different row on each run.

Apache Arrow Tables

Arrow Tables stored in local variables can be queried as if they are regular tables within DuckDB.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow Table
results = con.sql("SELECT * FROM my_arrow_table WHERE i = 2").to_arrow_table()
```

Apache Arrow Datasets

Arrow Datasets stored as variables can also be queried as if they were regular tables. Datasets are useful to point towards directories of Parquet files to analyze large datasets. DuckDB will push column selections and the row filters it can translate down into the dataset scan operation so that only the necessary data is pulled into memory.

```
import duckdb
import pyarrow as pa
import tempfile
import pyarrow.parquet as pq
import pyarrow.dataset as ds

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create example Parquet files and save in a folder
parquet_folder = tempfile.mkdtemp()
pq.write_to_dataset(my_arrow_table, parquet_folder)
```

```
# link to Parquet files using an Arrow Dataset
my_arrow_dataset = ds.dataset(parquet_folder)

# query the Apache Arrow Dataset "my_arrow_dataset" and return as an Arrow Table
results = con.sql("SELECT * FROM my_arrow_dataset WHERE i = 2").to_arrow_table()
```

Apache Arrow Scanners

Arrow Scanners stored as variables can also be queried as if they were regular tables. Scanners read over a dataset and select specific columns or apply row-wise filtering. This is similar to how DuckDB pushes column selections and filters down into an Arrow Dataset, but using Arrow compute operations instead. Arrow can use asynchronous IO to quickly access files.

```
import duckdb
import pyarrow as pa
import tempfile
import pyarrow.parquet as pq
import pyarrow.dataset as ds
import pyarrow.compute as pc

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create example Parquet files and save in a folder
parquet_folder = tempfile.mkdtemp()
pq.write_to_dataset(my_arrow_table, parquet_folder)

# link to Parquet files using an Arrow Dataset
my_arrow_dataset = ds.dataset(parquet_folder)

# define the filter to be applied while scanning
# equivalent to "WHERE i = 2"
scanner_filter = (pc.field("i") == pc.scalar(2))

arrow_scanner = ds.Scanner.from_dataset(my_arrow_dataset, filter = scanner_filter)

# query the Apache Arrow scanner "arrow_scanner" and return as an Arrow Table
results = con.sql("SELECT * FROM arrow_scanner").to_arrow_table()
```

Apache Arrow RecordBatchReaders

Arrow RecordBatchReaders are a reader for Arrow's streaming binary format and can also be queried directly as if they were tables. This streaming format is useful when sending Arrow data for tasks like interprocess communication or communicating between language runtimes.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()

my_recordbatch = pa.RecordBatch.from_pydict({'i': [1, 2, 3, 4],
                                             'j': ["one", "two", "three", "four"]})

my_recordbatchreader = pa.ipc.RecordBatchReader.from_batches(my_recordbatch.schema, [my_recordbatch])

# query the Apache Arrow RecordBatchReader "my_recordbatchreader" and return as an Arrow Table
results = con.sql("SELECT * FROM my_recordbatchreader WHERE i = 2").to_arrow_table()
```

Import from Apache Arrow

`CREATE TABLE AS` and `INSERT INTO` can be used to create a table from any query. We can then create tables or insert into existing tables by referring to the Apache Arrow object in the query. This example imports from an Arrow Table, but DuckDB can query different Apache Arrow formats as seen in the SQL on Arrow guide.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
my_arrow = pa.Table.from_pydict({'a': [42]})

# create the table "my_table" from the DataFrame "my_arrow"
duckdb.sql("CREATE TABLE my_table AS SELECT * FROM my_arrow")

# insert into the table "my_table" from the DataFrame "my_arrow"
duckdb.sql("INSERT INTO my_table SELECT * FROM my_arrow")
```

Export to Apache Arrow

All results of a query can be exported to an Apache Arrow Table using the `to_arrow_table` function. Alternatively, results can be returned as a RecordBatchReader using the `to_arrow_reader` function and results can be read one batch at a time.

Call these functions on the relation that `duckdb.sql`, `con.sql` or DuckDB's Relational API returns: its reader streams the result. On a `con.execute()` result they first copy all rows and warn with a `UserWarning`, and the reader does not stream. While a reader is partly read, other queries on its connection raise a `ResourceInUseException`. Read the reader to the end or close it first, or run the other queries on a cursor from `con.cursor()`.

> **Deprecated.** On relations, `fetch_arrow_table`, `fetch_record_batch` and `fetch_arrow_reader` are deprecated aliases of `to_arrow_table` and `to_arrow_reader`. Results of `con.execute()` and the duckdb module no longer have them.

Export to an Arrow Table

```
import duckdb
import pyarrow as pa

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow Table
results = duckdb.sql("SELECT * FROM my_arrow_table").to_arrow_table()
```

Export as a RecordBatchReader

```
import duckdb

con = duckdb.connect()
rel = con.sql("SELECT i FROM range(5) AS t(i)")
with rel.to_arrow_reader(batch_size=2) as reader:
    for batch in reader:
        print(batch.to_pydict())
```

```
{'i': [0, 1]}
{'i': [2, 3]}
{'i': [4]}
```

The small batch size makes the batches visible here. Omit it to use the default size.

Export from Relational API

Arrow objects can also be exported from the Relational API. A relation can be converted to an Arrow table using `DuckDBPyRelation.to_arrow_table`, and to an Arrow record batch reader using `DuckDBPyRelation.to_arrow_reader`.

```
import duckdb

# connect to an in-memory database
con = duckdb.connect()

con.execute('CREATE TABLE integers (i integer)')
con.execute('INSERT INTO integers VALUES (0), (1), (2), (3), (4), (5), (6), (7), (8), (9), (NULL)')

# Create a relation from the table and export the entire relation as Arrow
rel = con.table("integers")
relation_as_arrow = rel.to_arrow_table()

# Calculate a result using that relation and export that result to Arrow
res = rel.aggregate("sum(i)")
arrow_table = res.to_arrow_table()

# You can also create an Arrow record batch reader from a relation
arrow_batch_reader = res.to_arrow_reader()
for batch in arrow_batch_reader:
    # Process a single chunk here
    print(batch.to_pandas())
```

Export through the Arrow PyCapsule Interface

A relation implements `__arrow_c_stream__` of the Arrow PyCapsule interface, so libraries that accept an Arrow stream read it directly: `pyarrow.table(rel)`, `pyarrow.RecordBatchReader.from_stream(rel)` and `polars.DataFrame(rel)`. Polars reads it this way even without `pyarrow` installed.

```
import duckdb
import polars as pl
import pyarrow as pa

rel = duckdb.sql("SELECT i, 'v' || i AS s FROM range(3) AS t(i)")
print(pa.table(rel).to_pydict())
print(pl.DataFrame(rel))
```

```
{'i': [0, 1, 2], 's': ['v0', 'v1', 'v2']}
shape: (3, 2)
```

i	s
i64	str
0	v0
1	v1
2	v2

Relational API on Pandas

DuckDB offers a relational API that can be used to chain together query operations. These are lazily evaluated so that DuckDB can optimize their execution. These operators can act on Pandas DataFrames, DuckDB tables or views (which can point to any underlying storage format that DuckDB can read, such as CSV or Parquet files, etc.). Here we show a simple example of reading from a Pandas DataFrame and returning a DataFrame.

```python
import duckdb
import pandas

# connect to an in-memory database
con = duckdb.connect()

input_df = pandas.DataFrame.from_dict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create a DuckDB relation from a dataframe
rel = con.from_df(input_df)

# chain together relational operators (this is a lazy operation, so the operations are not yet executed)
# equivalent to: SELECT i, j, i*2 AS two_i FROM input_df WHERE i >= 2 ORDER BY i DESC LIMIT 2
transformed_rel = rel.filter('i >= 2').select('i, j, i*2 AS two_i').order('i DESC').limit(2)

# trigger execution by requesting .df() of the relation
# .df() could have been added to the end of the chain above - it was separated for clarity
output_df = transformed_rel.df()
```

Relational operators can also be used to group rows, aggregate, find distinct combinations of values, join, union and more. They are also able to directly insert results into a DuckDB table or write to a CSV.

Please see the Relational API page for more examples and the available relational methods on the `DuckDBPyRelation` class.

Multiple Python Threads

This page demonstrates how to simultaneously insert into and read from a DuckDB database across multiple Python threads. This could be useful in scenarios where new data is flowing in and an analysis should be periodically re-run. Note that this is all within a single Python process (see the FAQ for details on DuckDB concurrency). Feel free to follow along in this Google Colab notebook.

Setup

First, import DuckDB and several modules from the Python standard library. Note: if using Pandas, add `import pandas` at the top of the script as well (as it must be imported prior to the multi-threading). Then connect to a file-backed DuckDB database and create an example table to store inserted data. This table will track the name of the thread that completed the insert and automatically insert the timestamp when that insert occurred using the `DEFAULT` expression.

```python
import duckdb
from threading import Thread, current_thread
import random

duckdb_con = duckdb.connect('my_persistent_db.duckdb')
# Use connect without parameters for an in-memory database
# duckdb_con = duckdb.connect()
duckdb_con.execute("""
    CREATE OR REPLACE TABLE my_inserts (
        thread_name VARCHAR,
        insert_time TIMESTAMP DEFAULT current_timestamp
    )
""")
```

Reader and Writer Functions

Next, define functions to be executed by the writer and reader threads. Each thread must use the `.cursor()` method to create a thread-local connection to the same DuckDB file based on the original connection. This approach also works with in-memory DuckDB databases. A connection serves one call at a time: a call from a second thread while another call is running raises a `ResourceInUseException` instead of waiting.

```
def write_from_thread(duckdb_con):
    # Create a DuckDB connection specifically for this thread
    local_con = duckdb_con.cursor()
    # Insert a row with the name of the thread. insert_time is auto-generated.
    thread_name = str(current_thread().name)
    result = local_con.execute("""
        INSERT INTO my_inserts (thread_name)
        VALUES (?)
    """, (thread_name,)).fetchall()

def read_from_thread(duckdb_con):
    # Create a DuckDB connection specifically for this thread
    local_con = duckdb_con.cursor()
    # Query the current row count
    thread_name = str(current_thread().name)
    results = local_con.execute("""
        SELECT
            ? AS thread_name,
            count(*) AS row_counter,
            current_localtimestamp() AS read_time
        FROM my_inserts
    """, (thread_name,)).fetchall()
    print(results)
```

Create Threads

We define how many writers and readers to use, and define a list to track all of the threads that will be created. Then, create first writer and then reader threads. Next, shuffle them so that they will be kicked off in a random order to simulate simultaneous writers and readers. Note that the threads have not yet been executed, only defined.

```
write_thread_count = 50
read_thread_count = 5
threads = []

# Create multiple writer and reader threads (in the same process)
# Pass in the same connection as an argument
for i in range(write_thread_count):
    threads.append(Thread(target = write_from_thread,
                          args = (duckdb_con,),
                          name = 'write_thread_' + str(i)))

for j in range(read_thread_count):
    threads.append(Thread(target = read_from_thread,
                          args = (duckdb_con,),
                          name = 'read_thread_' + str(j)))

# Shuffle the threads to simulate a mix of readers and writers
random.seed(6) # Set the seed to ensure consistent results when testing
random.shuffle(threads)
```

Run Threads and Show Results

Now, kick off all threads to run in parallel, then wait for all of them to finish before printing out the results. Note that the timestamps of readers and writers are interspersed as expected due to the randomization.

```
# Kick off all threads in parallel
for thread in threads:
    thread.start()

# Ensure all threads complete before printing final results
for thread in threads:
    thread.join()
```

```
print(duckdb_con.execute("""
    SELECT *
    FROM my_inserts
    ORDER BY
        insert_time
""").df())
```

Integration with Ibis

Ibis is a Python dataframe library that supports 20+ backends, with DuckDB as the default. Ibis with DuckDB provides a Pythonic interface for SQL with great performance.

> **Warning.** Ibis 12.0.0 and earlier do not support DuckDB 2.0. The examples below use DuckDB 1.5. Follow the Installation section to keep DuckDB below 2.0.

Installation

Install Ibis with the DuckDB backend, keeping DuckDB below 2.0:

```
pip install 'ibis-framework[duckdb,examples]' 'duckdb<2'
```

The `examples` extra provides the sample data used below.

Create a Database File

Ibis can work with several file types, but at its core, it connects to existing databases and interacts with the data there. You can get started with your own DuckDB databases or create a new one with example data.

```
import ibis

con = ibis.connect("duckdb://penguins.ddb")
if "penguins" not in con.list_tables():
    con.create_table(
        "penguins", ibis.memtable(ibis.examples.penguins.fetch().to_pyarrow())
    )
```

You can now see the example dataset copied over to the database:

```
# reconnect to the persisted database (dropping temp tables)
con = ibis.connect("duckdb://penguins.ddb")
con.list_tables()
```

```
['penguins']
```

There's one table, called `penguins`. We can ask Ibis to give us an object that we can interact with.

```
penguins = con.table("penguins")
penguins
```

```
DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64
```

Ibis is lazily evaluated, so instead of seeing the data, we see the schema of the table. To peek at the data, we can call `head` and then `to_pandas` to get the first few rows of the table as a pandas DataFrame.

```
penguins.head().to_pandas()
```

```
  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  body_mass_g     sex  year
0  Adelie  Torgersen            39.1           18.7              181.0       3750.0    male  2007
1  Adelie  Torgersen            39.5           17.4              186.0       3800.0  female  2007
2  Adelie  Torgersen            40.3           18.0              195.0       3250.0  female  2007
3  Adelie  Torgersen             NaN            NaN                NaN          NaN     NaN  2007
4  Adelie  Torgersen            36.7           19.3              193.0       3450.0  female  2007
```

`to_pandas` takes the existing lazy table expression and evaluates it. If we leave it off, you'll see the Ibis representation of the table expression that `to_pandas` will evaluate (when you're ready!).

```
penguins.head()
```

```
r0 := DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64

Limit[r0, n=5]
```

Ibis returns results as a pandas DataFrame using `to_pandas`, but isn't using pandas to perform any of the computation. The query is executed by DuckDB. Only when `to_pandas` is called does Ibis then pull back the results and convert them into a DataFrame.

Interactive Mode

For the rest of this intro, we'll turn on interactive mode, which partially executes queries to give users a preview of the results. There is a small difference in the way the output is formatted, but otherwise this is the same as calling `to_pandas` on the table expression with a limit of 10 result rows returned.

```
ibis.options.interactive = True
penguins.head()
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
string	string	float64	float64	int64	int64	string	int64
Adelie	Torgersen	39.1	18.7	181	3750	male	2007
Adelie	Torgersen	39.5	17.4	186	3800	female	2007
Adelie	Torgersen	40.3	18.0	195	3250	female	2007
Adelie	Torgersen	NULL	NULL	NULL	NULL	NULL	2007
Adelie	Torgersen	36.7	19.3	193	3450	female	2007

Common Operations

Ibis has a collection of useful table methods to manipulate and query the data in a table.

filter

`filter` allows you to select rows based on a condition or set of conditions.

We can filter so we only have penguins of the species Gentoo:

```
penguins.filter(penguins.species == "Gentoo")
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
string	string	float64	float64	int64	int64	string	int64
Gentoo	Biscoe	46.1	13.2	211	4500	female	2007
Gentoo	Biscoe	50.0	16.3	230	5700	male	2007
Gentoo	Biscoe	48.7	14.1	210	4450	female	2007
Gentoo	Biscoe	50.0	15.2	218	5700	male	2007
Gentoo	Biscoe	47.6	14.5	215	5400	male	2007
Gentoo	Biscoe	46.5	13.5	210	4550	female	2007
Gentoo	Biscoe	45.4	14.6	211	4800	female	2007
Gentoo	Biscoe	46.7	15.3	219	5200	male	2007
Gentoo	Biscoe	43.3	13.4	209	4400	female	2007
Gentoo	Biscoe	46.8	15.4	215	5150	male	2007
…	…	…	…	…	…	…	…

Or filter for Gentoo penguins that have a body mass larger than 6 kg.

```
penguins.filter((penguins.species == "Gentoo") & (penguins.body_mass_g > 6000))
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
string	string	float64	float64	int64	int64	string	int64
Gentoo	Biscoe	49.2	15.2	221	6300	male	2007
Gentoo	Biscoe	59.6	17.0	230	6050	male	2007

You can use any Boolean comparison in a filter.

select

Your data analysis might not require all the columns present in a given table. `select` lets you pick out only those columns that you want to work with.

To select a column you can use the name of the column as a string:

```
penguins.select("species", "island", "year").limit(3)
```

species	island	year
string	string	int64
Adelie	Torgersen	2007
Adelie	Torgersen	2007
Adelie	Torgersen	2007

Or you can use column objects directly (this can be convenient when paired with tab-completion):

```
penguins.select(penguins.species, penguins.island, penguins.year).limit(3)
```

species	island	year
string	string	int64
Adelie	Torgersen	2007

```
┌─────────┬────────────┬─────────┐
│ Adelie  │ Torgersen  │  2007   │
│ Adelie  │ Torgersen  │  2007   │
└─────────┴────────────┴─────────┘
```

Or you can mix-and-match:

```
penguins.select("species", "island", penguins.year).limit(3)
```

```
┌─────────┬────────────┬─────────┐
│ species │ island     │ year    │
│ string  │ string     │ int64   │
├─────────┼────────────┼─────────┤
│ Adelie  │ Torgersen  │  2007   │
│ Adelie  │ Torgersen  │  2007   │
│ Adelie  │ Torgersen  │  2007   │
└─────────┴────────────┴─────────┘
```

mutate

`mutate` lets you add new columns to your table, derived from the values of existing columns.

```
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10)
```

```
┌─────────┬────────────┬────────────────┬───────────────┬───────────────────┬─────────────┬─────────┬─────────┬────────────────┐
│ species │ island     │ bill_length_mm │ bill_depth_mm │ flipper_length_mm │ body_mass_g │ sex     │ year    │ bill_length_cm │
├─────────┼────────────┼────────────────┼───────────────┼───────────────────┼─────────────┼─────────┼─────────┼────────────────┤
│ string  │ string     │ float64        │ float64       │ int64             │ int64       │ string  │ int64   │ float64        │
├─────────┼────────────┼────────────────┼───────────────┼───────────────────┼─────────────┼─────────┼─────────┼────────────────┤
│ Adelie  │ Torgersen  │           39.1 │          18.7 │               181 │        3750 │ male    │  2007   │           3.91 │
│ Adelie  │ Torgersen  │           39.5 │          17.4 │               186 │        3800 │ female  │  2007   │           3.95 │
│ Adelie  │ Torgersen  │           40.3 │          18.0 │               195 │        3250 │ female  │  2007   │           4.03 │
│ Adelie  │ Torgersen  │           NULL │          NULL │              NULL │        NULL │ NULL    │  2007   │           NULL │
│ Adelie  │ Torgersen  │           36.7 │          19.3 │               193 │        3450 │ female  │  2007   │           3.67 │
│ Adelie  │ Torgersen  │           39.3 │          20.6 │               190 │        3650 │ male    │  2007   │           3.93 │
│ Adelie  │ Torgersen  │           38.9 │          17.8 │               181 │        3625 │ female  │  2007   │           3.89 │
│ Adelie  │ Torgersen  │           39.2 │          19.6 │               195 │        4675 │ male    │  2007   │           3.92 │
│ Adelie  │ Torgersen  │           34.1 │          18.1 │               193 │        3475 │ NULL    │  2007   │           3.41 │
│ Adelie  │ Torgersen  │           42.0 │          20.2 │               190 │        4250 │ NULL    │  2007   │           4.20 │
│ …       │ …          │              … │             … │                 … │           … │ …       │     …   │              … │
└─────────┴────────────┴────────────────┴───────────────┴───────────────────┴─────────────┴─────────┴─────────┴────────────────┘
```

Notice that the table is a little too wide to display all the columns now (depending on your screen-size). `bill_length` is now present in millimeters *and* centimeters. Use a `select` to trim down the number of columns we're looking at.

```
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(
    "species",
    "island",
    "bill_depth_mm",
```

```
    "flipper_length_mm",
    "body_mass_g",
    "sex",
    "year",
    "bill_length_cm",
)
```

species	island	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year	bill_length_cm
string	string	float64	int64	int64	string	int64	float64
Adelie	Torgersen	18.7	181	3750	male	2007	3.91
Adelie	Torgersen	17.4	186	3800	female	2007	3.95
Adelie	Torgersen	18.0	195	3250	female	2007	4.03
Adelie	Torgersen	NULL	NULL	NULL	NULL	2007	NULL
Adelie	Torgersen	19.3	193	3450	female	2007	3.67
Adelie	Torgersen	20.6	190	3650	male	2007	3.93
Adelie	Torgersen	17.8	181	3625	female	2007	3.89
Adelie	Torgersen	19.6	195	4675	male	2007	3.92
Adelie	Torgersen	18.1	193	3475	NULL	2007	3.41
Adelie	Torgersen	20.2	190	4250	NULL	2007	4.20
…	…	…	…	…	…	…	…

selectors

Typing out *all* of the column names *except* one is a little annoying. Instead of doing that again, we can use a `selector` to quickly select or deselect groups of columns.

```
import ibis.selectors as s

penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(
    ~s.matches("bill_length_mm")
    # match every column except `bill_length_mm`
)
```

species	island	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year	bill_length_cm
string	string	float64	int64	int64	string	int64	float64
Adelie	Torgersen	18.7	181	3750	male	2007	3.91
Adelie	Torgersen	17.4	186	3800	female	2007	3.95
Adelie	Torgersen	18.0	195	3250	female	2007	4.03
Adelie	Torgersen	NULL	NULL	NULL	NULL	2007	NULL
Adelie	Torgersen	19.3	193	3450	female	2007	3.67
Adelie	Torgersen	20.6	190	3650	male	2007	3.93
Adelie	Torgersen	17.8	181	3625	female	2007	3.89
Adelie	Torgersen	19.6	195	4675	male	2007	3.92
Adelie	Torgersen	18.1	193	3475	NULL	2007	3.41
Adelie	Torgersen	20.2	190	4250	NULL	2007	4.20
…	…	…	…	…	…	…	…

You can also use a `selector` alongside a column name.

```
penguins.select("island", s.numeric())
```

island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	year
string	float64	float64	int64	int64	int64

Torgersen	39.1	18.7	181	3750	2007
Torgersen	39.5	17.4	186	3800	2007
Torgersen	40.3	18.0	195	3250	2007
Torgersen	NULL	NULL	NULL	NULL	2007
Torgersen	36.7	19.3	193	3450	2007
Torgersen	39.3	20.6	190	3650	2007
Torgersen	38.9	17.8	181	3625	2007
Torgersen	39.2	19.6	195	4675	2007
Torgersen	34.1	18.1	193	3475	2007
Torgersen	42.0	20.2	190	4250	2007
…	…	…	…	…	…

You can read more about `selectors` in the docs!

`order_by`

`order_by` arranges the values of one or more columns in ascending or descending order.

By default, `ibis` sorts in ascending order:

```
penguins.order_by(penguins.flipper_length_mm).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Adelie	Biscoe	172
Adelie	Biscoe	174
Adelie	Torgersen	176
Adelie	Dream	178
Adelie	Dream	178
Adelie	Dream	178
Chinstrap	Dream	178
Adelie	Dream	179
Adelie	Torgersen	180
Adelie	Biscoe	180
…	…	…

You can sort in descending order using the `desc` method of a column:

```
penguins.order_by(penguins.flipper_length_mm.desc()).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Gentoo	Biscoe	231
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230

Gentoo	Biscoe	229
Gentoo	Biscoe	229
…	…	…

Or you can use `ibis.desc`

```
penguins.order_by(ibis.desc("flipper_length_mm")).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Gentoo	Biscoe	231
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	229
Gentoo	Biscoe	229
…	…	…

aggregate

Ibis has several aggregate functions available to help summarize data.

`mean`, `max`, `min`, `count`, `sum` (the list goes on).

To aggregate an entire column, call the corresponding method on that column.

```
penguins.flipper_length_mm.mean()
```

200.915205

You can compute multiple aggregates at once using the `aggregate` method:

```
penguins.aggregate([penguins.flipper_length_mm.mean(), penguins.bill_depth_mm.max()])
```

Mean(flipper_length_mm)	Max(bill_depth_mm)
float64	float64
200.915205	21.5

But `aggregate` *really* shines when it's paired with `group_by`.

group_by

`group_by` creates groupings of rows that have the same value for one or more columns.

But it doesn't do much on its own: you can pair it with `aggregate` to get a result.

```
penguins.group_by("species").aggregate().order_by("species")
```

species
string
Adelie Chinstrap Gentoo

We grouped by the `species` column and handed it an "empty" aggregate command. The result of that is a column of the unique values in the `species` column.

If we add a second column to the `group_by`, we'll get each unique pairing of the values in those columns.

```
penguins.group_by(["species", "island"]).aggregate().order_by(["species", "island"])
```

species	island
string	string
Adelie	Biscoe
Adelie	Dream
Adelie	Torgersen
Chinstrap	Dream
Gentoo	Biscoe

Now, if we add an aggregation function to that, we start to really open things up.

```
penguins.group_by(["species", "island"]).aggregate(penguins.bill_length_mm.mean()).order_by(
    ["species", "island"]
)
```

species	island	Mean(bill_length_mm)
string	string	float64
Adelie	Biscoe	38.975000
Adelie	Dream	38.501786
Adelie	Torgersen	38.950980
Chinstrap	Dream	48.833824
Gentoo	Biscoe	47.504878

By adding that `mean` to the `aggregate`, we now have a concise way to calculate aggregates over each of the distinct groups in the `group_by`. And we can calculate as many aggregates as we need.

```
penguins.group_by(["species", "island"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
).order_by(["species", "island"])
```

species	island	Mean(bill_length_mm)	Max(flipper_length_mm)
string	string	float64	int64
Adelie	Biscoe	38.975000	203
Adelie	Dream	38.501786	208
Adelie	Torgersen	38.950980	210
Chinstrap	Dream	48.833824	212
Gentoo	Biscoe	47.504878	231

If we need more specific groups, we can add to the `group_by`.

```
penguins.group_by(["species", "island", "sex"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
).order_by(["species", "island", "sex"])
```

species	island	sex	Mean(bill_length_mm)	Max(flipper_length_mm)
string	string	string	float64	int64
Adelie	Biscoe	female	37.359091	199
Adelie	Biscoe	male	40.590909	203
Adelie	Dream	female	36.911111	202
Adelie	Dream	male	40.071429	208
Adelie	Dream	NULL	37.500000	179
Adelie	Torgersen	female	37.554167	196
Adelie	Torgersen	male	40.586957	210
Adelie	Torgersen	NULL	37.925000	193
Chinstrap	Dream	female	46.573529	202
Chinstrap	Dream	male	51.094118	212
…	…	…	…	…

Chaining It All Together

We've already chained some Ibis calls together. We used `mutate` to create a new column and then `select` to only view a subset of the new table. We were just chaining `group_by` with `aggregate`.

There's nothing stopping us from putting all of these concepts together to ask questions of the data.

How about:

- What was the largest female penguin (by body mass) on each island in the year 2008?

```
penguins.filter((penguins.sex == "female") & (penguins.year == 2008)).group_by(
    ["island"]
).aggregate(penguins.body_mass_g.max()).order_by("island")
```

island	Max(body_mass_g)
string	int64
Biscoe	5200
Dream	3900
Torgersen	3800

- What about the largest male penguin (by body mass) on each island for each year of data collection?

```
penguins.filter(penguins.sex == "male").group_by(["island", "year"]).aggregate(
    penguins.body_mass_g.max().name("max_body_mass")
).order_by(["year", "max_body_mass"])
```

island	year	max_body_mass
string	int64	int64
Dream	2007	4650
Torgersen	2007	4675
Biscoe	2007	6300
Torgersen	2008	4700

```
┌──────────┬──────┬──────────────┐
│ Dream    │ 2008 │         4800 │
│ Biscoe   │ 2008 │         6000 │
│ Torgersen│ 2009 │         4300 │
│ Dream    │ 2009 │         4475 │
│ Biscoe   │ 2009 │         6000 │
└──────────┴──────┴──────────────┘
```

Learn More

That's all for this quick-start guide. If you want to learn more, check out the Ibis documentation.

Integration with Polars

Polars is a DataFrames library built in Rust with bindings for Python and Node.js. It uses Apache Arrow's columnar format as its memory model. DuckDB can read Polars DataFrames and convert query results to Polars DataFrames. It does this internally using the efficient Apache Arrow integration. Note that the `pyarrow` library must be installed for the integration to work.

Installation

```
pip install -U duckdb 'polars[pyarrow]'
```

Polars to DuckDB

DuckDB can natively query Polars DataFrames and LazyFrames by referring to their names as they exist in the current scope. DuckDB pushes the column selections of a query, and the filters it can translate, into a LazyFrame, so Polars collects only the rows and columns the query needs. A DataFrame is converted in full and filtered afterwards.

```
import duckdb
import polars as pl

df = pl.DataFrame(
    {
        "A": [1, 2, 3, 4, 5],
        "fruits": ["banana", "banana", "apple", "apple", "banana"],
        "B": [5, 4, 3, 2, 1],
        "cars": ["beetle", "audi", "beetle", "beetle", "beetle"],
    }
)
duckdb.sql("SELECT * FROM df").show()
```

DuckDB to Polars

DuckDB can output results as Polars DataFrames using the `.pl()` method of a relation, the object that `duckdb.sql` and `con.sql` return. Results of `con.execute()` have no `.pl()` method.

```
df = duckdb.sql("""
    SELECT 1 AS id, 'banana' AS fruit
    UNION ALL
    SELECT 2, 'apple'
    UNION ALL
    SELECT 3, 'mango'""""
).pl()
print(df)
```

```
shape: (3, 2)
```

id	fruit
i32	str
1	banana
2	apple
3	mango

The optional `lazy` parameter allows returning Polars LazyFrames.

```
df = duckdb.sql("""
    SELECT 1 AS id, 'banana' AS fruit
    UNION ALL
    SELECT 2, 'apple'
    UNION ALL
    SELECT 3, 'mango'""""
).pl(lazy=True)
print(df)
```

```
naive plan: (run LazyFrame.explain(optimized=True) to see the optimized plan)

PYTHON SCAN []
PROJECT */2 COLUMNS
```

A relation also implements the Arrow PyCapsule interface, so `pl.DataFrame(rel)` reads it too, without `pyarrow` installed. `pl.from_arrow(rel)` returns a Series with one struct per row instead of a DataFrame.

To learn more about Polars, feel free to explore their Python API Reference.

Integration with PyTorch

PyTorch tensors can be built from DuckDB query results by exporting each result batch as Apache Arrow. This pattern is useful when training or serving models on Parquet data because DuckDB can filter and project the input before the batches are converted to tensors.

Installation

```
pip install -U duckdb numpy pyarrow torch
```

DuckDB to PyTorch

This example queries a Parquet-backed Arrow dataset, streams the result as `RecordBatch` objects and converts each batch into feature and label tensors.

```
import pathlib
import tempfile

import duckdb
import numpy as np
import pyarrow as pa
import pyarrow.dataset as ds
import pyarrow.parquet as pq
import torch

base_path = pathlib.Path(tempfile.mkdtemp())
parquet_dir = base_path / "train"

table = pa.table(
```

```
    {
        "feature_0": [0.1, 0.3, 0.5, 0.7],
        "feature_1": [1.0, 0.0, 1.0, 0.0],
        "label": [0, 1, 1, 0],
    }
)
pq.write_to_dataset(table, str(parquet_dir))

con = duckdb.connect()
train_dataset = ds.dataset(str(parquet_dir))

reader = con.sql("""
    SELECT feature_0, feature_1, label
    FROM train_dataset
    WHERE label = 1
""").to_arrow_reader(batch_size=2)

for batch in reader:
    features = torch.tensor(
        np.column_stack(
            [
                batch.column(0).to_numpy(),
                batch.column(1).to_numpy(),
            ]
        ),
        dtype=torch.float32,
    )
    labels = torch.tensor(batch.column(2).to_numpy(), dtype=torch.int64)

    print(features.shape, labels)
```

```
torch.Size([2, 2]) tensor([1, 1])
```

DuckDB pushes the `WHERE label = 1` filter and the selected columns into the dataset scan, so only the requested rows and columns are converted to tensors. The `to_arrow_reader` method of a relation streams the result in batches instead of materializing the full result in memory at once. While the loop holds the reader, other queries on `con` raise a `ResourceInUseException`; a loop that runs a query per batch needs a cursor from `con.cursor()`.

To get the whole result at once, the `torch()` method of a relation returns a dictionary with one tensor per column.

To learn more about querying Arrow objects directly, see the "SQL on Apache Arrow" guide and the "Export to Apache Arrow" guide.

Using fsspec Filesystems

DuckDB support for `fsspec` filesystems allows querying data in filesystems that DuckDB's `httpfs` extension does not support. `fsspec` has a large number of inbuilt filesystems, and there are also many external implementations. This capability is only available in DuckDB's Python client because `fsspec` is a Python library, while the `httpfs` extension is available in many DuckDB clients.

Example

The following is an example of using `fsspec` to query a file in Google Cloud Storage (instead of using their S3-compatible API).

Firstly, you must install `duckdb` and `fsspec`, and a filesystem interface of your choice.

```
pip install duckdb fsspec gcsfs
```

Then, you can register whichever filesystem you'd like to query:

```
import duckdb
from fsspec import filesystem

# this line will throw an exception if the appropriate filesystem interface is not installed
```

```
duckdb.register_filesystem(filesystem('gcs'))

duckdb.sql("SELECT * FROM read_csv('gcs:///bucket/file.csv')")
```

> These filesystems are not implemented in C++, hence, their performance may not be comparable to the ones provided by the `httpfs` extension. It is also worth noting that as they are third-party libraries, they may contain bugs that are beyond our control.

SQL Editors

Beekeeper Studio SQL Editor

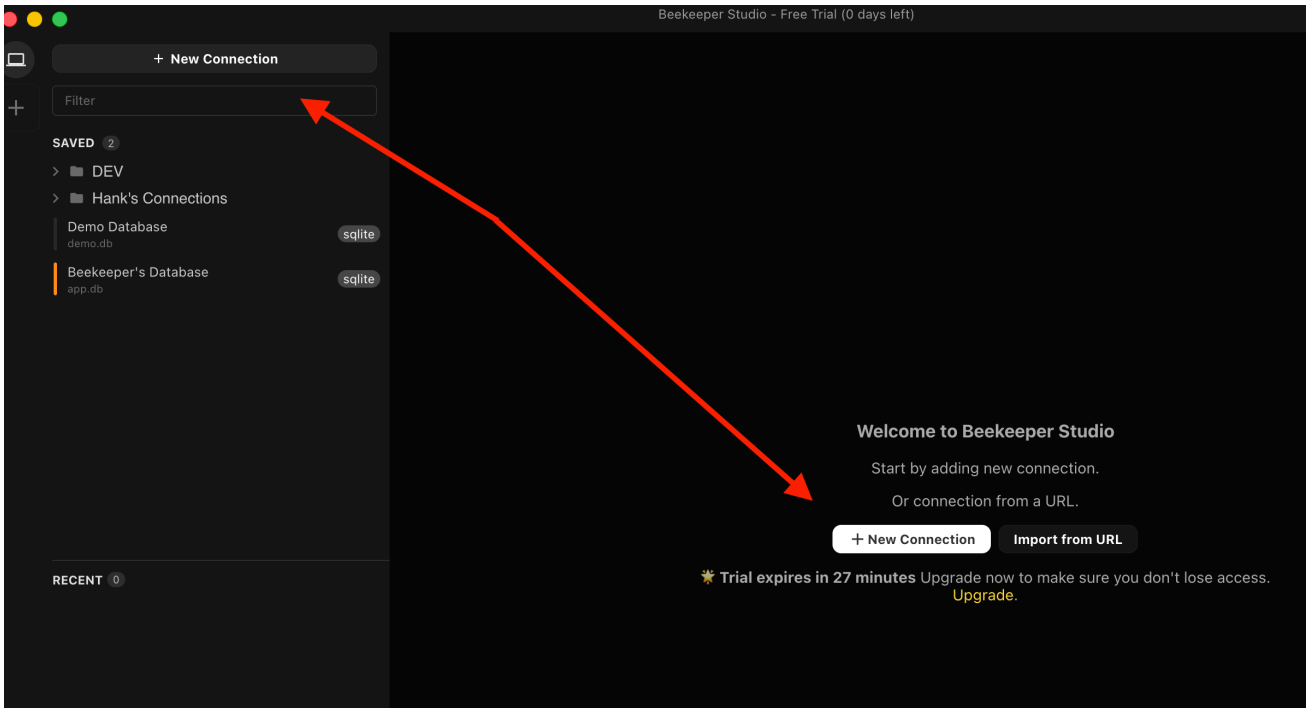
Beekeeper Studio is a free, open-source SQL editor and database manager with a modern, easy-to-use interface. It is available for Windows, macOS, and Linux, and also offers a paid Ultimate tier with cloud sync and team-sharing features. Beekeeper Studio connects to DuckDB natively rather than through a JDBC or ODBC driver, so no separate driver installation is required.

Installing Beekeeper Studio

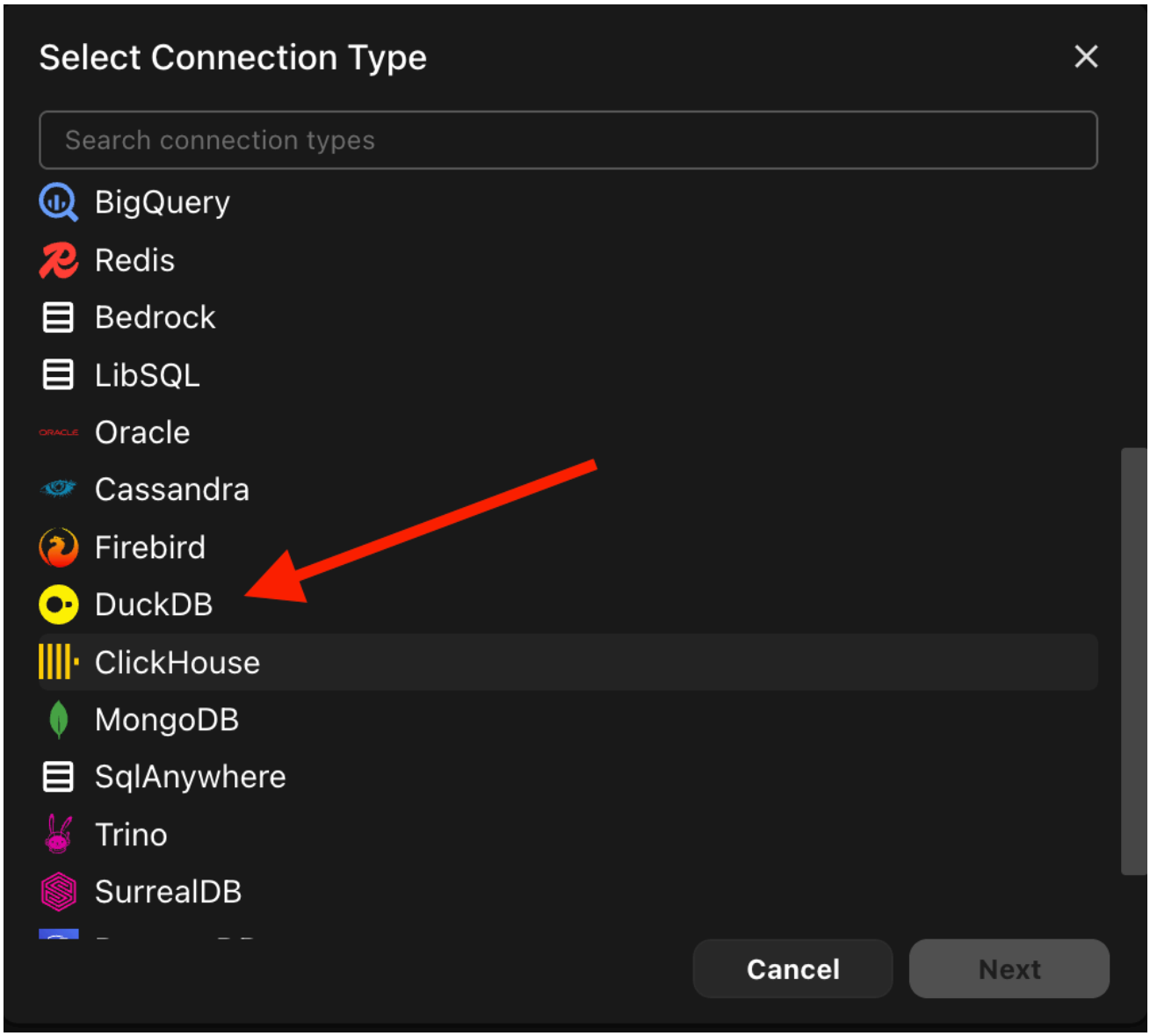
1. Install Beekeeper Studio using the download links and instructions found at their download page.

Connecting to a DuckDB Database

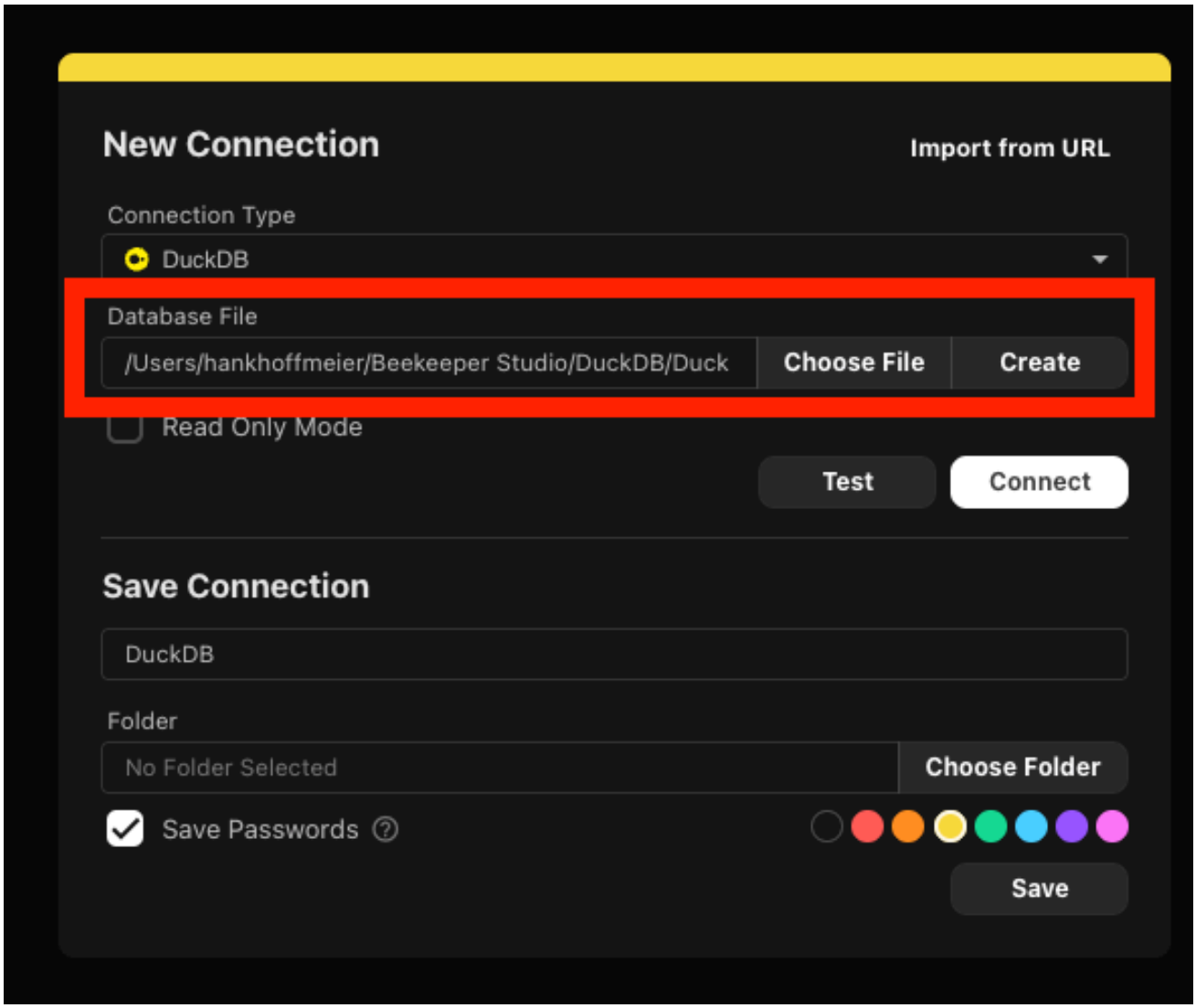
1. Open Beekeeper Studio and click **New Connection**.



2. Select **DuckDB** from the connection type dropdown.

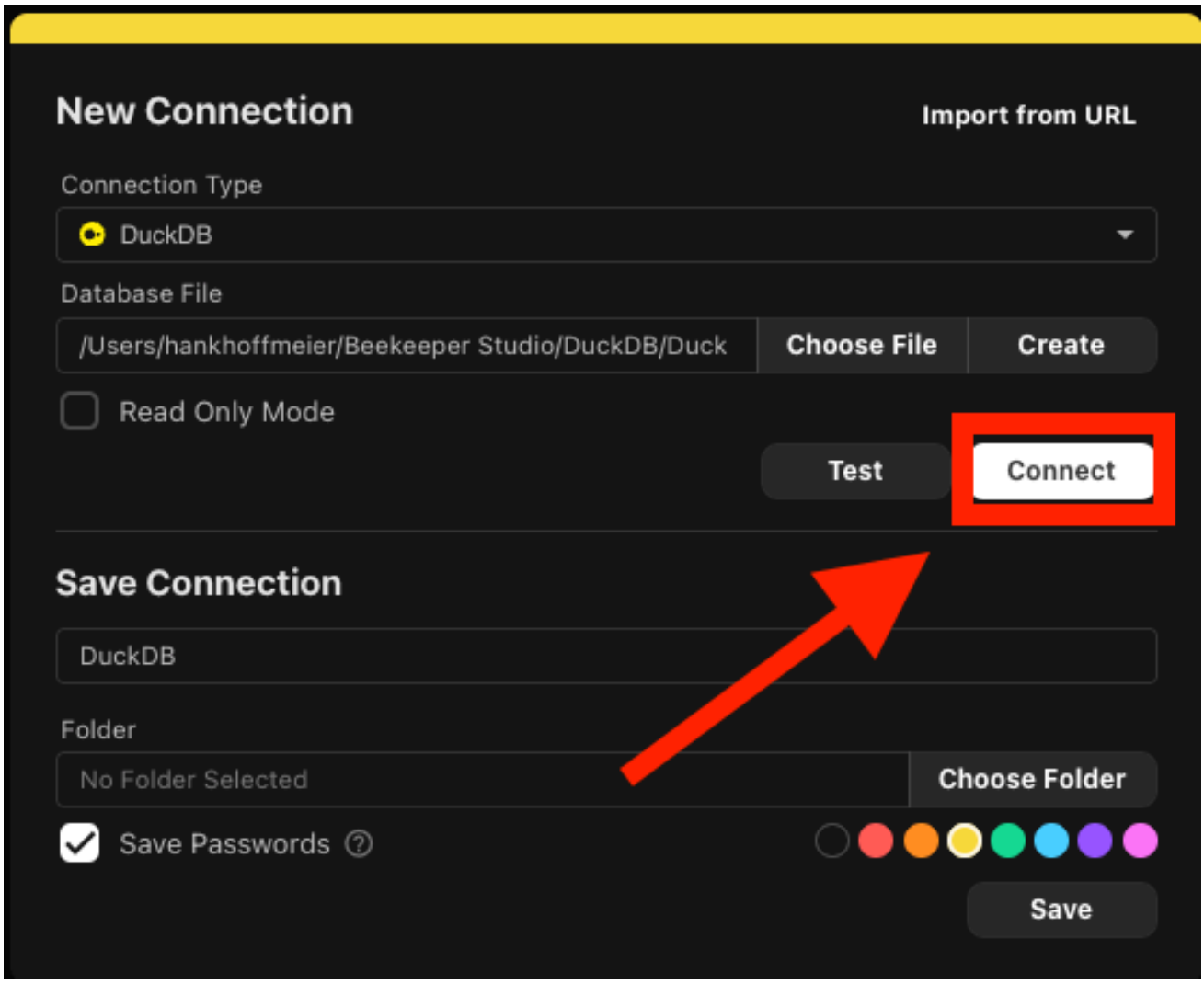


3. Enter the path to your DuckDB database file, or click **Create** to create a new one.

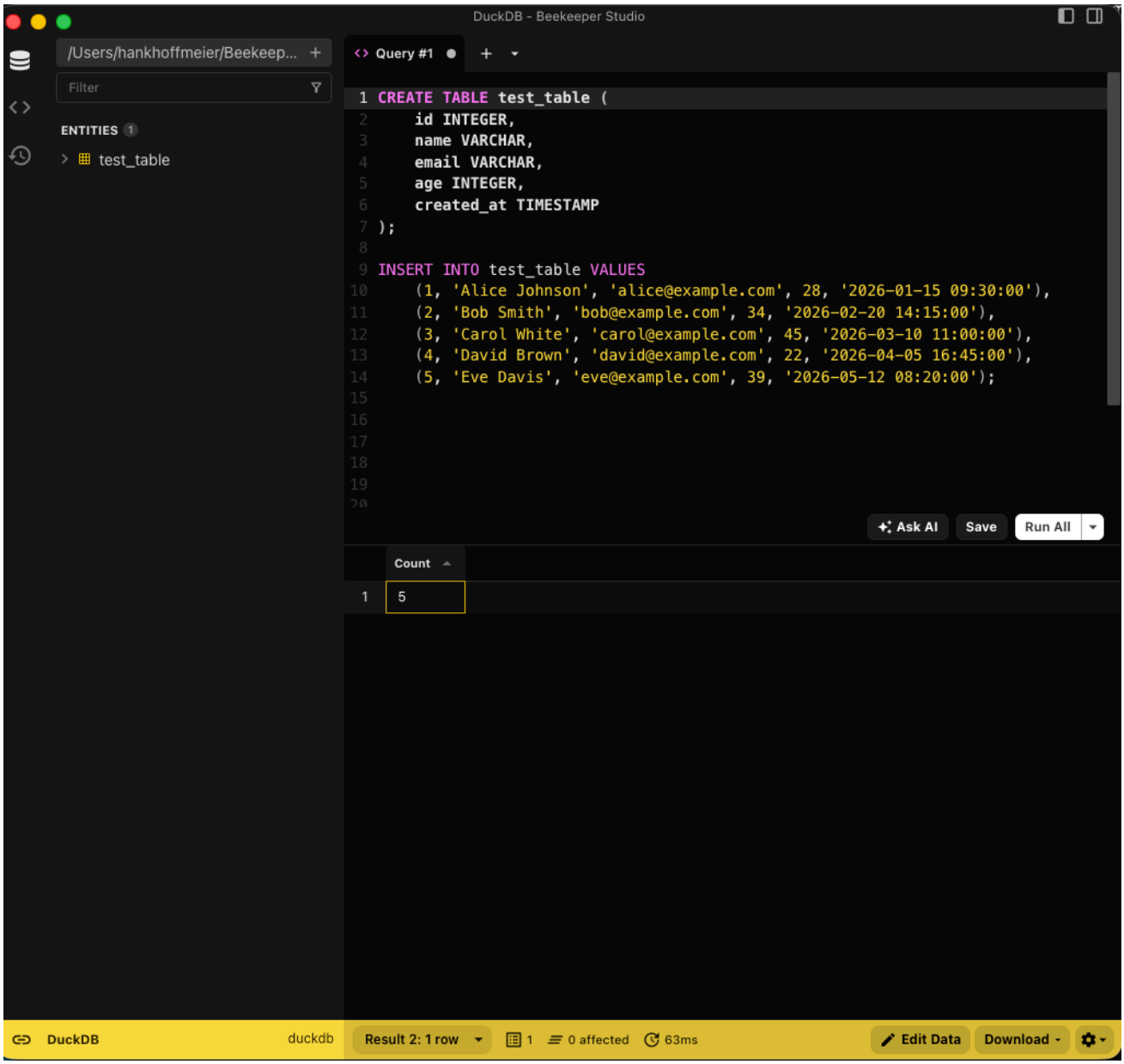


4. Optionally, name the connection and choose a color for the connection.

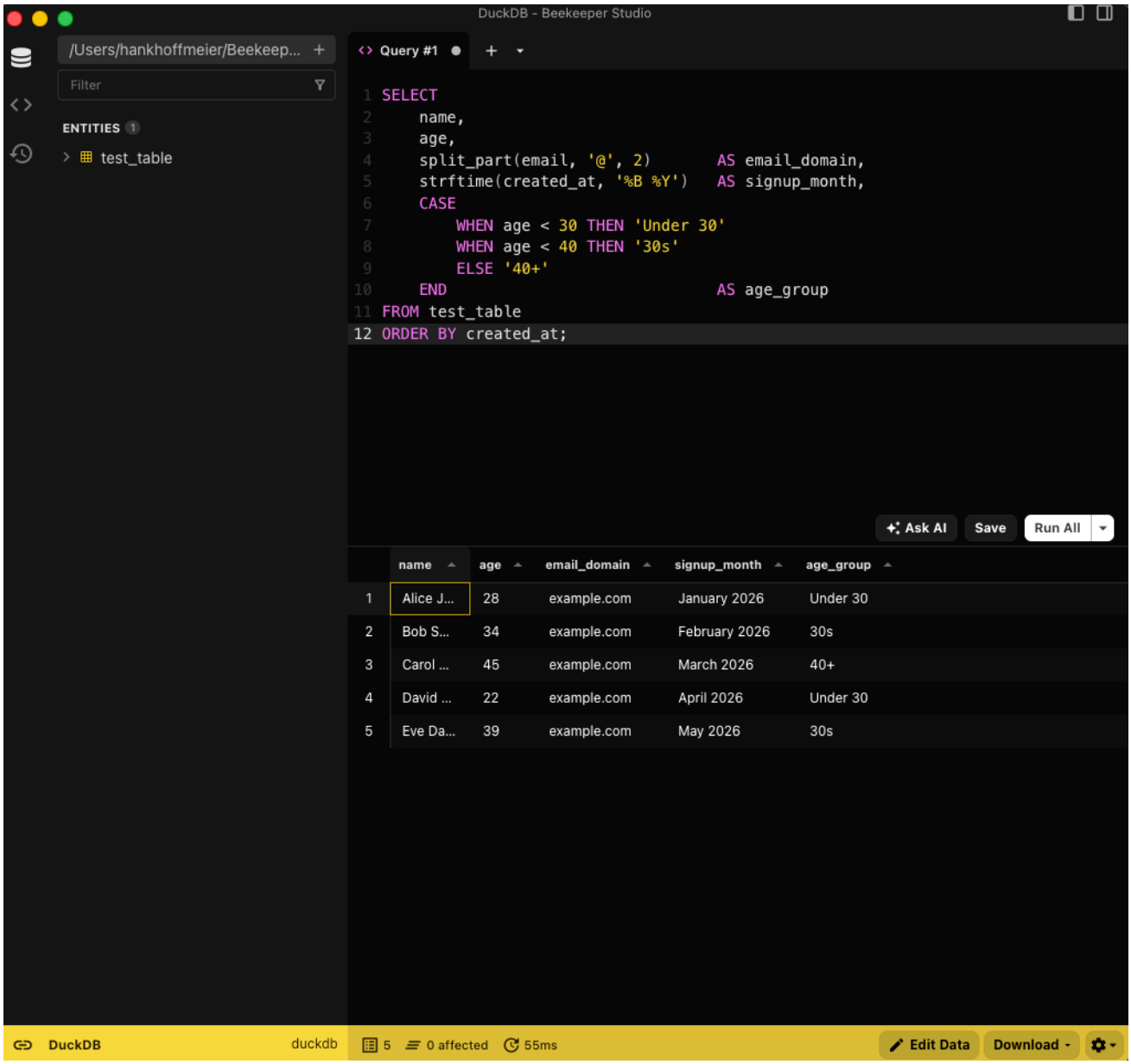
5. Click **Connect**.



6. Once connected, you can browse tables and views in the left-hand sidebar.



7. Open a new SQL tab, write a query, and run it to see the results.



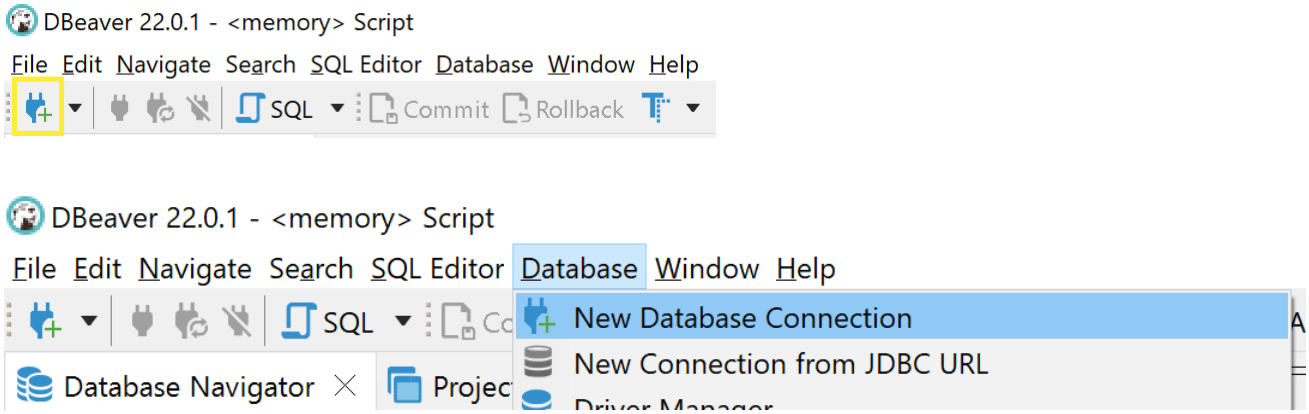
Now you're ready to query DuckDB with Beekeeper Studio.

DBeaver SQL IDE

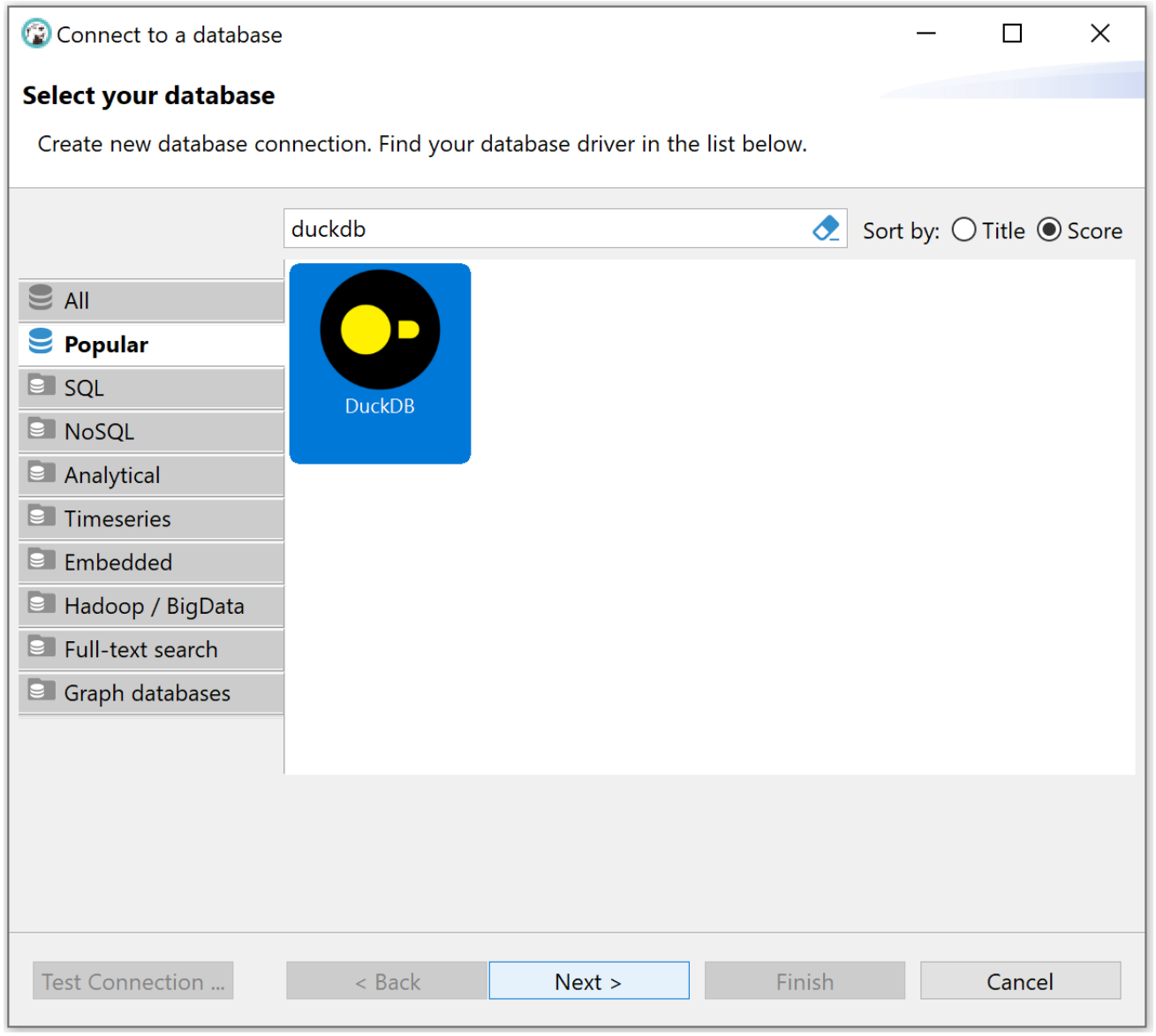
DBeaver is a powerful and popular desktop SQL editor and integrated development environment (IDE). It has both an open source and enterprise version. DBeaver is useful for visually inspecting the available tables in DuckDB and for quickly building complex queries. DuckDB's JDBC connector allows DBeaver to query DuckDB files, and by extension, any other files that DuckDB can access (like Parquet files).

Installing DBeaver

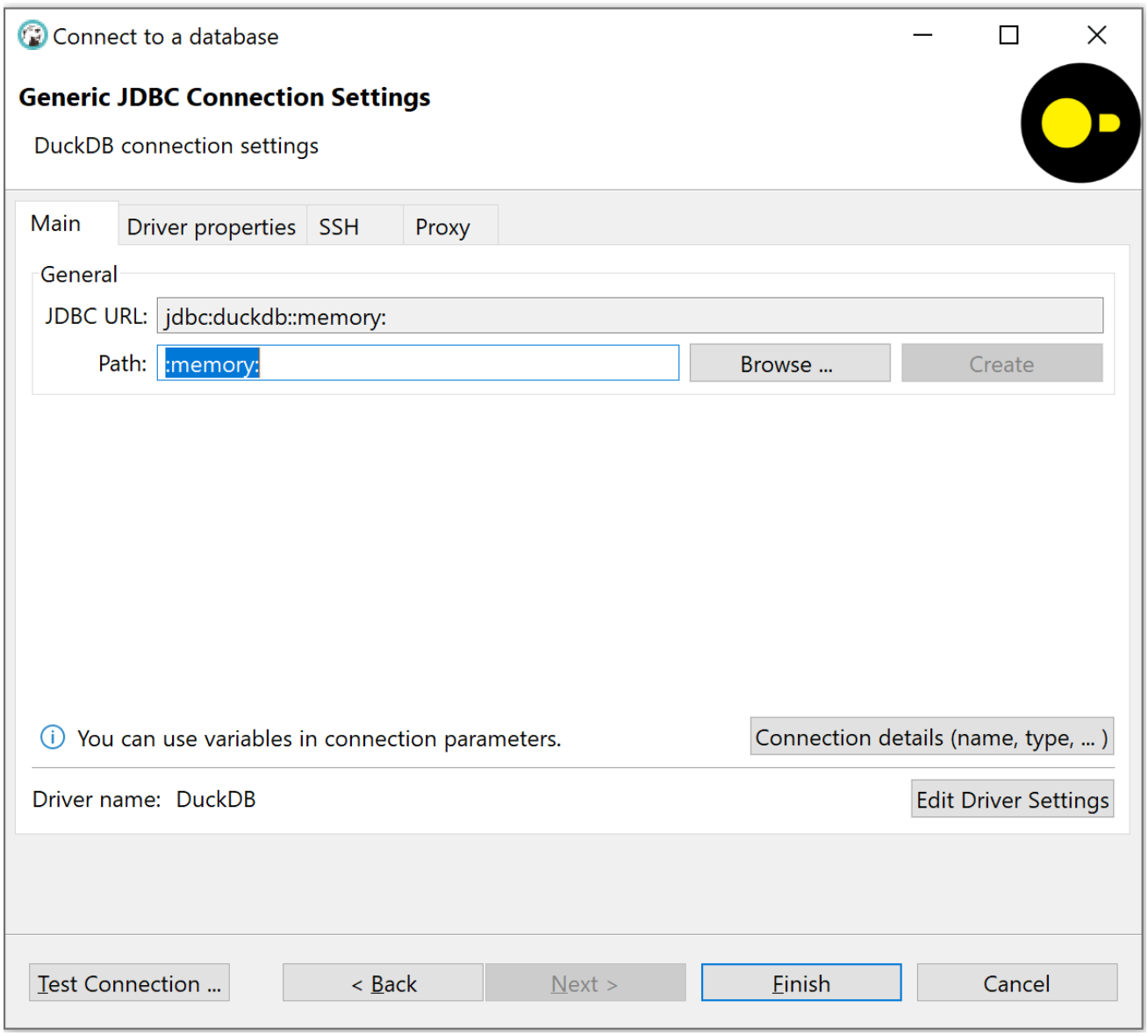
1. Install DBeaver using the download links and instructions found at their download page.
2. Open DBeaver and create a new connection. Either click on the "New Database Connection" button or go to Database > New Database Connection in the menu bar.



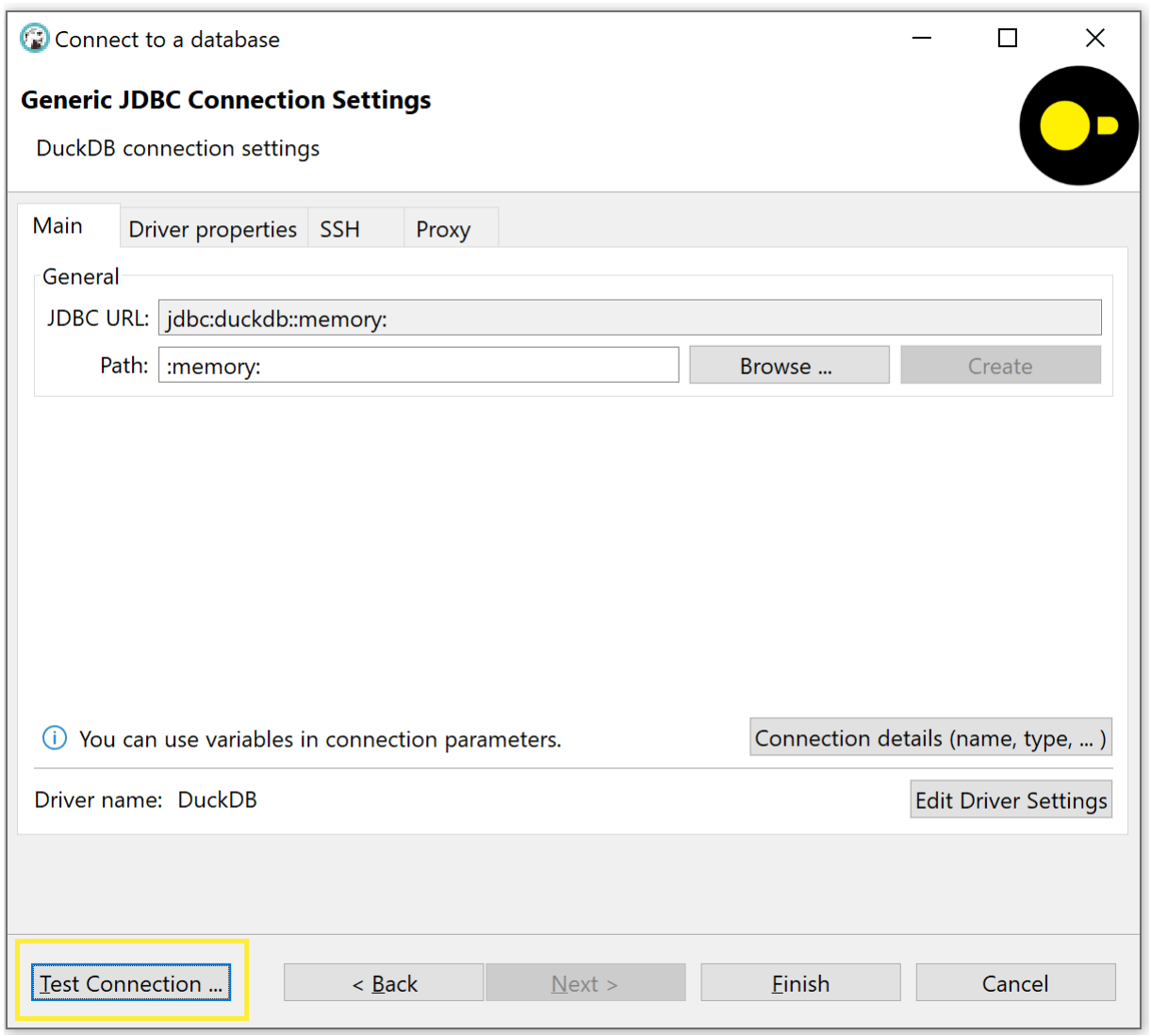
3. Search for DuckDB, select it, and click Next.



4. Enter the path or browse to the DuckDB database file you wish to query. To use an in-memory DuckDB (useful primarily if just interested in querying Parquet files, or for testing) enter `:memory:` as the path.

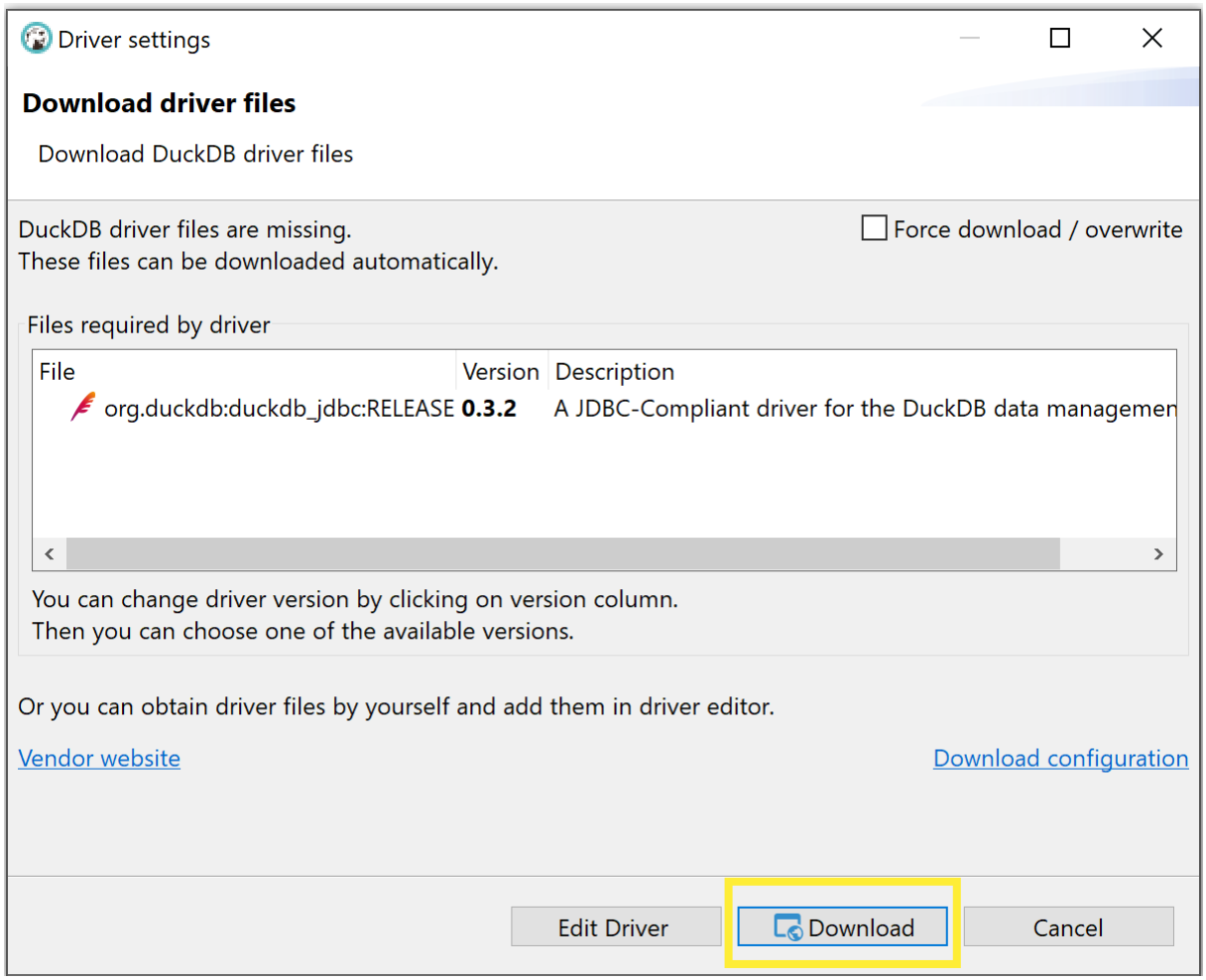


5. Click “Test Connection”. This will then prompt you to install the DuckDB JDBC driver. If you are not prompted, see alternative driver installation instructions below.

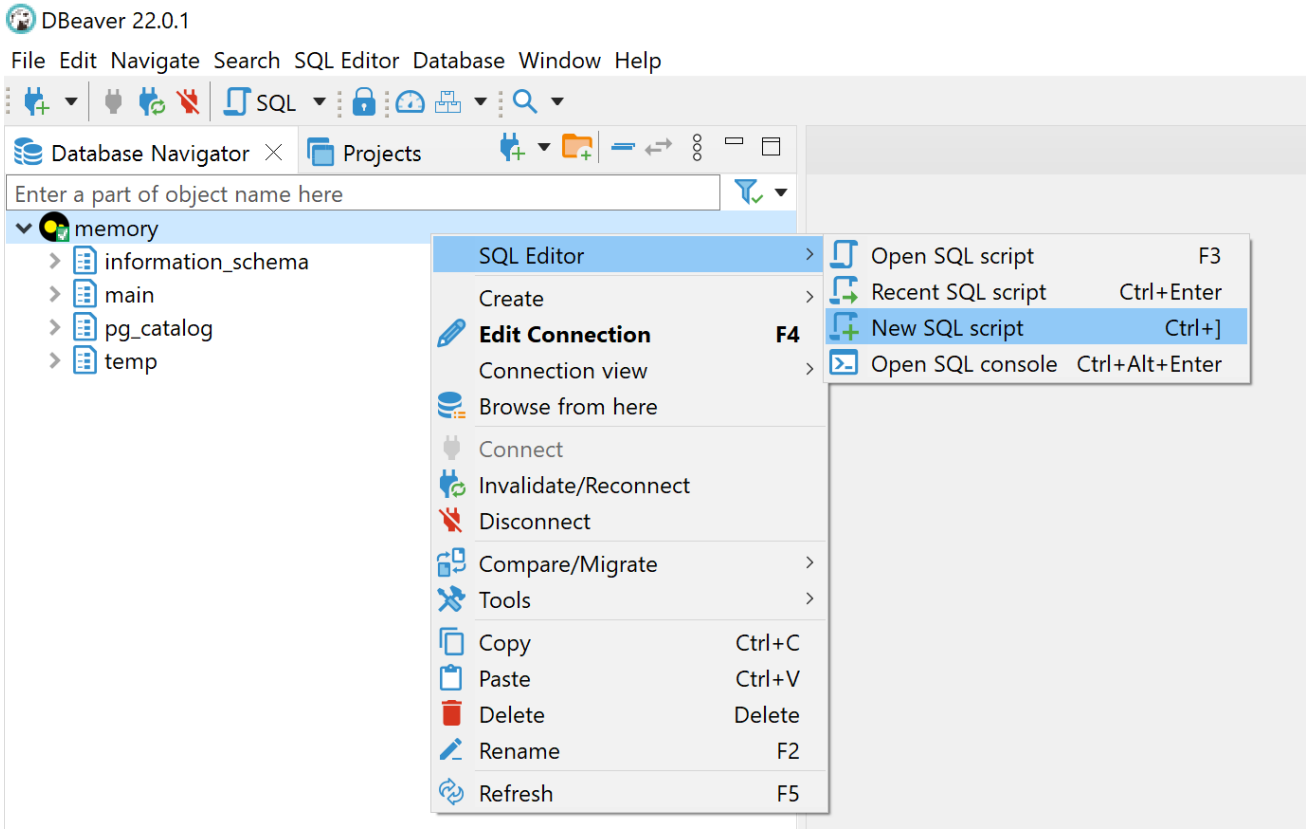


6. Click “Download” to download DuckDB's JDBC driver from Maven. Once download is complete, click “OK”, then click “Finish”.

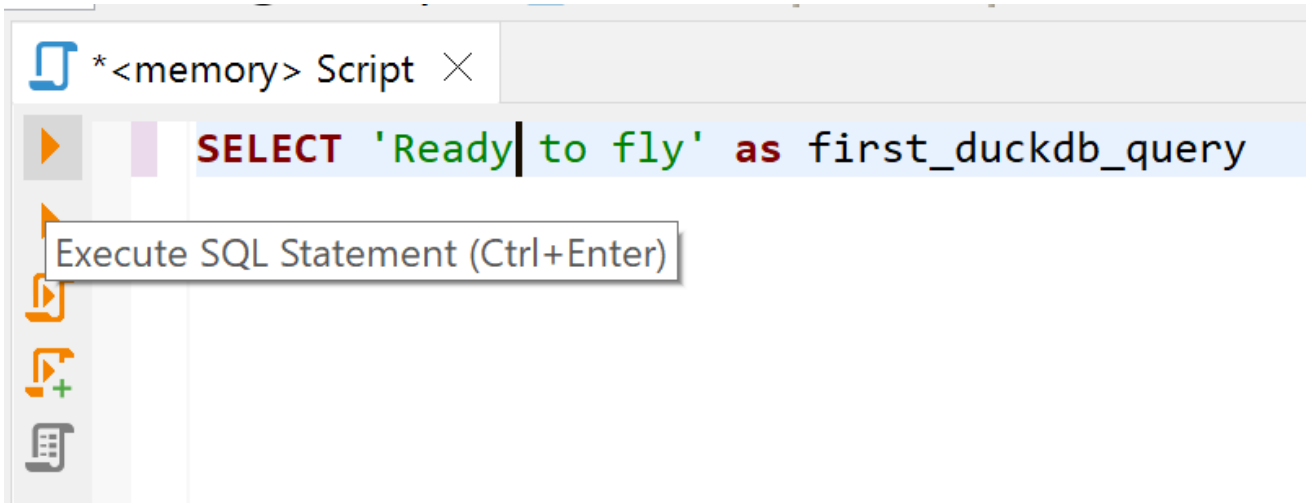
- Note: If you are in a corporate environment or behind a firewall, before clicking download, click the “Download Configuration” link to configure your proxy settings.



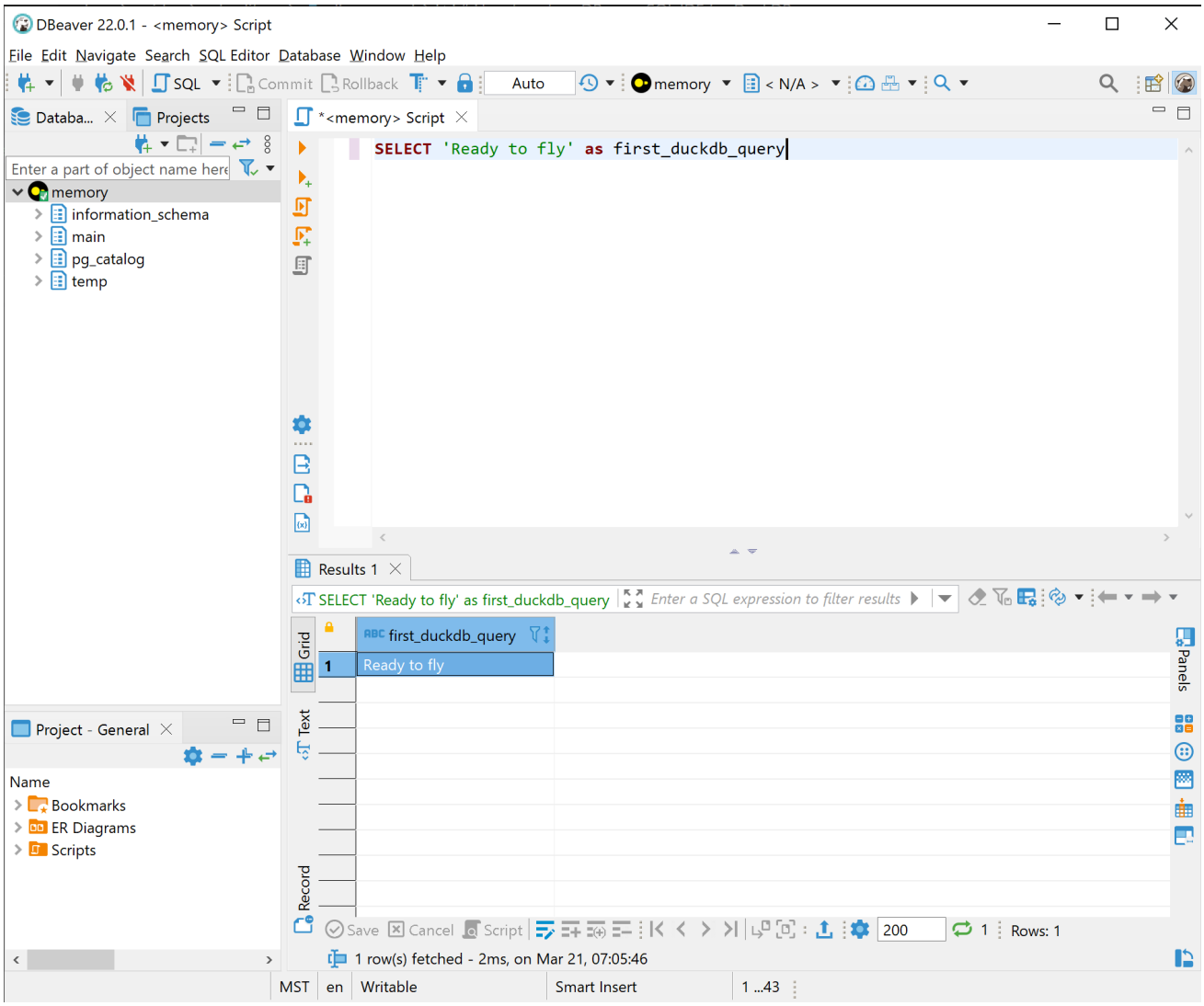
1. You should now see a database connection to your DuckDB database in the left hand “Database Navigator” pane. Expand it to see the tables and views in your database. Right click on that connection and create a new SQL script.



2. Write some SQL and click the “Execute” button.

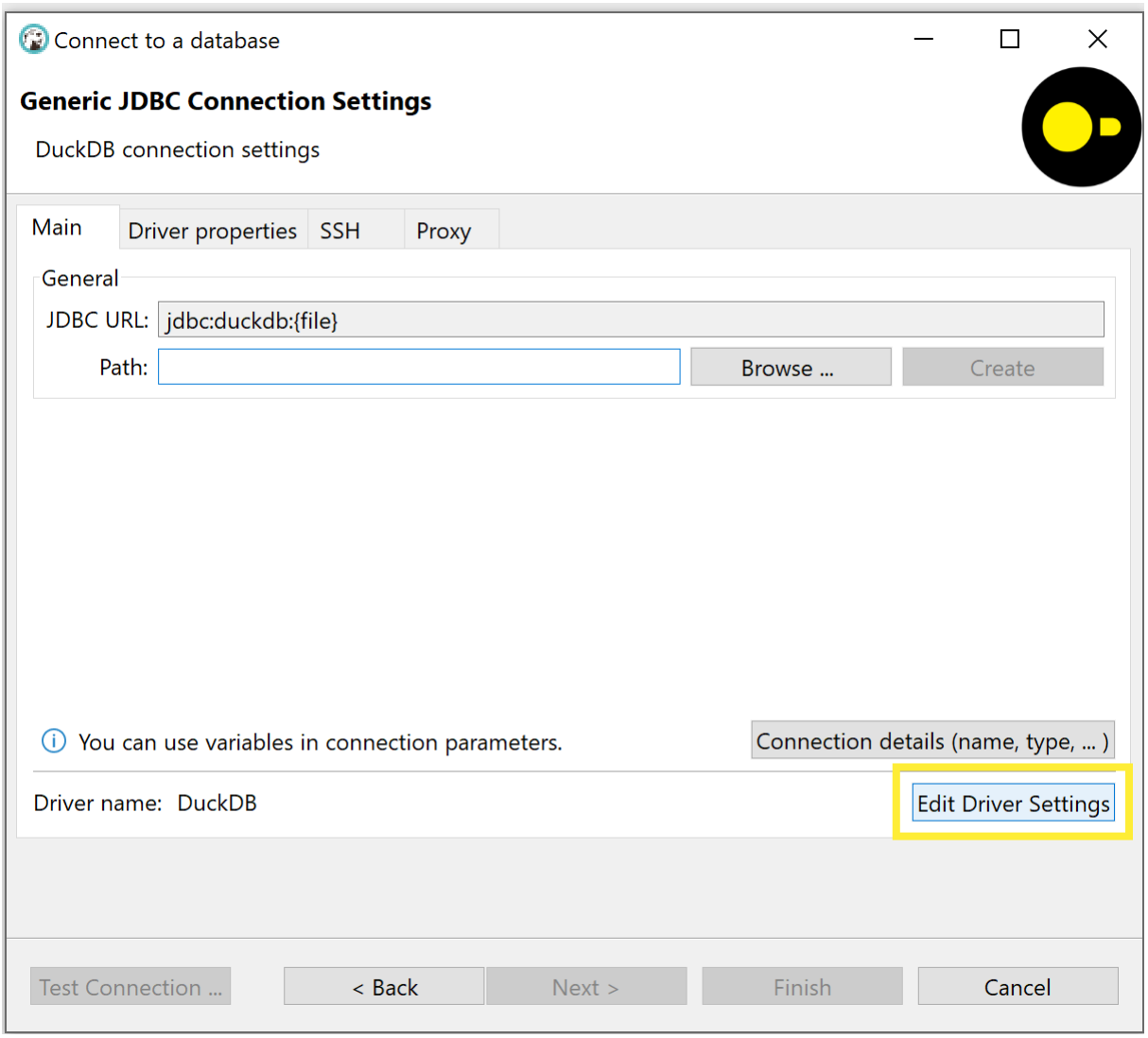


3. Now you're ready to fly with DuckDB and DBeaver!

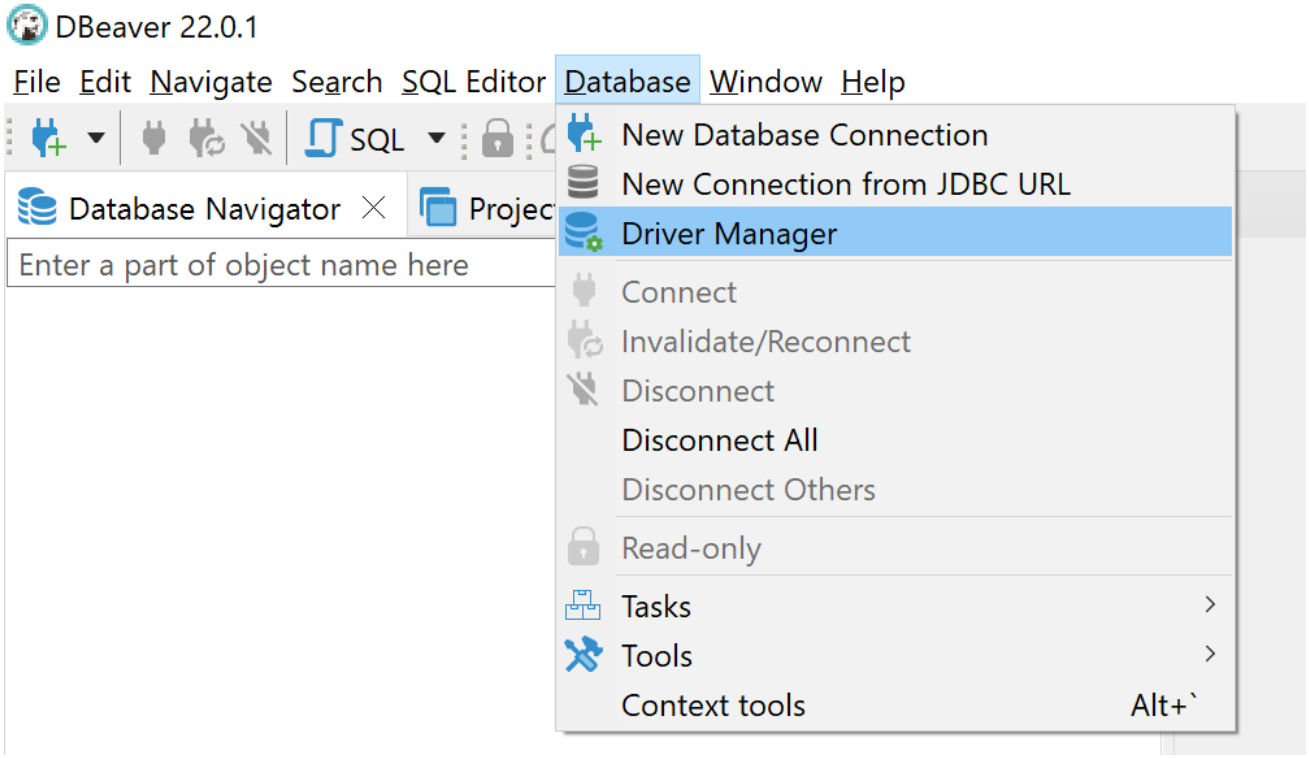


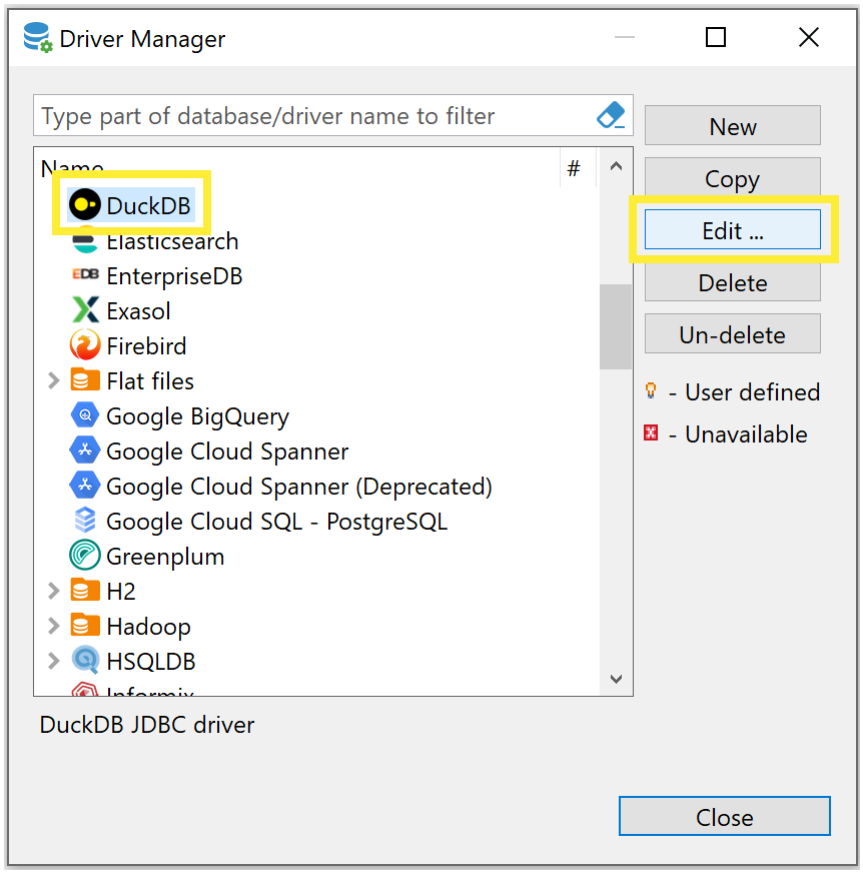
Alternative Driver Installation

1. If not prompted to install the DuckDB driver when testing your connection, return to the “Connect to a database” dialog and click “Edit Driver Settings”.

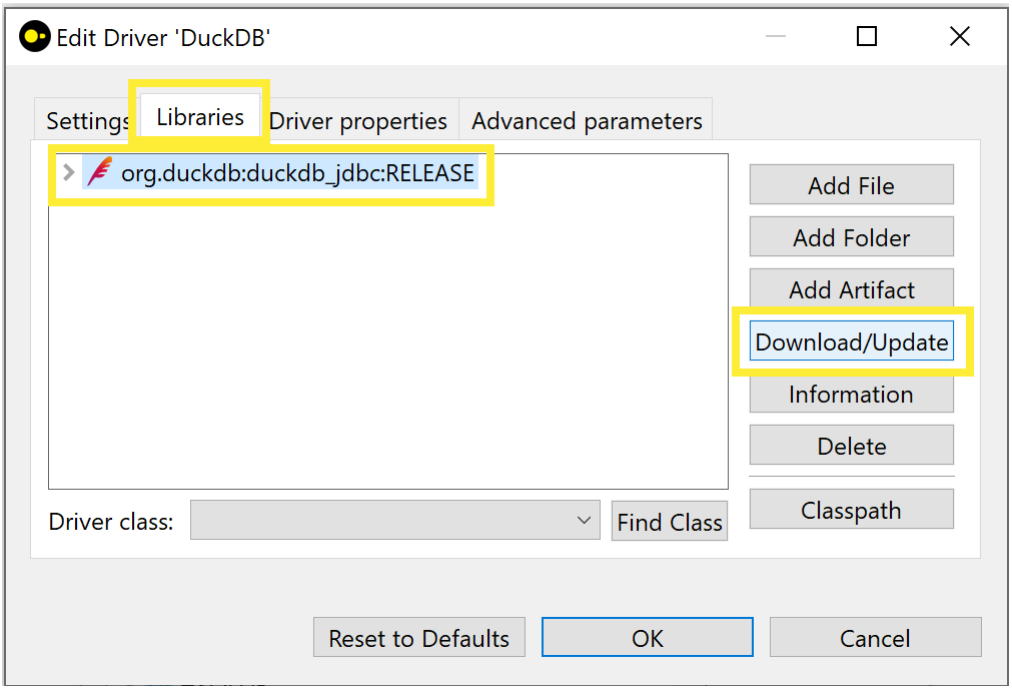


2. Alternatively, you can access the driver settings menu by returning to the main DBeaver window and clicking Database > Driver Manager in the menu bar. Then select DuckDB, then click Edit.



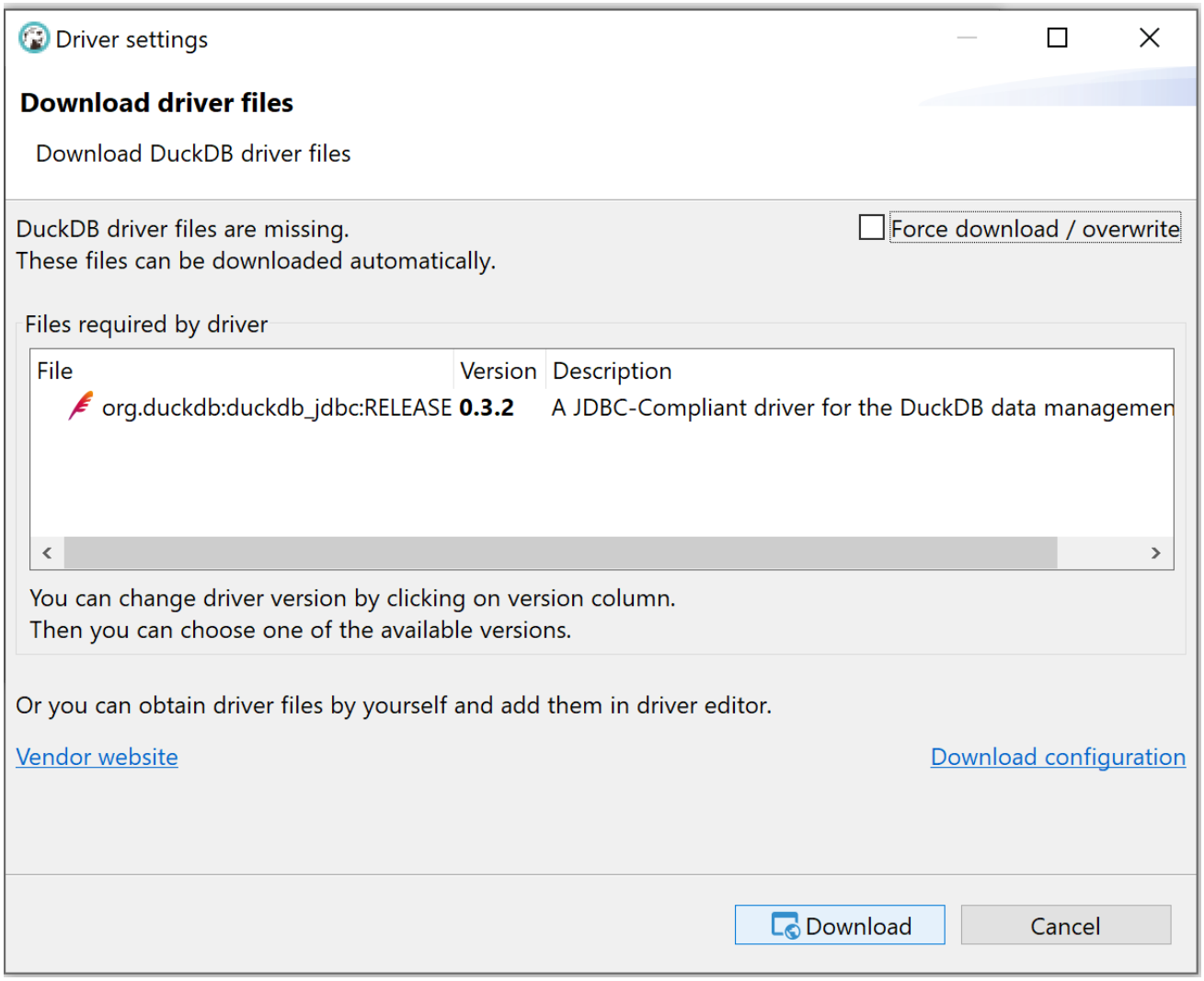


3. Go to the “Libraries” tab, then click on the DuckDB driver and click “Download/Update”. If you do not see the DuckDB driver, first click on “Reset to Defaults”.



4. Click “Download” to download DuckDB's JDBC driver from Maven. Once download is complete, click “OK”, then return to the main DBeaver window and continue with step 7 above.

- Note: If you are in a corporate environment or behind a firewall, before clicking download, click the “Download Configuration” link to configure your proxy settings.



Manual Driver Installation

If DBeaver cannot download the driver itself, for example when it runs without write access to its own directories or without internet access, you can download the JDBC driver manually and point DBeaver at the local file:

1. Open the `org.duckdb:duckdb_jdbc` artifact on Maven Central and download the `.jar` file for the version you want.
2. In DBeaver, go to Database > Driver Manager, select DuckDB, and click "Edit".
3. On the "Libraries" tab, click "Add File" and select the `.jar` file you downloaded. Remove any other entries in the list, or move the downloaded file to the top.
4. Click "Find Class" and confirm that `org.duckdb.DuckDBDriver` is detected, then click "OK".
5. Return to the main DBeaver window and continue with step 7 above.

DBine SQL Editor

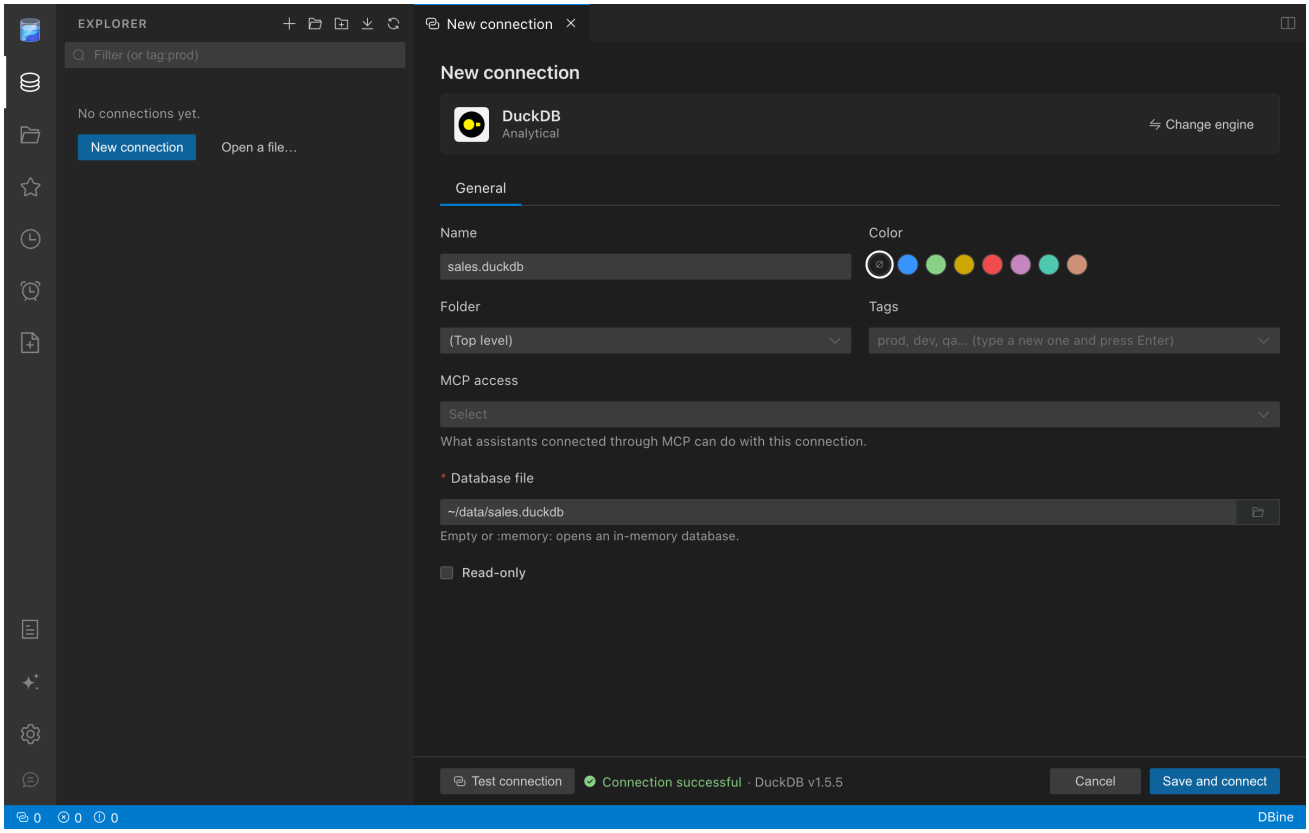
DBine is a free, open-source (Apache 2.0) desktop database manager for Windows, macOS, and Linux that works with 130+ databases and cloud services. Besides a SQL editor, it includes notebooks, charts built from query results, and schema compare. DBine downloads the official DuckDB library the first time a DuckDB connection is opened, so no separate driver installation is required.

Installing DBine

1. Download and install DBine from the releases page.

Connecting to a DuckDB Database

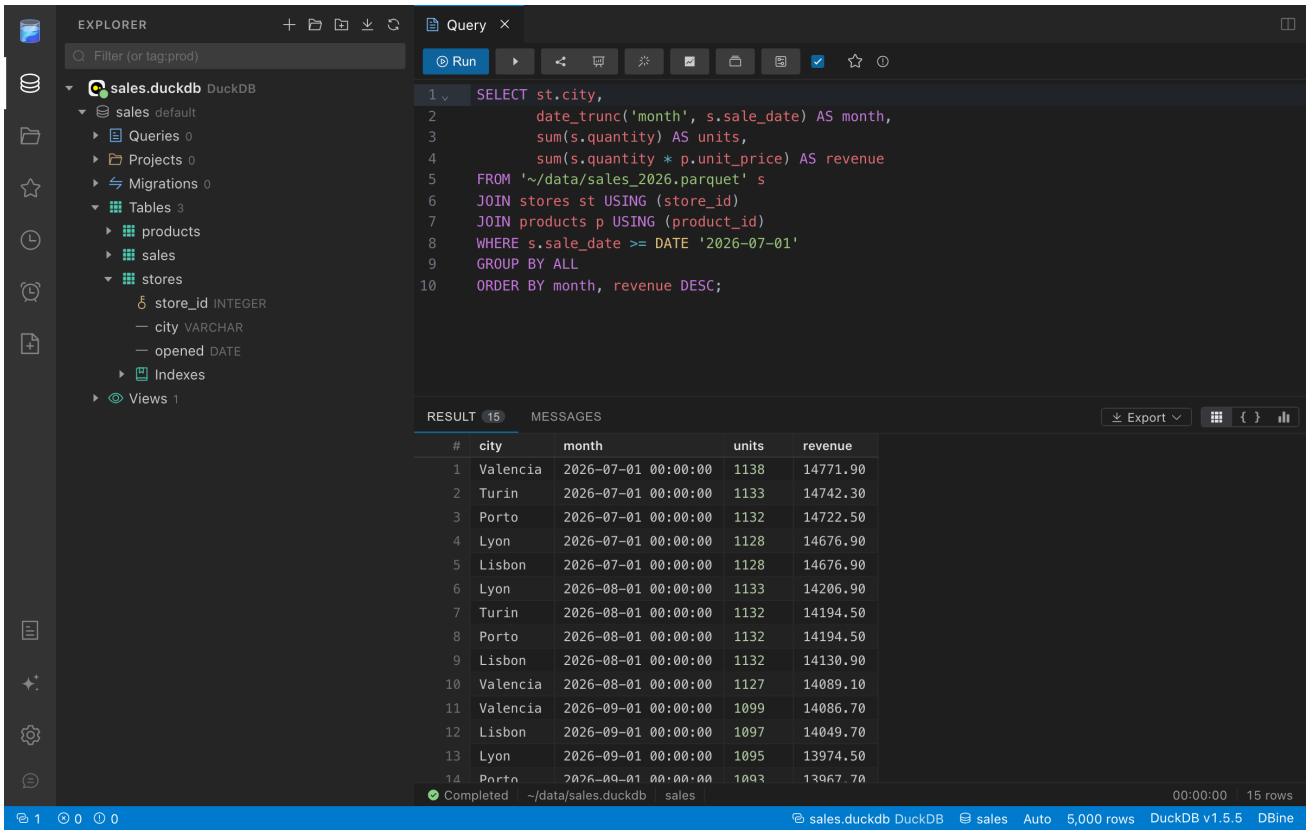
1. Open DBine and click **New connection**, then select **DuckDB** from the list of engines.
2. Give the connection a name and enter the path to your DuckDB database file in **Database file**, or click the folder button to pick it. Leave the field empty or enter `:memory:` to open an in-memory database.
3. Click **Test connection** to confirm the file can be opened, then click **Save and connect**.



> DuckDB allows only one process to open a database file for writing at a time. If another process (for example, the DuckDB CLI) has the file open, close it before connecting.

4. The connection appears in the **Explorer**. Expand it to browse the schemas, tables, views, and columns of the database.
5. Right-click the database and choose **New query**, write a query, and click **Run** to see the results. Parquet, CSV, and JSON files can be queried directly from the editor:

```
SELECT * FROM 'sales_2026.parquet' LIMIT 10;
```



Now you are ready to query DuckDB with DBine.

LibreDB Studio SQL Editor

LibreDB Studio is an open-source, self-hosted, browser-based SQL editor. It runs as a single server, either via `npx`, Docker, or a native package, and is reached through the browser rather than installed as a desktop application. LibreDB Studio connects to DuckDB directly through its native Node.js binding, so it queries a DuckDB database file (or an in-memory database) without going through a JDBC or ODBC driver.

Installing LibreDB Studio

1. Start LibreDB Studio with `npx`, which downloads and runs the latest release:

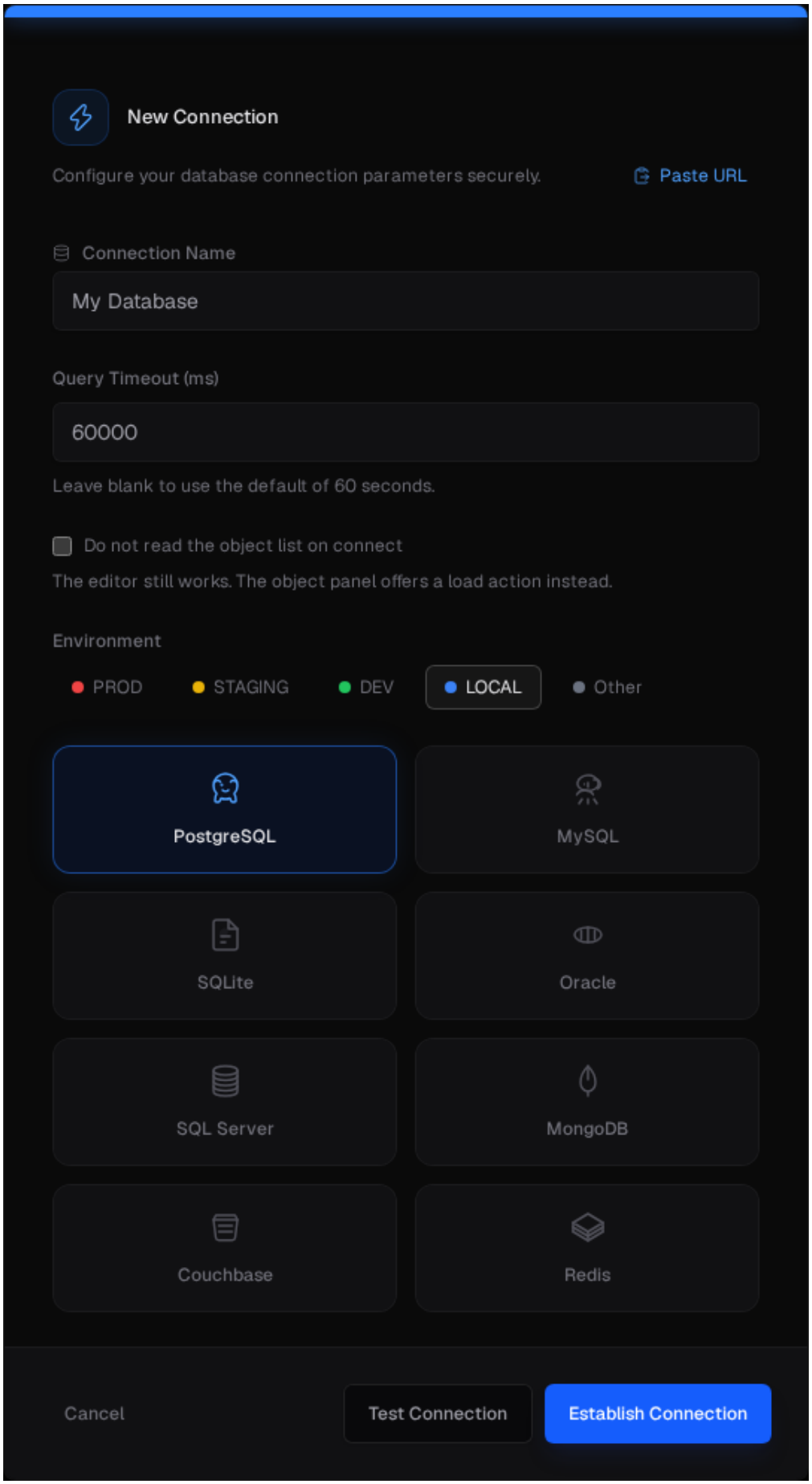
   ```
   npx @libredb/studio
   ```

   Docker, Homebrew, and other install options are listed in the project's README.

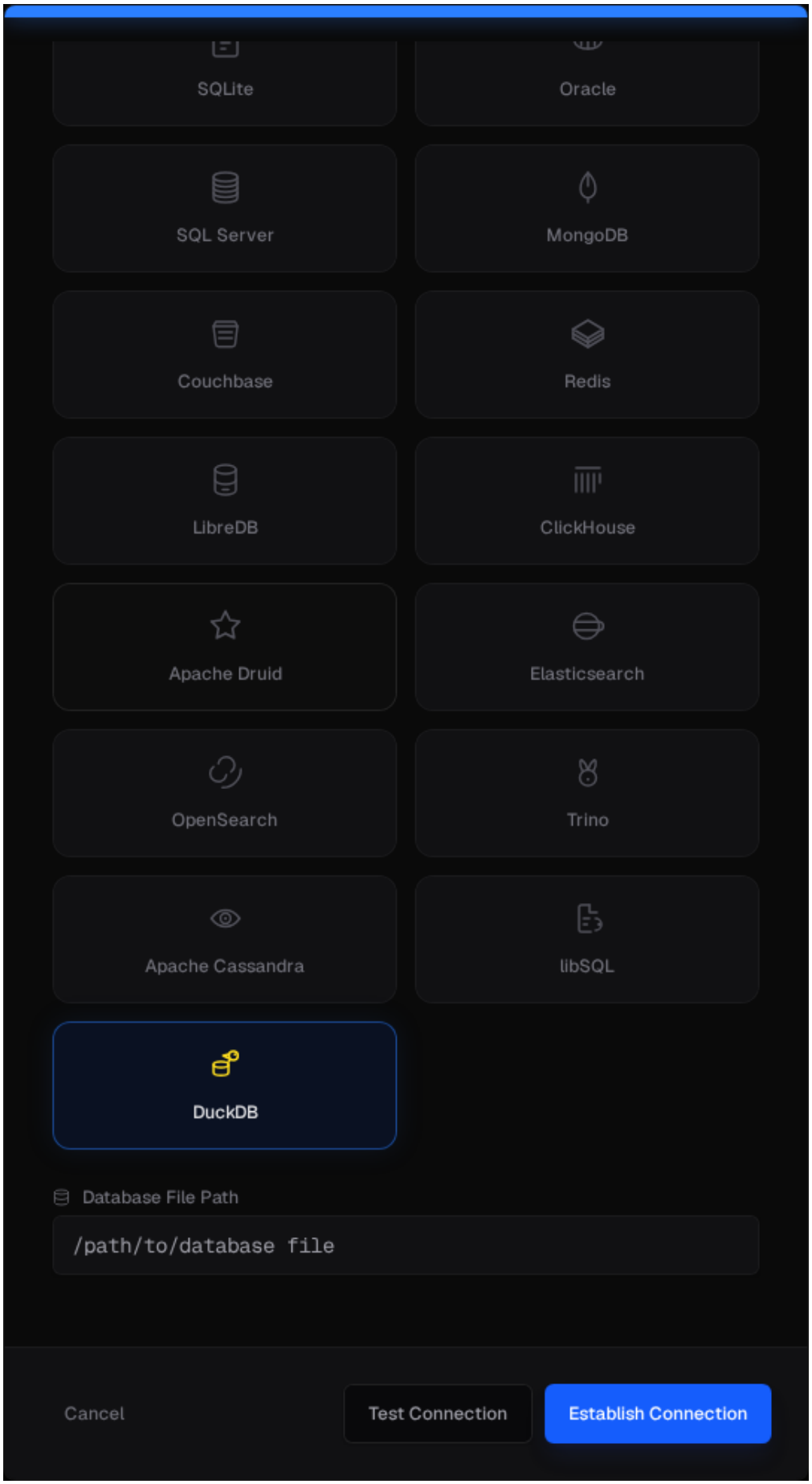
2. Open `http://localhost:3000` in a browser. On first run, the admin password is printed to the console.

Connecting to a DuckDB Database

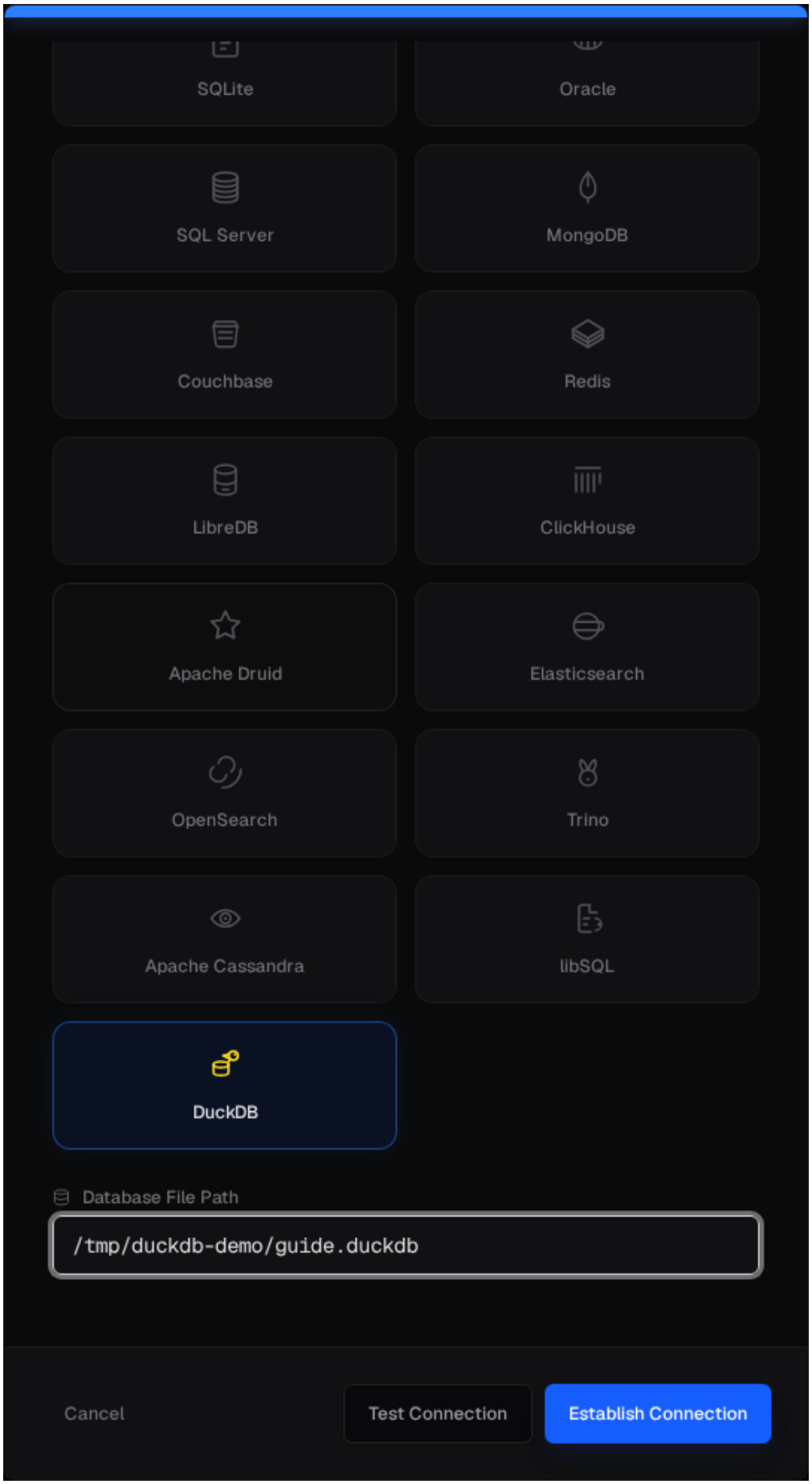
1. Click the **+** button next to the LibreDB Studio logo in the sidebar to open the New Connection dialog.


New Connection
Configure your database connection parameters securely.
Paste URL
Connection Name
My Database
Query Timeout (ms)
60000
Leave blank to use the default of 60 seconds.
Do not read the object list on connect
The editor still works. The object panel offers a load action instead.
Environment
PROD
STAGING
DEV
LOCAL
Other
PostgreSQL
MySQL
SQLite
Oracle
SQL Server
MongoDB
Couchbase
Redis
Cancel
Test Connection
Establish Connection

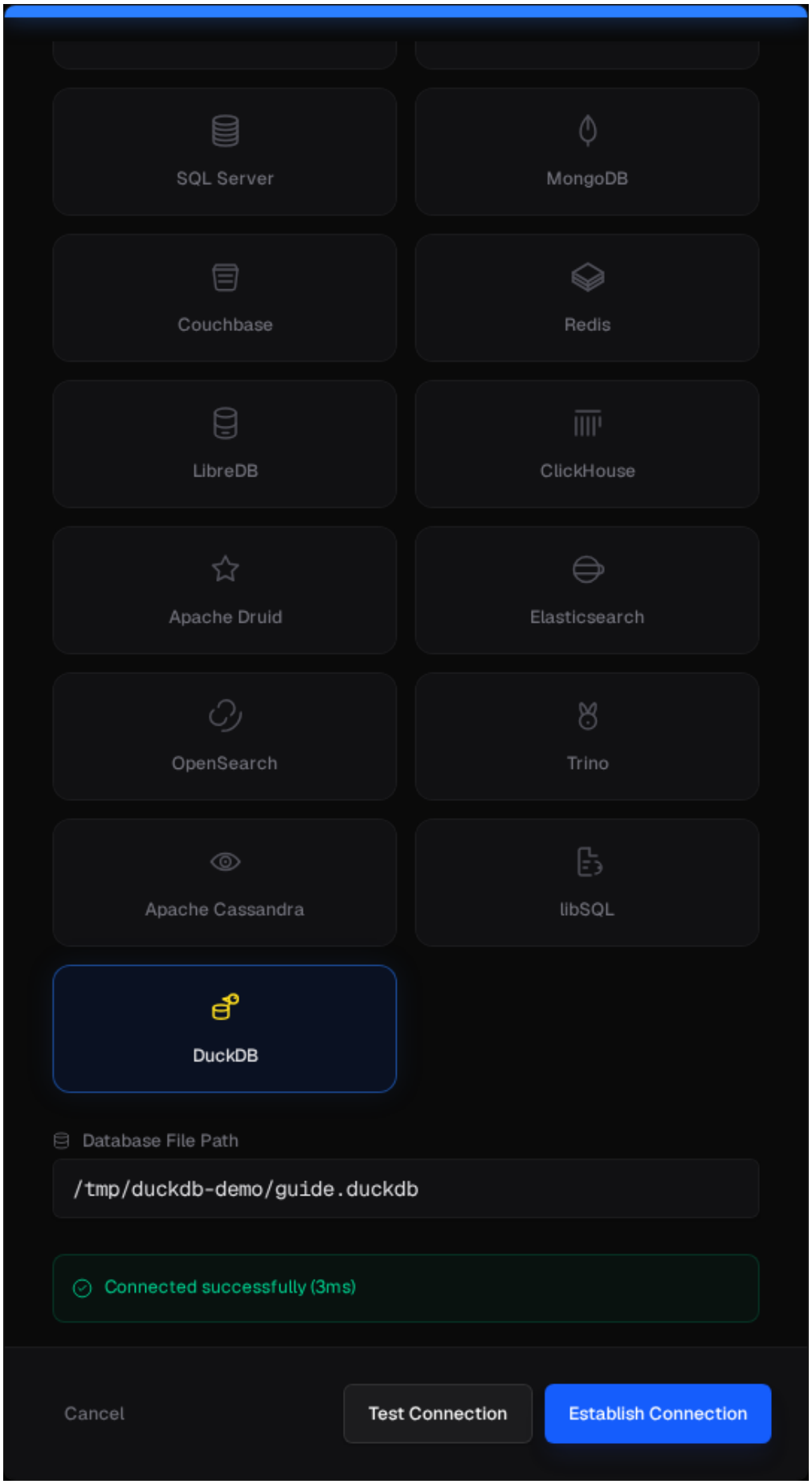
2. Select **DuckDB** from the database type grid, and give the connection a name.


SQLite
Oracle
SQL Server
MongoDB
Couchbase
Redis
LibreDB
ClickHouse
Apache Druid
Elasticsearch
OpenSearch
Trino
Apache Cassandra
libSQL
DuckDB
Database File Path
/path/to/database file
Cancel
Test Connection
Establish Connection

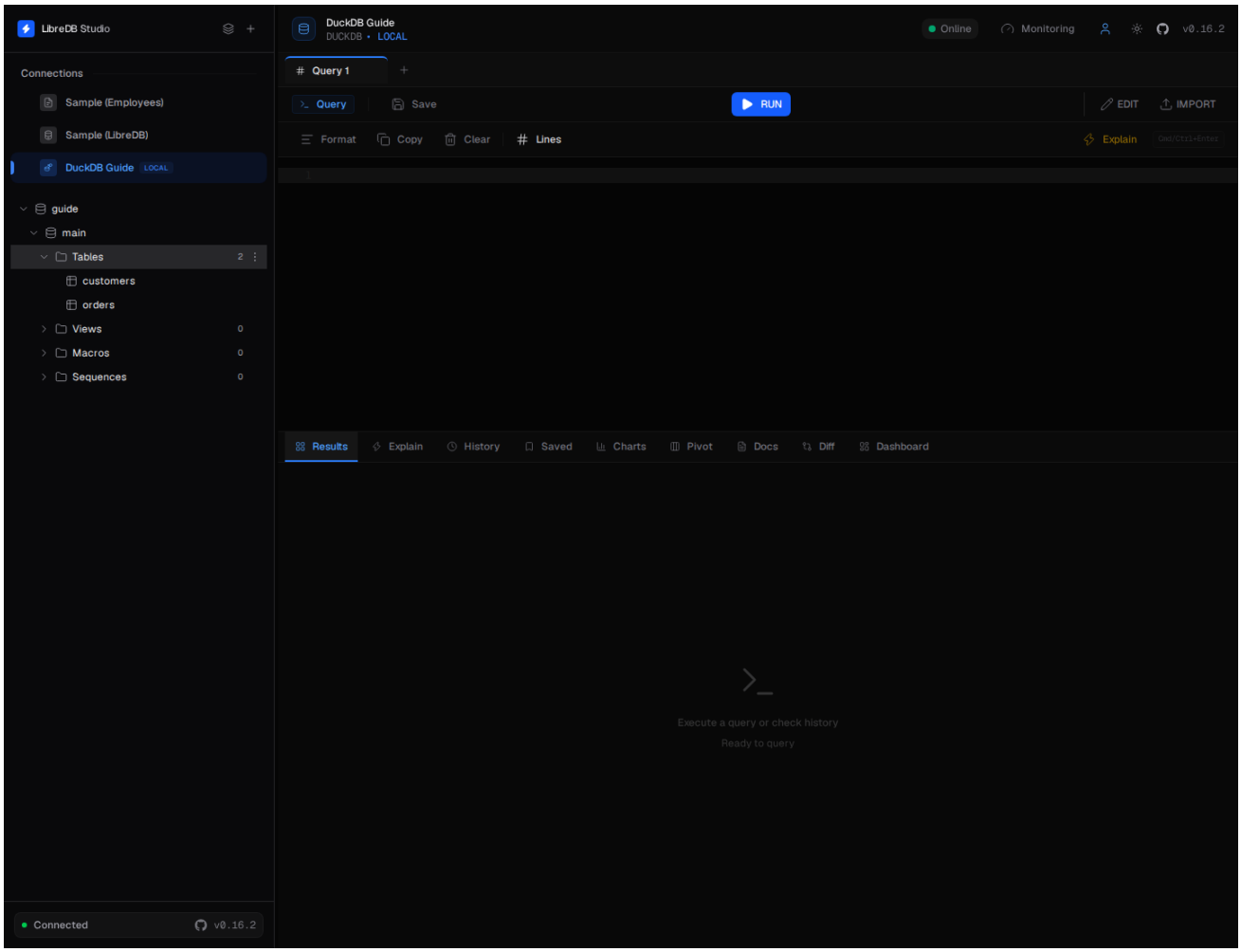
3. Enter the path to the DuckDB database file in **Database File Path**. To use an in-memory database, enter `:memory:`. Since LibreDB Studio runs as a server, the path is resolved on the machine running the server, not on the machine running the browser.


SQLite
Oracle
SQL Server
MongoDB
Couchbase
Redis
LibreDB
ClickHouse
Apache Druid
Elasticsearch
OpenSearch
Trino
Apache Cassandra
libSQL
DuckDB
Database File Path
/tmp/duckdb-demo/guide.duckdb
Cancel
Test Connection
Establish Connection

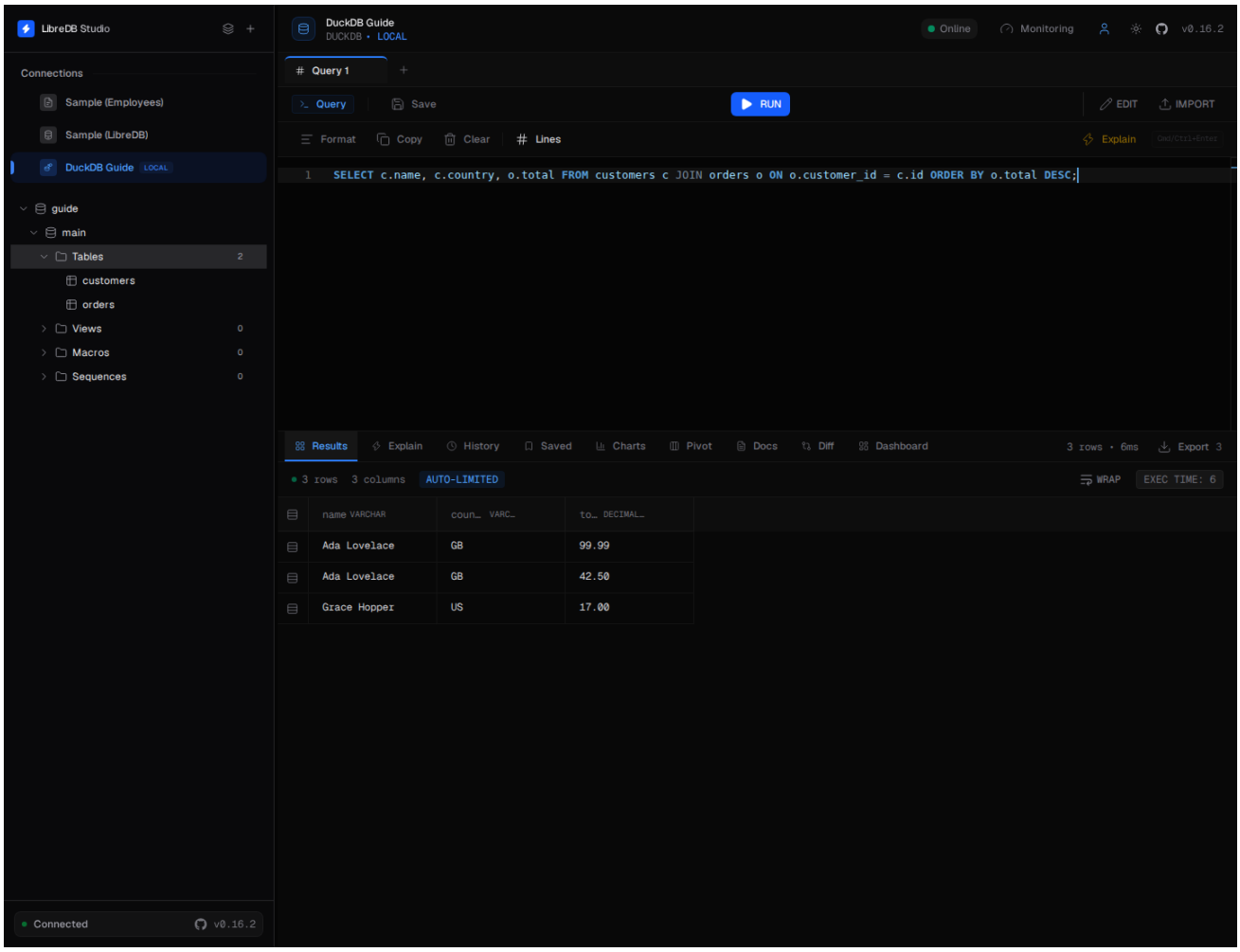
4. Click **Test Connection** to confirm the file can be opened, then click **Establish Connection** to save it.


SQL Server
MongoDB
Couchbase
Redis
LibreDB
ClickHouse
Apache Druid
Elasticsearch
OpenSearch
Trino
Apache Cassandra
libSQL
DuckDB
Database File Path
/tmp/duckdb-demo/guide.duckdb
Connected successfully (3ms)
Cancel
Test Connection
Establish Connection

5. The connection appears in the sidebar. Expand it to browse the tables, views, macros, and sequences in the database.



6. Open a query tab, write SQL, and run it with **Run** or Ctrl+Enter (Cmd+Enter on macOS).



Now you are ready to query DuckDB with LibreDB Studio.

VisuaLeaf SQL Editor

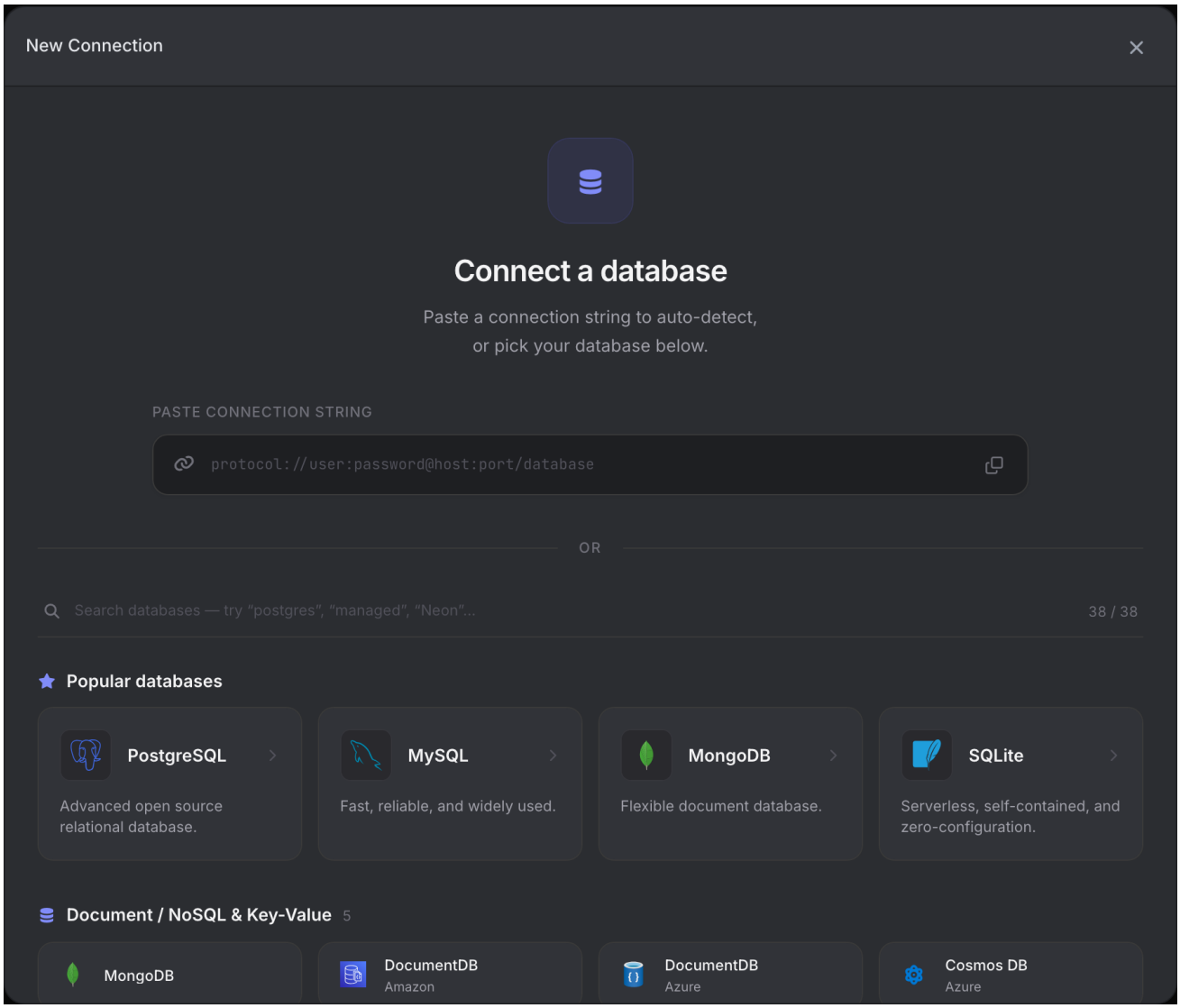
VisuaLeaf is a cross-platform desktop database client for Windows, macOS, and Linux. Besides a SQL editor, it includes a visual query builder, entity-relationship diagrams, and charts built from query results. VisuaLeaf ships with the DuckDB JDBC driver, so no separate driver installation is required. It offers a free Community Edition as well as paid editions with additional features.

Installing VisuaLeaf

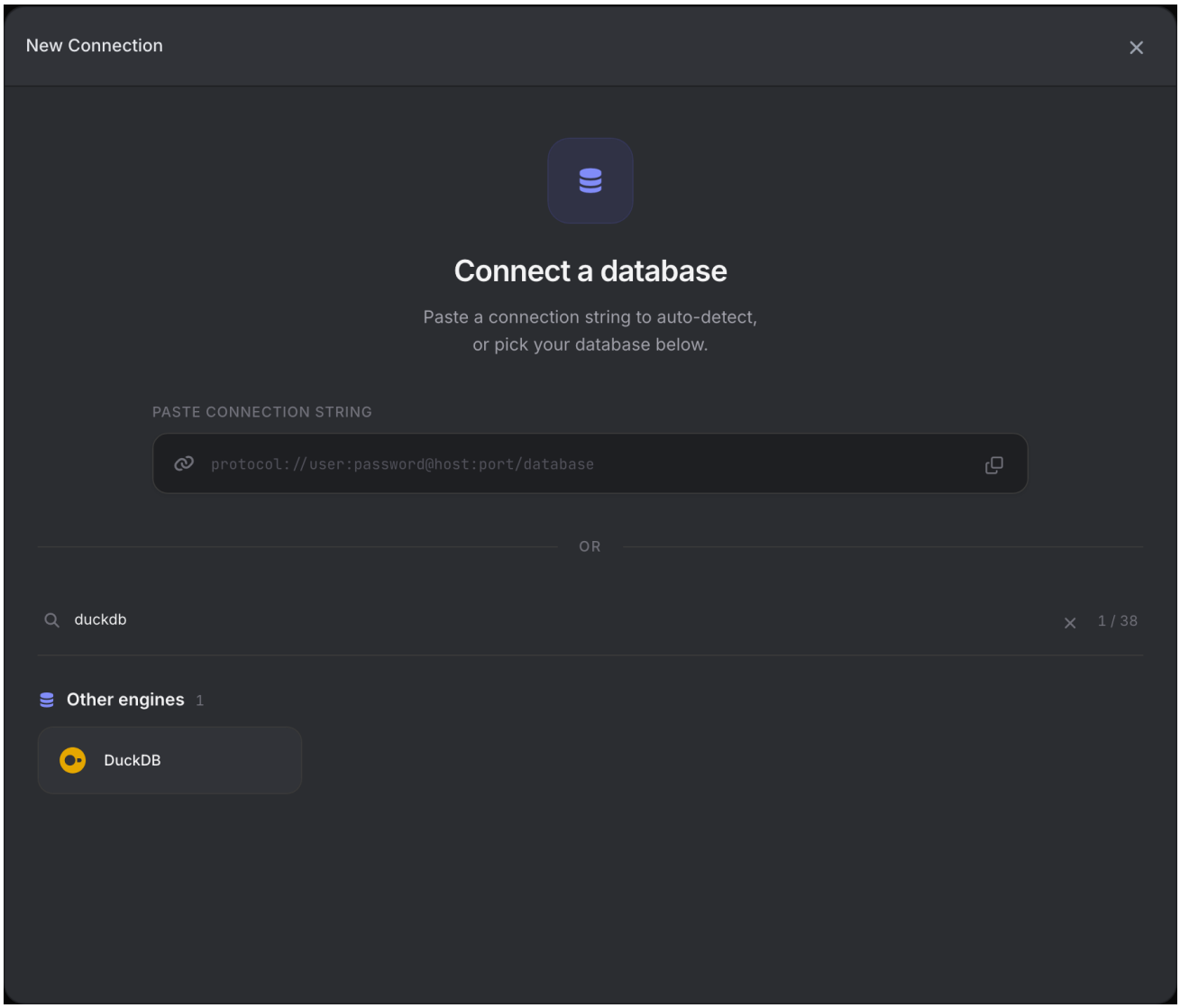
1. Download and install VisuaLeaf from the download page.

Connecting to a DuckDB Database

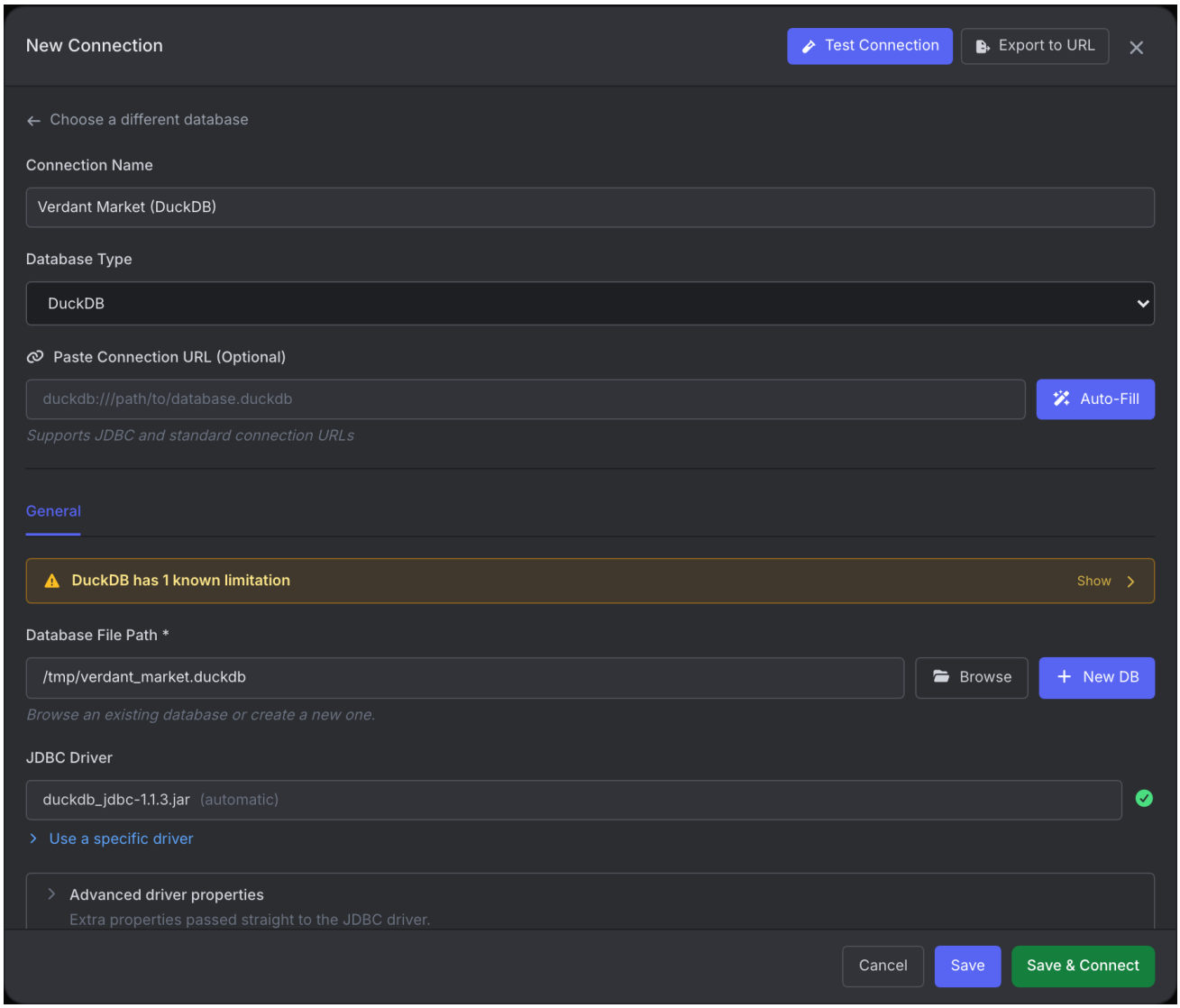
1. Open VisuaLeaf and click **New Connection** on the welcome screen.



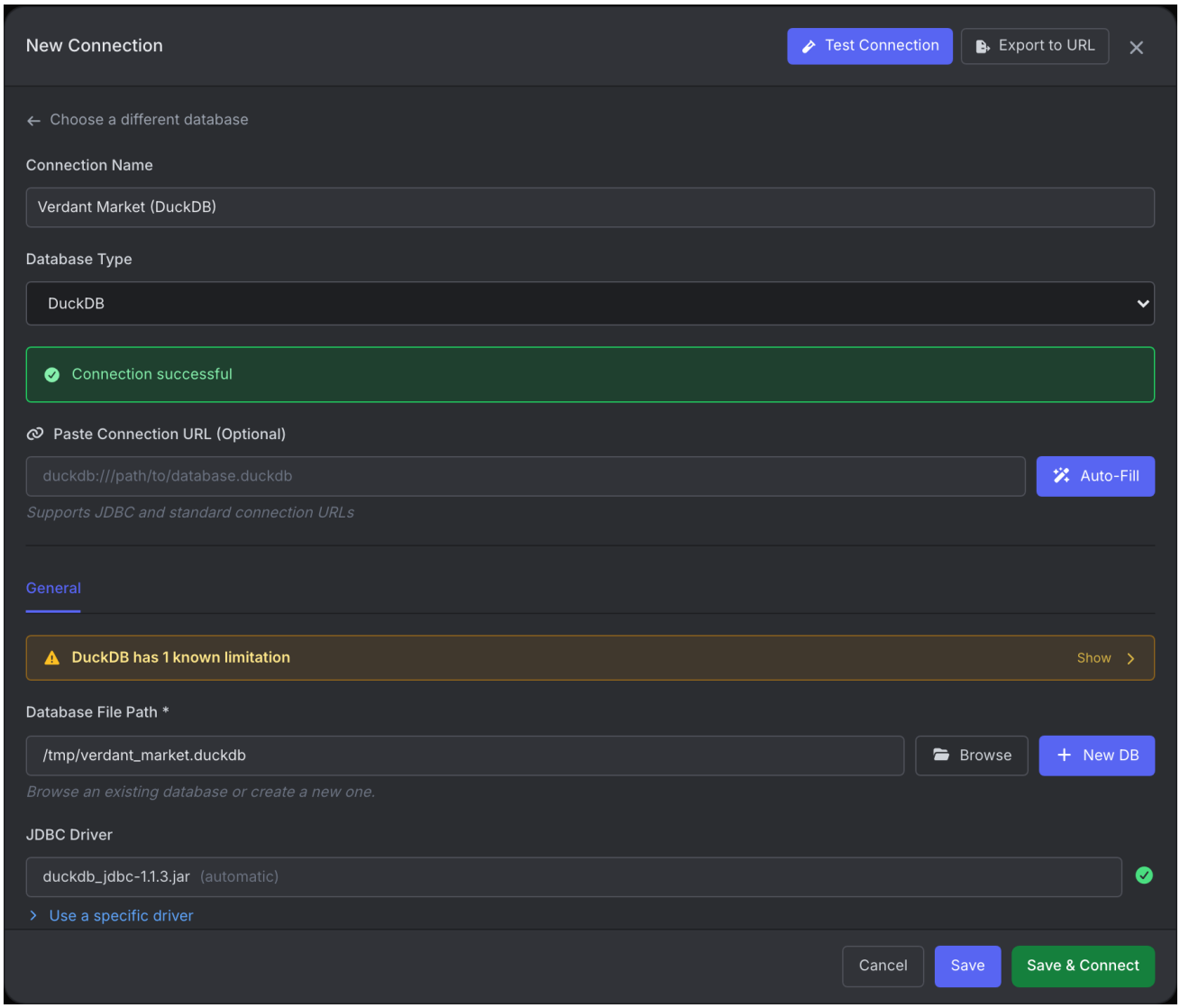
2. Type duckdb in the search box and select **DuckDB**.



3. Give the connection a name and enter the path to your DuckDB database file in **Database File Path**, or click **Browse** to pick it. To create a new database file instead, click **New DB**.

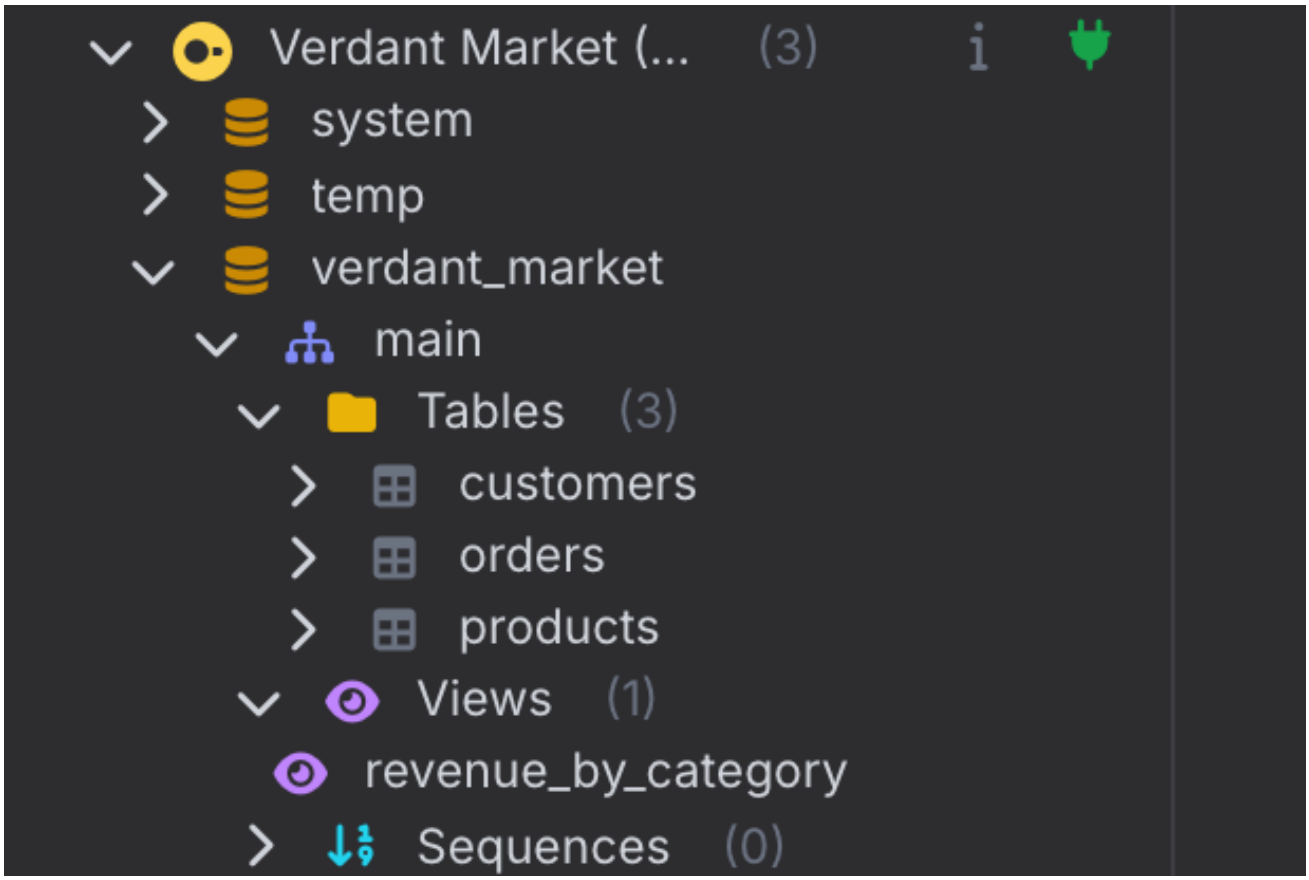


4. Click **Test Connection** to confirm the file can be opened, then click **Save & Connect**.

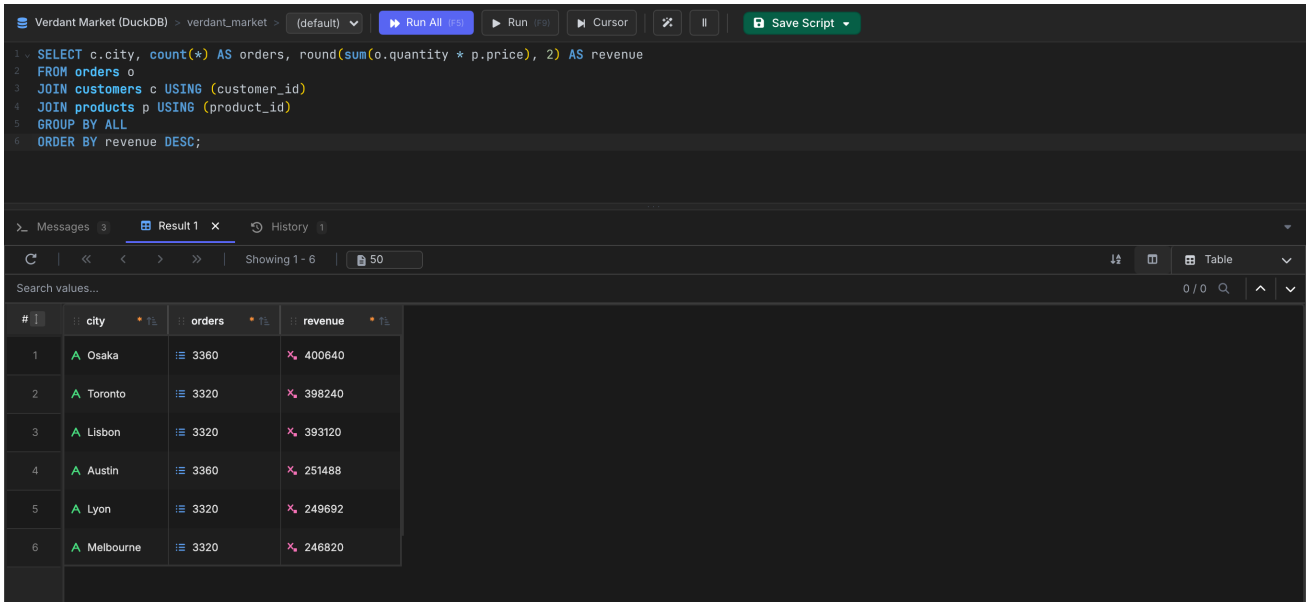


> DuckDB allows only one process to open a database file for writing at a time. If another process (for example, the DuckDB CLI) has the file open, close it before connecting.

5. The connection appears in the **Connections** panel. Expand it to browse the schemas, tables, views, and sequences in the database.



6. Right-click the database and choose **New SQL Editor**, write a query, and click **Run All** to see the results.



Now you are ready to query DuckDB with VisuaLeaf.

SQL Features

AsOf Join

What is an AsOf Join?

Time series data is not always perfectly aligned. Clocks may be slightly off, or there may be a delay between cause and effect. This can make connecting two sets of ordered data challenging. AsOf joins are a tool for solving this and other similar problems.

One of the problems that AsOf joins are used to solve is finding the value of a varying property at a specific point in time. This use case is so common that it is where the name came from:

*Give me the value of the property **as of this time**.*

More generally, however, AsOf joins embody some common temporal analytic semantics, which can be cumbersome and slow to implement in standard SQL.

Portfolio Example Dataset

Let's start with a concrete example. Suppose we have a table of stock `prices` with timestamps:

ticker	when	price
APPL	2001-01-01 00:00:00	1
APPL	2001-01-01 00:01:00	2
APPL	2001-01-01 00:02:00	3
MSFT	2001-01-01 00:00:00	1
MSFT	2001-01-01 00:01:00	2
MSFT	2001-01-01 00:02:00	3
GOOG	2001-01-01 00:00:00	1
GOOG	2001-01-01 00:01:00	2
GOOG	2001-01-01 00:02:00	3

We have another table containing portfolio `holdings` at various points in time:

ticker	when	shares
APPL	2000-12-31 23:59:30	5.16
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	24.13
GOOG	2000-12-31 23:59:30	9.33
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	10.58

ticker	when	shares
DATA	2000-12-31 23:59:30	6.65
DATA	2001-01-01 00:00:30	17.95
DATA	2001-01-01 00:01:30	18.37

To load these tables to DuckDB, run:

```
CREATE TABLE prices AS FROM 'https://duckdb.org/data/prices.csv';
CREATE TABLE holdings AS FROM 'https://duckdb.org/data/holdings.csv';
```

Inner AsOf Joins

We can compute the value of each holding at that point in time by finding the most recent price before the holding's timestamp by using an AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p
       ON h.ticker = p.ticker
      AND h.when >= p.when;
```

This attaches the value of the holding at that time to each row:

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16

It essentially executes a function defined by looking up nearby values in the `prices` table. Note also that missing `ticker` values do not have a match and don't appear in the output.

Outer AsOf Joins

Because AsOf produces at most one match from the right hand side, the left side table will not grow as a result of the join, but it could shrink if there are missing times on the right. To handle this situation, you can use an *outer* AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
ASOF LEFT JOIN prices p
           ON h.ticker = p.ticker
          AND h.when >= p.when
ORDER BY ALL;
```

As you might expect, this will produce `NULL` prices and values instead of dropping left side rows when there is no ticker or the time is before the prices begin.

ticker	when	value
APPL	2000-12-31 23:59:30	

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
DATA	2000-12-31 23:59:30	
DATA	2001-01-01 00:00:30	
DATA	2001-01-01 00:01:30	
GOOG	2000-12-31 23:59:30	
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16

AsOf Joins with the `USING` Keyword

So far we have been explicit about specifying the conditions for AsOf, but SQL also has a simplified join condition syntax for the common case where the column names are the same in both tables. This syntax uses the `USING` keyword to list the fields that should be compared for equality. AsOf also supports this syntax, but with two restrictions:

- The last field is the inequality
- The inequality is >= (the most common case)

Our first query can then be written as:

```
SELECT ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p USING (ticker, "when");
```

Clarification on Column Selection with `USING` in ASOF Joins

When you use the `USING` keyword in a join, the columns specified in the `USING` clause are merged in the result set. This means that if you run:

```
SELECT *
FROM holdings h
ASOF JOIN prices p USING (ticker, "when");
```

You will get back only the columns `h.ticker, h.when, h.shares, p.price`. The columns `ticker` and `when` will appear only once, with `ticker` and `when` coming from the left table (holdings).

This behavior is fine for the `ticker` column because the value is the same in both tables. However, for the `when` column, the values might differ between the two tables due to the >= condition used in the AsOf join. The AsOf join is designed to match each row in the left table (`holdings`) with the nearest preceding row in the right table (`prices`) based on the `when` column.

If you want to retrieve the `when` column from both tables to see both timestamps, you need to list the columns explicitly rather than relying on `*`, like so:

```
SELECT h.ticker, h.when AS holdings_when, p.when AS prices_when, h.shares, p.price
FROM holdings h
ASOF JOIN prices p USING (ticker, "when");
```

This ensures that you get the complete information from both tables, avoiding any potential confusion caused by the default behavior of the `USING` keyword.

See Also

For implementation details, see the blog post "DuckDB's AsOf joins: Fuzzy Temporal Lookups".

Full-Text Search

DuckDB supports full-text search via the `fts` extension. A full-text index allows for a query to quickly search for all occurrences of individual words within longer text strings.

Example: Shakespeare Corpus

Here's an example of building a full-text index of Shakespeare's plays.

```
CREATE TABLE corpus AS
    SELECT * FROM 'https://blobs.duckdb.org/data/shakespeare.parquet';

DESCRIBE corpus;
```

column_name	column_type	null	key	default	extra
line_id	VARCHAR	YES	NULL	NULL	NULL
play_name	VARCHAR	YES	NULL	NULL	NULL
line_number	VARCHAR	YES	NULL	NULL	NULL
speaker	VARCHAR	YES	NULL	NULL	NULL
text_entry	VARCHAR	YES	NULL	NULL	NULL

The text of each line is in `text_entry`, and a unique key for each line is in `line_id`.

Creating a Full-Text Search Index

First, we create the index, specifying the table name, the unique id column, and the column(s) to index. We will just index the single column `text_entry`, which contains the text of the lines in the play.

```
PRAGMA create_fts_index('corpus', 'line_id', 'text_entry');
```

The table is now ready to query using the Okapi BM25 ranking function. Rows with no match return a `NULL` score.

What does Shakespeare say about butter?

```
SELECT *
FROM (
    SELECT
        fts_main_corpus.match_bm25(line_id, 'butter') AS score,
        line_id, play_name, speaker, text_entry
    FROM corpus
) sq
WHERE score IS NOT NULL
ORDER BY score DESC;
```

score	line_id	play_name	speaker	text_entry
4.427313429798464	H4/2.4.494	Henry IV	Carrier	As fat as butter.
3.836270302568675	H4/1.2.21	Henry IV	FALSTAFF	prologue to an egg and butter.
3.836270302568675	H4/2.1.55	Henry IV	Chamberlain	They are up already, and call for eggs and butter;
3.3844488405497115	H4/4.2.21	Henry IV	FALSTAFF	toasts-and-butter, with hearts in their bellies no

score	line_id	play_name	speaker	text_entry
3.3844488405497115	H4/4.2.62	Henry IV	PRINCE HENRY	already made thee butter. But tell me, Jack, whose
3.3844488405497115	AWW/4.1.40	Alls well that ends well	PAROLLES	butter-womans mouth and buy myself another of
3.3844488405497115	AYLI/3.2.93	As you like it	TOUCHSTONE	right butter-womens rank to market.
3.3844488405497115	KL/2.4.132	King Lear	Fool	kindness to his horse, buttered his hay.
3.0278411214953107	AWW/5.2.9	Alls well that ends well	Clown	henceforth eat no fish of fortunes buttering.
3.0278411214953107	MWW/2.2.260	Merry Wives of Windsor	FALSTAFF	Hang him, mechanical salt-butter rogue! I will
3.0278411214953107	MWW/2.2.284	Merry Wives of Windsor	FORD	rather trust a Fleming with my butter, Parson Hugh
3.0278411214953107	MWW/3.5.7	Merry Wives of Windsor	FALSTAFF	Ill have my brains taen out and buttered, and give
3.0278411214953107	MWW/3.5.102	Merry Wives of Windsor	FALSTAFF	to heat as butter; a man of continual dissolution
2.739219044070792	H4/2.4.115	Henry IV	PRINCE HENRY	Didst thou never see Titan kiss a dish of butter?

Unlike standard indexes, full-text indexes don't auto-update as the underlying data is changed, so you need to `PRAGMA drop_fts_index('corpus')` and recreate it when appropriate.

Note on Generating the Corpus Table

For more details, see the "Generating a Shakespeare corpus for full-text searching from JSON" blog post.

- The Columns are: `line_id`, `play_name`, `line_number`, `speaker`, `text_entry`.
- We need a unique key for each row in order for full-text searching to work.
- The `line_id` `KL/2.4.132` means King Lear, Act 2, Scene 4, Line 132.

Graph Queries

DuckDB supports graph queries via the DuckPGQ community extension, which implements the SQL/PGQ syntax from the SQL:2023 standard.

> **Warning.** DuckPGQ is a community extension and is still under active development. It may not be available for every DuckDB version: check the DuckPGQ community extension page for the supported versions. Moreover, some features may be incomplete. See the DuckPGQ website for the latest status.

Graph queries allow you to find patterns and paths in connected data, such as social networks, financial transactions, or knowledge graphs, using a visual, intuitive syntax.

Installing DuckPGQ

```
INSTALL duckpgq FROM community;
LOAD duckpgq;
```

Starting with DuckDB 2.0, extensions that add new syntax hook into the grammar of the PEG parser, and you have to opt in to an extension's grammar explicitly. To enable the SQL/PGQ syntax of DuckPGQ (e.g., `CREATE PROPERTY GRAPH` and `GRAPH_TABLE`), run:

```
SET active_grammar_extensions = ['duckpgq'];
```

Creating a Property Graph

A property graph consists of vertices (nodes) and edges (relationships). You create one as a layer on top of existing tables:

```
CREATE TABLE Person (id BIGINT, name VARCHAR);
CREATE TABLE Knows (person1_id BIGINT, person2_id BIGINT, since DATE);

INSERT INTO Person VALUES (1, 'Alice'), (2, 'Bob'), (3, 'Charlie');
INSERT INTO Knows VALUES (1, 2, '2020-01-01'), (2, 3, '2021-06-15');

CREATE PROPERTY GRAPH social_network
VERTEX TABLES (
    Person
)
EDGE TABLES (
    Knows
        SOURCE KEY (person1_id) REFERENCES Person (id)
        DESTINATION KEY (person2_id) REFERENCES Person (id)
);
```

Pattern Matching

Use the GRAPH_TABLE function with MATCH to find patterns. The syntax uses `()` for nodes and `[]` for edges:

```
FROM GRAPH_TABLE (social_network
    MATCH (a:Person)-[k:Knows]->(b:Person)
    COLUMNS (a.name AS person1, b.name AS person2, k.since)
);
```

person1	person2	since
Alice	Bob	2020-01-01
Bob	Charlie	2021-06-15

Path Finding

Find paths of variable length using quantifiers like `{1,5}` (1 to 5 hops) or + (one or more):

```
FROM GRAPH_TABLE (social_network
    MATCH p = ANY SHORTEST (a:Person)-[k:Knows]->{1,3}(b:Person)
    WHERE a.name = 'Alice' AND b.name = 'Charlie'
    COLUMNS (a.name AS start_person, b.name AS end_person, path_length(p) AS hops)
);
```

start_person	end_person	hops
Alice	Charlie	2

Graph Algorithms

DuckPGQ includes built-in graph algorithms:

Function	Description
`pagerank(graph, vertex_label, edge_label)`	Computes PageRank centrality scores
`local_clustering_coefficient(graph, vertex_label, edge_label)`	Measures how connected a node's neighbors are
`weakly_connected_component(graph, vertex_label, edge_label)`	Identifies connected components

Example:

```
FROM pagerank('social_network', 'Person', 'Knows');
```

Use Case: Financial Fraud Detection

Graph queries excel at finding suspicious patterns in financial data. See the "Uncovering Financial Crime with DuckDB and Graph Queries" blog post for a detailed example of detecting money laundering patterns.

Cleanup

To remove a property graph:

```
DROP PROPERTY GRAPH social_network;
```

Further Reading

- DuckPGQ Documentation
- DuckPGQ Community Extension
- "Uncovering Financial Crime with DuckDB and Graph Queries" blog post

query and query_table Functions

The `query_table` and `query` functions enable powerful and more dynamic SQL.

The `query_table` function returns the table whose name is specified by its string argument; the `query` function returns the table obtained by executing the query specified by its string argument.

Both functions only accept constant strings. For example, they allow passing in a table name as a prepared statement parameter:

```
CREATE TABLE my_table (i INTEGER);
INSERT INTO my_table VALUES (42);

PREPARE select_from_table AS SELECT * FROM query_table($1);
EXECUTE select_from_table('my_table');
```

i
42

When combined with the `COLUMNS` expression, we can write very generic SQL-only macros. For example, below is a custom version of `SUMMARIZE` that computes the `min` and `max` of every column in a table:

```
CREATE OR REPLACE MACRO my_summarize(table_name) AS TABLE
SELECT
    unnest([*COLUMNS('alias_.*')]) AS column_name,
    unnest([*COLUMNS('min_.*')]) AS min_value,
    unnest([*COLUMNS('max_.*')]) AS max_value
FROM (
    SELECT
        any_value(alias(COLUMNS(*))) AS "alias_\0",
        min(COLUMNS(*))::VARCHAR AS "min_\0",
        max(COLUMNS(*))::VARCHAR AS "max_\0"
    FROM query_table(table_name::VARCHAR)
);

SELECT *
FROM my_summarize('https://blobs.duckdb.org/data/ontime.parquet')
LIMIT 3;
```

column_name	min_value	max_value
year	2017	2017
quarter	1	3
month	1	9

The `query` function allows for even more flexibility. For example, users who prefer pandas' `stack` syntax over SQL's `UNPIVOT` syntax, may use:

```
CREATE OR REPLACE MACRO stack(table_name, index, name, values) AS TABLE
FROM query(
    'UNPIVOT ' || table_name
    || ' ON COLUMNS(* EXCLUDE (' || array_to_string(index, ', ')
    || ')) INTO NAME ' || name || ' VALUES ' || values
);

WITH cities AS (
    FROM (
        VALUES
            ('NL', 'Amsterdam', '10', '12', '15'),
            ('US', 'New York', '100', '120', '150')
    ) _(country, city, '2000', '2010', '2020')
)
SELECT *
FROM stack('cities', ['country', 'city'], 'year', 'population');
```

country	city	year	population
NL	Amsterdam	2000	10
NL	Amsterdam	2010	12
NL	Amsterdam	2020	15
US	New York	2000	100
US	New York	2010	120
US	New York	2020	150

Merge Statement for SCD Type 2

This is a practical, step-by-step guide to using DuckDB's MERGE statement (introduced in v1.4.0) to perform upserts and build Slowly Changing Dimension Type 2 (SCD Type 2) tables. Type 2 SCDs let you keep full historical versions of records while clearly identifying the current version, perfect for audit trails, data warehousing, and analytical workloads. Type 2 SCDs are practical when you want to know previous values of your primary key data, when it changed and for how long it was in a particular state.

Why Use MERGE in DuckDB?

- Single SQL statement for `INSERT`, `UPDATE`, and soft `DELETE` (upsert and expire).
- Much cleaner and faster than equivalent Python/Pandas logic.
- Full history tracking without hard deletes.
- Works directly on Parquet, CSV, databases, thanks to DuckDB's connectivity!

Prerequisites

- Basic SQL knowledge

Key Terminology

Term	Meaning
Target table	The main/master table you are updating (e.g., `master_ducks`)
Source table	The incoming/new data (e.g., `incoming_ducks`)
MERGE INTO	Specifies the target table
USING	Specifies the source table/query
ON	Join condition (usually primary/business key + current flag)
WHEN MATCHED	Row exists in both → typically UPDATE (or DELETE)
WHEN NOT MATCHED BY TARGET	New row (insert)
WHEN NOT MATCHED BY SOURCE	Row disappeared → soft-delete/expire old version
RETURNING merge_action	Optional: shows what happened to each row (INSERT/UPDATE/DELETE)

Build an SCD Type 2 Dimension Table

We'll track ducks and preserve history whenever their name, breed, or location changes.

> DuckDB has a frontend notebook UI, this is great for managing several SQL statements and segmenting your code. The UI ships with the DuckDB CLI, so if you have the CLI installed you can use the front end. To start the notebook front end just run: `duckdb -ui` and you can navigate to http://localhost:4213/ to start writing your SQL code inside of your notebooks. Just copy and paste the following code blocks to follow this guide.

Step 1: Create the Incoming (source) Table

This table represents today's transactional data.

```
CREATE TABLE IF NOT EXISTS incoming_ducks (
    duck_id     INTEGER,
    duck_name   VARCHAR,
    breed       VARCHAR,
    location    VARCHAR,
    begin_date  DATE,
    end_date    DATE,
    is_current  BOOLEAN
);

INSERT INTO incoming_ducks VALUES
    (101, 'Quackers',   'Mallard',       'Pond B',      CURRENT_DATE - INTERVAL '1 day', NULL, true),
    (102, 'Waddles',    'Pekin',         'Pond A',      CURRENT_DATE - INTERVAL '1 day', NULL, true),
    (104, 'Splash',     'Muscovy',       'Pond C',      CURRENT_DATE - INTERVAL '1 day', NULL, true),
    (105, 'Puddles',    'Indian Runner', 'Relocated',   CURRENT_DATE - INTERVAL '1 day', NULL, true);
```

Step 2: Create the Master (target) Table

This table represents the type 2 SCD data (i.e., transaction data with history).

```
CREATE TABLE IF NOT EXISTS master_ducks (
    record_id   INTEGER PRIMARY KEY,
    duck_id     INTEGER NOT NULL,
    duck_name   VARCHAR,
    breed       VARCHAR,
    location    VARCHAR,
    begin_date  DATE NOT NULL,
    end_date    DATE,
    is_current  BOOLEAN NOT NULL DEFAULT true
);

CREATE SEQUENCE IF NOT EXISTS duck_record_seq START 1;

INSERT INTO master_ducks VALUES
    (nextval('duck_record_seq'), 101, 'Quackers', 'Mallard',       'Pond A', CURRENT_DATE - INTERVAL '2
days', NULL, true),
    (nextval('duck_record_seq'), 102, 'Waddles',  'Pekin',         'Pond A', CURRENT_DATE - INTERVAL '2
days', NULL, true),
    (nextval('duck_record_seq'), 103, 'Feathers', 'Rouen',         'Pond B', CURRENT_DATE - INTERVAL '2
days', NULL, true),
    (nextval('duck_record_seq'), 105, 'Puddles',  'Indian Runner', 'Pond A', CURRENT_DATE - INTERVAL '2
days', NULL, true);
```

Step 3: Perform the Merge Statement

This statement will perform the merge, it will check for differences between the data of target and source and follow the `WHEN MATCHED` or `WHEN NOT MATCHED` logic specified.

```
MERGE INTO master_ducks AS target
USING incoming_ducks AS source
ON target.duck_id = source.duck_id AND target.is_current = true

WHEN MATCHED AND (
       target.duck_name <> source.duck_name OR
       target.breed     <> source.breed     OR
       target.location  <> source.location
) THEN UPDATE SET
    end_date    = CURRENT_DATE - INTERVAL '1 day',
    is_current  = false

WHEN NOT MATCHED BY SOURCE AND target.is_current = true THEN UPDATE SET
    end_date    = CURRENT_DATE - INTERVAL '1 day',
    is_current  = false
```

```
WHEN NOT MATCHED BY TARGET THEN INSERT (
    record_id, duck_id, duck_name, breed, location,
    begin_date, end_date, is_current
) VALUES (
    nextval('duck_record_seq'),
    source.duck_id, source.duck_name, source.breed, source.location,
    source.begin_date, source.end_date, source.is_current
)

RETURNING merge_action, *;
```

Step 4: Insert New Current Versions for Changed Records

This statement inserts the new current records into the master table. While it's possible to achieve the same result using the MERGE statement's RETURNING clause, this two-step approach is more straightforward and easier to understand.

```
INSERT INTO master_ducks (
    record_id, duck_id, duck_name, breed, location,
    begin_date, end_date, is_current
)
SELECT
    nextval('duck_record_seq'),
    source.duck_id,
    source.duck_name,
    source.breed,
    source.location,
    CURRENT_DATE AS begin_date,
    NULL AS end_date,
    true AS is_current
FROM incoming_ducks AS source
INNER JOIN master_ducks AS target
    ON source.duck_id = target.duck_id
WHERE target.is_current = false
  AND target.end_date = CURRENT_DATE - INTERVAL '1 day';
```

Step 5: Query The Results

The following queries can be used to examine the data resulting from the MERGE statement.

```
-- All history
SELECT * FROM master_ducks ORDER BY duck_id, begin_date DESC;

-- Only current records
SELECT * FROM master_ducks WHERE is_current = true;

-- Only expired historical records
SELECT * FROM master_ducks WHERE is_current = false ORDER BY duck_id, begin_date DESC;
```

Step 6: Examine a Single Duck

To better illustrate the concept, let's examine a single duck, to drive home the value add for type 2 SCDs. If we select from the master table after running the merge statement and the post update insert statement, we can see the individual rows for `Quackers`.

To view the original row of data that is historical:

```
SELECT * FROM master_ducks WHERE duck_name = 'Quackers' AND is_current = false;
```

Returns:

record_id	duck_id	duck_name	breed	location	begin_date	end_date	is_current
1	101	Quackers	Mallard	Pond A	2025-11-24	2025-11-25	false

Note:

- The `end_date` is NOT NULL, it has the date when this duck's data was updated.
- The `is_current` is `false` indicating this is a historical record.
- The field that will change is `location`, it is currently `Pond A` and will be updated to `Pond B`.

To view the current row of data:

```
SELECT * FROM master_ducks WHERE duck_name = 'Quackers' AND is_current = true;
```

record_id	duck_id	duck_name	breed	location	begin_date	end_date	is_current
10	101	Quackers	Mallard	Pond B	2025-11-26	NULL	true

Note:

- The `end_date` is NULL, the NULL in this context indicates this is the latest record for this `duck_id`.
- The `is_current` is `true` also indicating this is a current record.
- The `location` is now `Pond B`.

To view all of `Quackers` data, which will contain both current and non-current rows:

```
SELECT * FROM master_ducks WHERE duck_name = 'Quackers';
```

record_id	duck_id	duck_name	breed	location	begin_date	end_date	is_current
1	101	Quackers	Mallard	Pond A	2025-11-24	2025-11-25	false
10	101	Quackers	Mallard	Pond B	2025-11-26	NULL	true

Common Patterns and Variations

Use Case	Clause to Use
Simple upsert (no history)	`WHEN MATCHED THEN UPDATE` and `WHEN NOT MATCHED BY TARGET THEN INSERT`
Upsert and delete missing rows	Add `WHEN NOT MATCHED BY SOURCE THEN DELETE`
Only insert new, never update	Omit `WHEN MATCHED`
Return affected rows	Add `RETURNING merge_action, *`

Best Practices

- Remember that `TARGET` is the master table and `SOURCE` is the incoming table or query.
- Keep `end_date` `NULL` for current rows (makes queries faster).
- Wrap `MERGE` and `INSERT` statements in a transaction when needed.
- Use a primary key or a surrogate key for uniqueness.
- Test with `RETURNING` first.

Timestamp Issues

Timestamp with Time Zone Promotion Casts

Working with time zones in SQL can be quite confusing at times. For example, when filtering to a date range, one might try the following query:

```
SET timezone = 'America/Los_Angeles';

CREATE TABLE times AS
    FROM range('2025-08-30'::TIMESTAMPTZ, '2025-08-31'::TIMESTAMPTZ, INTERVAL 1 HOUR) tbl(t);

FROM times WHERE t <= '2025-08-30';
```

```
┌──────────────────────────┐
│            t             │
│ timestamp with time zone │
├──────────────────────────┤
│ 2025-08-30 00:00:00-07   │
└──────────────────────────┘
```

But if you change to another time zone, the results of the query change:

```
SET timezone = 'HST';
FROM times WHERE t <= '2025-08-30';
```

```
┌──────────────────────────┐
│            t             │
│ timestamp with time zone │
├──────────────────────────┤
│ 2025-08-29 21:00:00-10   │
│ 2025-08-29 22:00:00-10   │
│ 2025-08-29 23:00:00-10   │
│ 2025-08-30 00:00:00-10   │
└──────────────────────────┘
```

Or worse:

```
SET timezone = 'America/New_York';
FROM times WHERE t <= '2025-08-30';
```

```
┌──────────────────────────┐
│            t             │
│ timestamp with time zone │
├──────────────────────────┤
│          0 rows          │
└──────────────────────────┘
```

These confusing results are due to the SQL casting rules from `DATE` to `TIMESTAMP WITH TIME ZONE`. This cast is required to promote the date to midnight *in the current time zone*.

In general, unless you need to use the current time zone for display (or other temporal binning operations) you should use plain `TIMESTAMP`s for temporal data. This will avoid confusing issues such as this, and the arithmetic operations are generally faster.

Time Zone Performance

DuckDB's time zone support is provided by the `icu` extension. Starting with DuckDB 2.0, the extension no longer depends on the *International Components for Unicode* (ICU) library: it implements time zones and calendars natively, using time zone data built from the IANA time zone database. This implementation supports daylight savings time past 2037 (Note: Pandas gives incorrect results past that year) and is faster than the previous ICU-based one.

Even so, time zone-aware binning is more expensive than arithmetic on plain `TIMESTAMP`s. If the same temporal bins are computed repeatedly, one workaround is to precompute them in a calendar table for the timestamps being modeled. For example, if the application is modeling electrical supply and demand out to 2100 at hourly resolution, one can create the calendar table like so:

```
SET timezone = 'Europe/Amsterdam';

CREATE OR REPLACE TABLE hourly AS
    SELECT
        ts,
        year::SMALLINT AS year,
        month::TINYINT AS month,
        day::TINYINT AS day,
        hour::TINYINT AS hour,
    FROM (
        SELECT ts, unnest(date_part(['year', 'month', 'day', 'hour',], ts))
        FROM generate_series(
            '2020-01-01'::DATE::TIMESTAMPTZ,
            '2100-01-01'::DATE::TIMESTAMPTZ,
            INTERVAL 1 HOUR) tbl(ts)
    ) parts;
```

You can then join this ~700K row table against any timestamp column to quickly obtain the temporal bin values for the time zone in question. The inner casts are not required, but result in a smaller table because `date_part` returns 64 bit integers for all parts.

Notice that we can extract *all* of the parts with a single call to `date_part`. This part list version of the function is faster than extracting the parts one by one because the underlying binning computation computes all parts, so picking out the ones in the list avoids duplicate time zone conversions.

Also notice that we are leveraging the `DATE` cast rules from the previous section to bound the calendar to the model domain.

Half-Open Intervals

Another subtle problem in using SQL for temporal analytics is the `BETWEEN` operator. Temporal analytics almost always uses half-open binning intervals to avoid overlaps at the ends. Unfortunately, the `BETWEEN` operator is a closed-closed interval:

```
x BETWEEN begin AND end
-- expands to
begin <= x AND x <= end
-- not
begin <= x AND x < end
```

To avoid this problem, make sure you are explicit about comparison boundaries instead of using `BETWEEN`.

Snippets

Create Synthetic Data

DuckDB lets you quickly generate synthetic datasets by combining a few building blocks: functions and clauses that generate *rows*, functions that generate *values* to fill those rows, and extensions that generate whole benchmark datasets. This page shows each building block with a runnable example.

Generating Rows

- range functions
- hash functions, e.g., `hash`, `md5`, `sha256`
- the Faker Python package via the Python function API
- using cross products (Cartesian products)

`range` and `generate_series`

The range functions turn a start, stop, and step into a table of rows, which is the usual starting point for a synthetic dataset:

```
SELECT i FROM range(1, 4) t(i);
```

i
int64
1
2
3

`range` excludes the stop value, while `generate_series` includes it.

VALUES

The VALUES clause generates rows from literal tuples, which is convenient for a small fixed set of values to draw from:

```
SELECT * FROM (VALUES ('Alice', 'Engineering'), ('Bob', 'Sales')) t(name, department);
```

name	department
varchar	varchar
Alice	Engineering
Bob	Sales

`repeat_row`

`repeat_row(varargs, num_rows)` returns a fixed number of identical rows, which is useful as a scaffold to fill with generated values:

```
FROM repeat_row('placeholder', 0, num_rows = 3);
```

column0 varchar	column1 int32
placeholder	0
placeholder	0
placeholder	0

Cross Products

A cross product (Cartesian product) multiplies rows from two sources, so combining two `range` calls is a compact way to produce a large number of rows:

```
SELECT i, j FROM range(1, 3) s(i) CROSS JOIN range(1, 3) t(j);
```

Generating Values

Random Numbers

`random()` returns a random DOUBLE in the range `[0, 1)`, which you can scale and cast into any numeric range:

```
SELECT (random() * 100)::INTEGER AS score FROM range(3);
```

UUIDs

Several utility functions generate universally unique identifiers, which make good synthetic primary keys: `uuid()` and its alias `uuidv4()` generate random (version 4) UUIDs, `uuidv7()` generates time-ordered (version 7) UUIDs, and `gen_random_uuid()` is an alias for `uuid()`:

```
SELECT uuidv7() AS id FROM range(3);
```

Current Date and Time

Functions that return the current date or time are handy for populating timestamp columns: `today()` returns the current date, `get_current_time()` returns the current time, and `current_localtimestamp()` returns the current local timestamp:

```
SELECT today() AS date, get_current_time() AS time, current_localtimestamp() AS timestamp;
```

Hashing

Hash functions such as `hash`, `md5` and `sha256` turn a row number into a deterministic pseudo-random value, which is useful when you want reproducible identifiers:

```
SELECT hash(i) AS id, md5(i::VARCHAR) AS token FROM range(3) t(i);
```

Repeating Strings

`repeat(string, count)` builds a string by repeating another, which is useful for padding values to a target length:

```
SELECT repeat('ab', 3) AS padded;
```

Sequences

A sequence produces a monotonically increasing series of numbers. `nextval('sequence_name')` advances the sequence and returns the next value, while `currval('sequence_name')` returns the value most recently produced:

```
CREATE SEQUENCE id_seq START 100;
SELECT nextval('id_seq') AS id FROM range(3);
```

Generating Text with Faker

For realistic-looking names, addresses and other text, you can call the Faker Python package through the Python function API:

```
import duckdb

from duckdb.sqltypes import *
from faker import Faker

fake = Faker()

def random_date():
    return fake.date_between()

def random_short_text():
    return fake.text(max_nb_chars=20)

def random_long_text():
    return fake.text(max_nb_chars=200)

con = duckdb.connect()
con.create_function("random_date",       random_date,       [], DATE,    type="native", side_effects=True)
con.create_function("random_short_text", random_short_text, [], VARCHAR, type="native", side_effects=True)
con.create_function("random_long_text",  random_long_text,  [], VARCHAR, type="native", side_effects=True)

res = con.sql("""
                SELECT
                   hash(i * 10 + j) AS id,
                   random_date() AS creation_date,
                   random_short_text() AS short,
                   random_long_text() AS long,
                   IF (j % 2, TRUE, FALSE) AS bool
                FROM generate_series(1, 5) s(i)
                CROSS JOIN generate_series(1, 2) t(j)
                """)
res.show()
```

This generates the following:

```
┌──────────────────────┬───────────────┬──────────────────┬──────────────────────────────────────────────────┬─────────┐
│          id          │ creation_date │      short       │                      long                        │  bool   │
│        uint64        │     date      │     varchar      │                      varchar                     │ boolean │
├──────────────────────┼───────────────┼──────────────────┼──────────────────────────────────────────────────┼─────────┤
│  6770051751173734325 │ 1998-01-04    │ Head keep face age. │ Stay different gun according make large read… │ true    │
```

```
│ 16510940941872865459 │ 2006-07-30 │ Ten quickly expect. │ Early race claim. Middle economy box nor. An… │ true  │
│ 13285076694688170502 │ 2018-02-19 │ Spring writer onto. │ Lose painting time dog sure marriage month. … │ true  │
│ 11757770452869451863 │ 2011-09-01 │ Recently single.    │ Power class party information this. Appear e… │ true  │
│  2064835973596856015 │ 2014-03-09 │ Write nor national. │ Total need tend Democrat. Group under boy no… │ true  │
│ 17776805813723356275 │ 2000-10-21 │ Commercial occur.   │ Professor big but executive plan fast body b… │ false │
│ 13540103502347468651 │ 2023-08-23 │ Specific author.    │ Across cut market eight. Thing pull open spa… │ false │
│  4800297459639118879 │ 2004-02-18 │ Beautiful learn.    │ Upon common box beautiful. War check investm… │ false │
│  7199933130570745587 │ 2024-11-30 │ Fact nation cup.    │ Trouble that sing former sing sure range ser… │ false │
│ 18103378254596719331 │ 2000-07-27 │ Late ago serious.   │ Million operation six school station give us… │ false │
└──────────────────────┴────────────┴─────────────────────┴───────────────────────────────────────────────┴───────┘
  10 rows                                                                                          5 columns
```

Generating Benchmark Datasets with Extensions

To generate large, well-defined datasets, DuckDB ships with the TPC-H and TPC-DS extensions. Each provides a data generator function that creates and populates the benchmark's tables at a given scale factor:

```
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 0.1);
SELECT count(*) AS lineitem_rows FROM lineitem;
```

Community Extensions

The `fakeit` community extension provides table functions that generate fake data (such as names, emails, and addresses) directly in SQL, similar to the Faker Python package.

Dutch Railway Datasets

Examples in this documentation often use datasets based on the Dutch Railway datasets. These high-quality datasets are maintained by the team behind the Rijden de Treinen *(Are the trains running?)* application. This page contains download links to our mirrors to the datasets.

> In 2024, we have published a blog post on the analysis of these datasets.

Loading the Datasets

You can load the datasets directly as follows:

```
CREATE TABLE services AS
    FROM 'https://blobs.duckdb.org/nl-railway/services-2025-03.csv.gz';

DESCRIBE services;
```

column_name	column_type	null	key	default	extra
Service:RDT-ID	BIGINT	YES	NULL	NULL	NULL
Service:Date	DATE	YES	NULL	NULL	NULL
Service:Type	VARCHAR	YES	NULL	NULL	NULL
Service:Company	VARCHAR	YES	NULL	NULL	NULL
Service:Train number	BIGINT	YES	NULL	NULL	NULL
Service:Completely cancelled	BOOLEAN	YES	NULL	NULL	NULL
Service:Partly cancelled	BOOLEAN	YES	NULL	NULL	NULL
Service:Maximum delay	BIGINT	YES	NULL	NULL	NULL
Stop:RDT-ID	BIGINT	YES	NULL	NULL	NULL
Stop:Station code	VARCHAR	YES	NULL	NULL	NULL
Stop:Station name	VARCHAR	YES	NULL	NULL	NULL
Stop:Arrival time	TIMESTAMP WITH TIME ZONE	YES	NULL	NULL	NULL
Stop:Arrival delay	BIGINT	YES	NULL	NULL	NULL
Stop:Arrival cancelled	BOOLEAN	YES	NULL	NULL	NULL
Stop:Departure time	TIMESTAMP WITH TIME ZONE	YES	NULL	NULL	NULL
Stop:Departure delay	BIGINT	YES	NULL	NULL	NULL
Stop:Departure cancelled	BOOLEAN	YES	NULL	NULL	NULL

Datasets

80-Month Datasets

- 2019-01 to 2025-08: 80 months as uncompressed CSVs in a single zip

Yearly Datasets

The yearly datasets are about 350 MB each.

- 2019
- 2020
- 2021
- 2022
- 2023
- 2024
- 2025

Monthly Datasets

The monthly datasets are about 30 MB each.

- 2024-01
- 2024-02
- 2024-03
- 2024-04

- 2024-05
- 2024-06
- 2024-07
- 2024-08
- 2024-09
- 2024-10
- 2024-11
- 2024-12
- 2025-01
- 2025-02
- 2025-03
- 2025-04
- 2025-05
- 2025-06
- 2025-07
- 2025-08
- 2025-09
- 2025-10
- 2025-11
- 2025-12
- 2026-01
- 2026-02
- 2026-03
- 2026-04

Sharing Macros

DuckDB has a powerful macro mechanism that allows creating shorthands for common tasks.

Sharing a Scalar Macro

First, we define a macro that pretty-prints a non-negative integer as a short string with thousands, millions, and billions (without rounding) as follows:

`duckdb pretty_print_integer_macro.duckdb`

```
CREATE MACRO pretty_print_integer(n) AS
    CASE
        WHEN n >= 1_000_000_000 THEN printf('%dB', n // 1_000_000_000)
        WHEN n >= 1_000_000     THEN printf('%dM', n // 1_000_000)
        WHEN n >= 1_000         THEN printf('%dk', n // 1_000)
        ELSE printf('%d', n)
    END;

SELECT pretty_print_integer(25_500_000) AS x;
```

x
varchar
25M

As one would expect, the macro gets persisted in the database. But this also means that we can host it on an HTTPS endpoint and share it with anyone! We have published this macro on `blobs.duckdb.org`.

You can try it from DuckDB:

```
duckdb
```

Make sure that the `httpfs` extension is installed:

```
INSTALL httpfs;
```

You can now attach to the remote endpoint and use the macro:

```
ATTACH 'https://blobs.duckdb.org/data/pretty_print_integer_macro.duckdb'
    AS pretty_print_macro_db;

SELECT pretty_print_macro_db.pretty_print_integer(42_123) AS x;
```

```
┌─────────┐
│    x    │
│ varchar │
├─────────┤
│ 42k     │
└─────────┘
```

Sharing a Table Macro

It's also possible to share table macros. For example, we created the `checksum` macro as follows:

```
duckdb compute_table_checksum.duckdb
```

```
CREATE MACRO checksum(table_name) AS TABLE
    SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
    FROM query_table(table_name);
```

To use it, make sure that the `httpfs` extension is installed:

```
INSTALL httpfs;
```

You can attach to the remote endpoint and use the macro:

```
ATTACH 'https://blobs.duckdb.org/data/compute_table_checksum.duckdb'
    AS compute_table_checksum_db;

CREATE TABLE stations AS
    FROM 'https://blobs.duckdb.org/stations.parquet';

.mode line
FROM compute_table_checksum_db.checksum('stations');
```

```
         id = 207501589970998739957162925925638303138
       code = 126327004005066229305810236187733612209
        uic = 194659031858447341987368539307022394076
 name_short = 225741449355216776462496463050578341773
name_medium = 339714102140420031901247247513112906072
  name_long = 126079956280724674884063510870679874110
       slug = 286823566458906756841161390341104965945
    country = 143068442936912051858689770843609587944
       type = 566566231547078545614740060408879751
    geo_lat = 160608116135251821259126521573759502306
    geo_lng = 201985085848282999780447884260076663724
```

Analyzing a Git Repository

You can use DuckDB to analyze Git logs using the output of the `git log` command.

Exporting the Git Log

We start by picking a character that doesn't occur in any part of the commit log (author names, messages, etc). Since version v1.2.0, DuckDB's CSV reader supports 4-byte delimiters, making it possible to use emojis! ⊠

Despite being featured in the Emoji Movie (IMDb rating: 3.4), we can assume that the Fish Cake with Swirl emoji (🍥) is not a common occurrence in most Git logs. So, let's clone the duckdb/duckdb repository and export its log as follows:

```
git log --date=iso-strict --pretty=format:%ad🍥%h🍥%an🍥%s > git-log.csv
```

The resulting file looks like this:

```
2025-02-25T18:12:54+01:00🍥d608a31e13🍥Mark🍥MAIN_BRANCH_VERSIONING: Adopt also for Python build and
amalgamation (#16400)
2025-02-25T15:05:56+01:00🍥920b39ad96🍥Mark🍥Read support for Parquet Float16 (#16395)
2025-02-25T13:43:52+01:00🍥61f55734b9🍥Carlo Piovesan🍥MAIN_BRANCH_VERSIONING: Adopt also for Python build and
amalgamation
2025-02-25T12:35:28+01:00🍥87eff7ebd3🍥Mark🍥Fix issue #16377 (#16391)
2025-02-25T10:33:49+01:00🍥35af26476e🍥Hannes Mühleisen🍥Read support for Parquet Float16
```

Loading the Git Log into DuckDB

Start DuckDB and read the log as a CSV 🍥SV:

```sql
CREATE TABLE commits AS
    FROM read_csv(
            'git-log.csv',
            delim = '🍥',
            header = false,
            column_names = ['timestamp', 'hash', 'author', 'message']
        );
```

This will result in a nice DuckDB table:

```sql
FROM commits
LIMIT 5;
```

timestamp timestamp with time zone	hash varchar	author varchar	message varchar
2025-02-25 18:12:54+01	d608a31e13	Mark	MAIN_BRANCH_VERSIONING: Adopt also for Python build and amalgamation (#16400)
2025-02-25 15:05:56+01	920b39ad96	Mark	Read support for Parquet Float16 (#16395)
2025-02-25 13:43:52+01	61f55734b9	Carlo Piovesan	MAIN_BRANCH_VERSIONING: Adopt also for Python build and amalgamation
2025-02-25 12:35:28+01	87eff7ebd3	Mark	Fix issue #16377 (#16391)
2025-02-25 10:33:49+01	35af26476e	Hannes Mühleisen	Read support for Parquet Float16

Analyzing the Log

We can analyze the table as any other in DuckDB.

Common Topics

Let's start with a simple question: which topic was the most commonly mentioned in the commit messages: CI, CLI, or Python?

```
SELECT
    message.lower().regexp_extract('\b(ci|cli|python)\b') AS topic,
    count(*) AS num_commits
FROM commits
WHERE topic <> ''
GROUP BY ALL
ORDER BY num_commits DESC;
```

topic varchar	num_commits int64
ci	828
python	666
cli	49

Out of these three topics, commits related to continuous integration dominate the log!

We can also do a more exploratory analysis by looking at all words in the commit messages. To do so, we first tokenize the messages:

```
CREATE TABLE words AS
    SELECT unnest(
        message
            .lower()
            .regexp_replace('\W', ' ')
            .trim(' ')
            .string_split_regex('\W')
        ) AS word
FROM commits;
```

Then, we remove stopwords using a pre-defined list:

```
CREATE TABLE stopwords AS
    SELECT unnest(['a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an', 'and', 'any',
    'are', 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between', 'both', 'but', 'by',
    'can', 'did', 'do', 'does', 'doing', 'don', 'down', 'during', 'each', 'few', 'for', 'from', 'further',
    'had', 'has', 'have', 'having', 'he', 'her', 'here', 'hers', 'herself', 'him', 'himself', 'his', 'how',
    'i', 'if', 'in', 'into', 'is', 'it', 'its', 'itself', 'just', 'me', 'more', 'most', 'my', 'myself',
    'no', 'nor', 'not', 'now', 'of', 'off', 'on', 'once', 'only', 'or', 'other', 'our', 'ours',
    'ourselves', 'out', 'over', 'own', 's', 'same', 'she', 'should', 'so', 'some', 'such', 't', 'than',
    'that', 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there', 'these', 'they', 'this',
    'those', 'through', 'to', 'too', 'under', 'until', 'up', 'very', 'was', 'we', 'were', 'what', 'when',
    'where', 'which', 'while', 'who', 'whom', 'why', 'will', 'with', 'you', 'your', 'yours', 'yourself',
    'yourselves']) AS word;

CREATE OR REPLACE TABLE words AS
    FROM words
    NATURAL ANTI JOIN stopwords
    WHERE word != '';
```

> We use the `NATURAL ANTI JOIN` clause here, which allows us to elegantly filter out values that occur in the `stopwords` table.

Finally, we select the top-20 most common words.

```
SELECT word, count(*) AS count FROM words
GROUP BY ALL
ORDER BY count DESC
LIMIT 20;
```

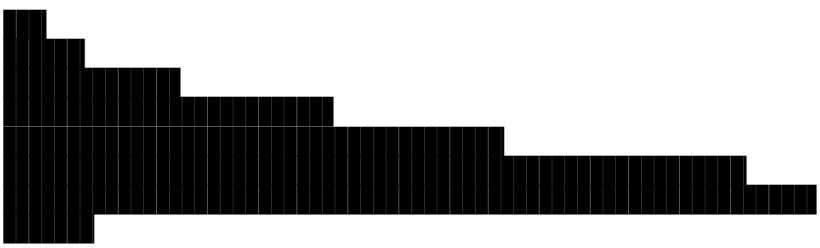
word varchar	count int64
merge	12550
fix	6402
branch	6005
pull	5950
request	5945
add	5687
test	3801
master	3289
tests	2339
issue	1971
main	1935
remove	1884
format	1819
duckdb	1710
use	1442
mytherin	1410
fixes	1333
hawkfish	1147
feature	1139
function	1088
20 rows	

As expected, there are many Git terms (`merge`, `branch`, `pull`, etc.), followed by terminology related to development (`fix`, `test`/`tests`, `issue`, `format`). We also see the account names of some developers (`mytherin`, `hawkfish`), which are likely there due to commit messages for merging pull requests (e.g., "Merge pull request #13776 from Mytherin/expressiondepth"). Finally, we also see some DuckDB-related terms such as `duckdb` (shocking!) and `function`.

Visualizing the Number of Commits

Let's visualize the number of commits each year:

```
SELECT
    year(timestamp) AS year,
    count(*) AS num_commits,
    num_commits.bar(0, 20_000) AS num_commits_viz
FROM commits
GROUP BY ALL
ORDER BY ALL;
```

year int64	num_commits int64	num_commits_viz varchar
2018	870	
2019	1621	
2020	3484	
2021	6488	
2022	9817	
2023	14585	
2024	15949	
2025	1788	

We see a steady growth over the years – especially considering that many of DuckDB's functionalities and clients, which were originally part of the main repository, are now maintained in separate repositories (e.g., Java, R).

Happy hacking!

Importing Duckbox Tables

> The scripts provided in this page work on Linux, macOS, and WSL.

By default, the DuckDB CLI client renders query results in the duckbox format, which uses rich, ASCII-art inspired tables to show data. These tables are often shared verbatim in other documents. For example, take the table used to demonstrate new CSV features in the DuckDB 1.2.0 release blog post:

```
┌─────────┬───────┐
│    a    │   b   │
│ varchar │ int64 │
├─────────┼───────┤
│ hello   │    42 │
│ world   │    84 │
└─────────┴───────┘
```

What if we would like to load this data back to DuckDB? This is not supported by default but it can be achieved by some scripting: we can turn the table into a `│`-separated file and read it with DuckDB's CSV reader. Note that the separator is not the pipe character `|`, instead it is the "Box Drawings Light Vertical" character `│`.

Loading Duckbox Tables to DuckDB

First, we save the table above as `duckbox.csv`. Then, we clean it using `sed`:

```
echo -n > duckbox-cleaned.csv
sed -n "2s/^│ *//;s/ *│$//;s/ *│ */│/p;2q" duckbox.csv >> duckbox-cleaned.csv
sed "1,4d;\$d;s/^│ *//;s/ *│$//;s/ *│ */│/g" duckbox.csv >> duckbox-cleaned.csv
```

The `duckbox-cleaned.csv` file looks as follows:

```
a│b
hello│42
world│84
```

We can then simply load this to DuckDB via:

```
FROM read_csv('duckbox-cleaned.csv', delim = '│');
```

And export it to a CSV:

```
COPY (FROM read_csv('duckbox-cleaned.csv', delim = '│')) TO 'out.csv';
```

```
a,b
hello,42
world,84
```

Using `shellfs`

To parse duckbox tables with a single `read_csv` call – and without creating any temporary files –, we can use the `shellfs` community extension:

```
INSTALL shellfs FROM community;
LOAD shellfs;
FROM read_csv(
        '(sed -n "2s/^│ *//;s/ *│$//;s/ *│ */│/p;2q" duckbox.csv; ' ||
        'sed "1,4d;\$d;s/^│ *//;s/ *│$//;s/ *│ */│/g" duckbox.csv) |',
        delim = '│'
    );
```

We can also create a table macro:

```
CREATE MACRO read_duckbox(path) AS TABLE
    FROM read_csv(
            printf(
                '(sed -n "2s/^│ *//;s/ *│$//;s/ *│ */│/p;2q" %s; ' ||
                'sed "1,4d;\$d;s/^│ *//;s/ *│$//;s/ *│ */│/g" %s) |',
                path, path
            ),
            delim = '│'
        );
```

Then, reading a duckbox table is as simple as:

```
FROM read_duckbox('duckbox.csv');
```

> `shellfs` is a community extension and it comes without any support or guarantees. Only use it if you can ensure that its inputs are appropriately sanitized. Please consult the Securing DuckDB page for more details.

Limitations

Please consider the following limitations when running this script:

- This approach only works if the table does not have long pipe `│` characters. It also trims spaces from the table cell values. Make sure to factor in these assumptions when running the script.
- The script is compatible with both BSD `sed` (which is the default on macOS) and GNU `sed` (which is the default on Linux and available on macOS as `gsed`).
- Only the data types supported by the CSV sniffer are parsed correctly. Values containing nested data will be parsed as a VARCHAR.

Copying an In-Memory Database to a File

Imagine the following situation – you started DuckDB in in-memory mode but would like to persist the state of your database to disk. To achieve this, **attach to a new disk-based database** and use the `COPY FROM DATABASE ... TO` command:

```
ATTACH 'my_database.db';
COPY FROM DATABASE memory TO my_database;
DETACH my_database;
```

> Ensure that the disk-based database file does not exist before attaching to it.

Calculating a Database Checksum

To calculate a database checksum, use a commutative operation such as `bit_xor` on all columns, all rows, and all tables.

First, create a table macro that performs this for each table:

```
CREATE OR REPLACE MACRO table_checksum(table_name) AS TABLE
    SELECT bit_xor(value) AS table_checksum
    FROM (
        UNPIVOT (
            SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
            FROM query_table(table_name)
        )
        ON columns(*)
    );
```

You need to call this macro for all tables. In the CLI, you can do so by dynamically generating a SQL file:

```
.mode list
.headers off
.once checksum_all_tables.sql
SELECT printf('CREATE OR REPLACE TABLE table_checksums (checksum UHUGEINT);')
UNION ALL
SELECT printf('INSERT INTO table_checksums FROM table_checksum(''%s'');', table_name)
FROM duckdb_tables()
WHERE NOT internal;
.mode duckbox
```

Read the SQL file back and calculate the checksum for the database:

```
.read checksum_all_tables.sql
SELECT bit_xor(checksum) AS database_checksum
FROM table_checksums;
```

Troubleshooting

Command Line

Unexpected Markdown Output When Piping

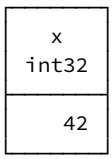
If the output of the command line client is piped to another process and an AI agent environment variable (such as `AI_AGENT`) is set, the CLI switches to agent mode and prints compact Markdown tables instead of the usual duckbox tables:

```
echo "SELECT 42 AS x;" | duckdb | cat
```

```
| x:INTEGER |
|---|
| 42 |
```

To turn off agent mode, use the `-no-agent` command line argument or set the `DUCKDB_AGENT_MODE` environment variable to `0`:

```
echo "SELECT 42 AS x;" | duckdb -no-agent | cat
```



Passing an explicit output mode flag, such as `-csv` or `-json`, also disables agent mode.

Crashes

DuckDB is thoroughly tested via an extensive test suite. However, bugs can still occur and these can sometimes lead to crashes. This page contains practical information on how to troubleshoot DuckDB crashes.

Types of Crashes

There are a few major types of crashes:

- **Termination signals:** The process stops with a `SIGSEGV` (segmentation fault), `SIGABRT`, etc.: these should never occur. Please submit an issue.
- **Internal errors:** an operation may result in an `Internal Error`, e.g.:

  ```
  INTERNAL Error:
  Attempted to access index 3 within vector of size 3
  ```

  After encountering an internal error, DuckDB enters a restricted mode where any further operations will result in the following error message:

  ```
  FATAL Error:
  Failed: database has been invalidated because of a previous fatal error.
  The database must be restarted prior to being used again.
  ```

- **Out of memory errors:** In rare cases, DuckDB can crash due to memory exhaustion. This crash can also be a symptom of the operating system killing the process. For example, many Linux distributions run an OOM reaper or OOM killer process, which kills processes to free up their memory and thus prevents the operating system from running out of memory. If your DuckDB session runs out of memory, consult the "OOM errors" page.
- **Corrupted database or WAL file:** DuckDB displays the following error:

  ```
  Corrupt database file: computed checksum [...] does not match stored checksum [...] in block at location [...]
  ```

  If the WAL file gets corrupted, DuckDB will display:

  ```
  Failure while replaying WAL file "[...].wal":
  Corrupt WAL file: entry at byte position [...] computed checksum [...] does not match stored checksum [...]
  ```

  Similarly to internal errors, calls to a database after encountering these errors will return a `FATAL error`.

  Most often the checksum error is the symptom of a faulty disk. If you encounter such an error with DuckDB, please run a disk check on the machine and attempt to can reproduce the issue on different computers. If you can reproduce the issue on different setups, report it in the issue tracker.

Recovering Data

If your DuckDB session was writing to a persistent database file prior to crashing, there might be a WAL (write-ahead log) file next to your database named `<database_filename>.wal`{:.language-sql .highlight}. To recover data from the WAL file, simply start a new DuckDB session on the persistent database. DuckDB will then replay the write-ahead log and perform a checkpoint operation, restoring the database to the state before the crash.

Troubleshooting the Crash

Using the Latest Stable and Preview Builds

DuckDB is constantly improving, so there is a chance that the bug you have encountered has already been fixed in the codebase. First, try updating to the **latest stable build**. If this doesn't resolve the problem, try using the **preview build** (also known as the "nightly build").

If you would like to use DuckDB with an open pull request applied to the codebase, you can try building it from source.

Search for Existing Issues

There is a chance that someone else already reported the bug that causes the crash. Please search in the GitHub issue tracker for the error message to see potentially related issues. DuckDB has a large community and there may be some suggestions for a workaround.

Disabling the Query Optimizer

Some crashes are caused by DuckDB's query optimizer component. To identify whether the optimizer is causing the crash, try to turn it off and re-run the query:

```
PRAGMA disable_optimizer;
```

If the query finishes successfully, then the crash was caused by one or more optimizer rules. To pinpoint the specific rules that caused the crash, you can try to selectively disable optimizer rules. This way, your query can still benefit from the rest of the optimizer rules.

Try to Isolate the Issue

Some issues are caused by the interplay of different components and extensions, or are specific to certain platforms or client languages. You can often isolate the issue to a smaller problem.

Reproducing in Plain SQL Issues can also occur due to differences in client libraries. To understand whether this is the case, try reproducing the issue using plain SQL queries with the DuckDB CLI client. If you cannot reproduce the issue in the command line client, it is likely related to the client library.

Different Hardware Setup According to our experience, several crashes occur due to faulty hardware (overheating hard drives, overclocked CPUs, etc.). Therefore, it's worth trying another computer to run the same workload.

Decomposing the Query It's a good idea to try to break down the query into multiple smaller queries with each using a separate DuckDB extension and SQL feature.

For example, if you have a query that targets a dataset in an AWS S3 bucket and performs two joins on it, try to rewrite it as a series of smaller steps as follows. Download the dataset's files manually and load them into DuckDB. Then perform the first join and the second join separately. If the multi-step approach still exhibits the crash at some step, then the query that triggers the crash is a good basis for a minimal reproducible example. If the multi-step approach works and the multi-step process no longer crashes, try to reconstruct the original query and observe which step reintroduces the error. In both cases, you will have a better understanding of what is causing the issue and potentially also a workaround that you can use right away. In any case, please consider submitting an issue with your findings.

Submitting an Issue

If you found a crash in DuckDB, please consider submitting an issue in our GitHub issue tracker with a minimal reproducible example.

Out of Memory Errors

DuckDB has a state of the art out-of-core query engine that can spill to disk for larger-than-memory processing. We continuously strive to improve DuckDB's scalability and prevent out of memory errors whenever possible. That said, you may still experience out-of-memory errors if you run queries with multiple blocking operators, certain aggregate functions (such as `string_agg()` or aggregates with an `ORDER BY` clause), or if you have very little available memory compared to the dataset size.

Types of "Out of Memory" Errors

Out of memory errors mainly occur in two forms:

`OutOfMemoryException`

Most of the time DuckDB runs out of memory with an `OutOfMemoryException` followed by a message from the memory allocator. For example:

```
Out of Memory Error: failed to pin block of size 256.0 KiB (476.7 MiB/476.8 MiB used)
```

OOM Reaper (Linux)

Many Linux distributions have an OOM killer or OOM reaper process whose goal is to prevent memory overcommitment. If the OOM reaper killed your process, you often see the following message where DuckDB was running:

```
Killed
```

To get more detailed information, check the diagnostic messages using the `dmesg` command (you may need `sudo`):

```
sudo dmesg
```

If the process was killed by the OOM killer/reaper, you will find an entry like this:

```
[Fri Apr 18 02:04:10 2025] Out of memory: Killed process 54400 (duckdb) total-vm:1037911068kB,
anon-rss:770031964kB, file-rss:0kB, shmem-rss:0kB, UID:1000 pgtables:1814612kB oom_score_adj:0
```

Troubleshooting Out-of-Memory Errors

To prevent out-of-memory errors, please consult the "Out-of-Memory" issues page of the Performance Guide.

See Also

For more information on DuckDB's memory management, see the "Memory Management in DuckDB" blog post.

Glossary of Terms

This page contains a glossary of a few common terms used in DuckDB.

Terms

In-Process Database Management System

The DBMS runs in the client application's process instead of running as a separate process, which is common in the traditional client–server setup. An alternative term is **embeddable** database management system. In general, the term *"embedded database management system"* should be avoided, as it can be confused with DBMSs targeting *embedded systems* (which run on e.g., microcontrollers).

Replacement Scan

In DuckDB, replacement scans are used when a table name used by a query does not exist in the catalog. These scans can substitute another data source instead of the table. Using replacement scans allows DuckDB to, e.g., seamlessly read Pandas DataFrames or read input data from remote sources without explicitly invoking the functions that perform this (e.g., reading Parquet files from https). For details, see the C API – Replacement Scans page.

Extension

DuckDB has a flexible extension mechanism that allows for dynamically loading extensions. These may extend DuckDB's functionality by providing support for additional file formats, introducing new types, and domain-specific functionality. For details, see the Extensions page.

Platform

The platform is a combination of the operating system (e.g., Linux, macOS, Windows), system architecture (e.g., AMD64, ARM64), and, optionally, the compiler used (e.g., GCC4). Platforms are used to distribute DuckDB binaries and extension packages.

Browsing Offline

You can browse the DuckDB documentation offline in the following formats:

- Single Markdown file (approx. 6 MB)
- PDF file (approx. 40 MB)

Operations Manual

Overview

We designed DuckDB to be easy to deploy and operate. We believe that most users do not need to consult the pages of the operations manual. However, there are certain setups – e.g., when DuckDB is running in mission-critical infrastructure – where we would like to offer advice on how to configure DuckDB. The operations manual contains advice for these cases and also offers convenient configuration snippets such as Gitignore files.

For advice on getting the best performance from DuckDB, see also the Performance Guide.

Migrating to DuckDB 2.0

DuckDB 2.0.0 is packed with new features and performance improvements, including query optimization and asynchronous I/O. It also ships a small set of breaking changes, most notably a new default storage format, which DuckDB versions before 2.0.0 cannot read, and the next step of the lambda syntax transition. This page provides guidance on migrating existing deployments to DuckDB 2.0.0.

Before You Migrate

Before migrating to DuckDB 2.0.0, assess whether your workload is mission-critical. For such workloads, consider waiting for upcoming patch releases, which address issues that surface shortly after a major release. To help facilitate this, the DuckDB development team extended support for DuckDB 1.5 until March 2027. Note that DuckDB 1.4 LTS reaches its end of life on November 17, 2026.

> **Best practice.** In all cases, we recommend setting up a shadow system that tracks your existing workload. This lets you test DuckDB 2.0.0 in an environment outside of your production system before switching over.

For simple deployments, migrating is straightforward. Complex deployments, especially those that run many DuckDB processes, require more care. We expect the most common source of issues to be the storage format, which is described first below.

Breaking Changes

Storage Format

DuckDB versions before 2.0.0 *cannot read* database files that use the DuckDB 2.0.0 storage format. By default, every new database file created by v2.0.0 uses this format (storage version 69).

The other direction keeps working: DuckDB 2.0.0 can read and write existing database files created by earlier versions. These files keep their storage version when v2.0.0 writes to them, so they remain readable by the earlier version.

For example, creating a database with v2.0.0 and then opening it with v1.5.6 results in an error:

```
~/.duckdb/cli/latest/duckdb my.db -c "CREATE TABLE t (x INT);"
~/.duckdb/cli/1.5.6/duckdb my.db
```

```
IO Error: Trying to read a database file with version number 69, but we can only read versions between 64 and 68.
The database file was created with a newer version of DuckDB.

Newer DuckDB version might introduce backward incompatible changes (possibly guarded by compatibility settings).
See the storage page for migration strategy and more information: https://duckdb.org/internals/storage
```

> **Warning.** In a deployment with many DuckDB processes running different versions, a database file created by a v2.0.0 process is unreadable by the remaining older processes. Roll out the upgrade carefully and verify that every process that touches a shared database file is on a compatible version.

Writing Files for Older Versions If you need database files that remain readable by an older DuckDB version, write them with an explicit storage version using the `-storage-version` command line argument:

```
~/.duckdb/cli/latest/duckdb my.db -storage-version v1.5.6 -c "CREATE TABLE t (x INT);"
~/.duckdb/cli/1.5.6/duckdb my.db
```

The storage version can also be set with the `storage_compatibility_version` configuration option or the `STORAGE_VERSION` option of the ATTACH statement. For the full set of methods and the mapping between DuckDB versions and storage versions, see the Storage Versions and Format page.

Converting Existing Databases To move an existing database file to the v2.0.0 storage format, open it with the older DuckDB version and export it, then import it with v2.0.0:

```
/older/duckdb mydata.old.db -c "EXPORT DATABASE 'tmp'"
/newer/duckdb mydata.new.db -c "IMPORT DATABASE 'tmp'"
```

For details on this and other conversion methods, including `COPY FROM DATABASE`, see How to Move between Storage Formats.

Lambda Syntax

DuckDB 2.0.0 takes the next step in the lambda syntax transition that began in v1.3. The old single arrow syntax (`x -> x + 1`) is now disabled by default in favor of the Python-style syntax (`lambda x : x + 1`).

```
-- disabled by default in v2.0.0
SELECT list_transform([1, 2, 3], x -> x + 1);
-- use the Python-style syntax instead
SELECT list_transform([1, 2, 3], lambda x : x + 1);
```

If you cannot update your queries yet, you can re-enable the old syntax for the session:

```
SET lambda_syntax = 'enable_single_arrow';
```

This flag is a temporary bridge: DuckDB 2.1 will remove it and the single arrow syntax entirely. For details, see Deprecated Syntax.

SQL Parser

DuckDB 2.0.0 replaced the PostgreSQL-derived parser with a new PEG-based parser. It is designed to be compatible with the previous parser, so queries that parsed before should continue to parse, and error messages now include more precise source locations. If a query that parsed on an earlier version fails on v2.0.0, please file an issue. For details, see the PEG parser page.

A few expressions deliberately behave differently in v2.0.0, which can silently change query results:

- The `~` and `!~` operators now perform partial regular expression matching, following PostgreSQL: `'abc' ~ 'b'` now returns `true` (previously `false`). The deprecated `regex_match_operator_semantics` setting restores full matching: `SET regex_match_operator_semantics = 'full';`{:.language-sql .highlight}
- The factorial operator `!` binds more tightly: `2 * 3!` now returns `12` (previously `720`). Use parentheses to get the old behavior, e.g., `(2 * 3)!`.
- `X'…'` and `B'…'` literals now produce `BLOB` and `BITSTRING` values instead of the VARCHAR strings `'x…'` and `'b…'`.

If you would like to revert to the old parser, use the `legacy_parser` extension. To install and activate it, run the following commands:

```
INSTALL AND LOAD legacy_parser;
SET enable_legacy_parser = true;
```

C API v2

DuckDB 2.0.0 ships a revamped C API, which is considerably more powerful than its predecessor and exposes components such as file systems and table functions to extensions. The C API v2 is not yet stable: its interface may change in upcoming releases. For background on extension stability levels, see Versioning of Extensions; for the C API reference, see the C API page.

ICU Removed from the Core Build

DuckDB 2.0.0 removed the dependency on the ICU library. Time zones, calendars and collations are now implemented natively by the `icu` extension, and will continue to behave as before. No query changes are required, but if you ship a custom build, note that ICU is no longer bundled. If you find that ICU's behavior changed, please file an issue.

Division by Zero

Integer division (`//`) and modulo (`%`) by zero, e.g., `1 // 0` and `1 % 0`, now throw an error instead of returning `NULL`:

```
Invalid Input Error: Division by zero in expression (1 // 0). Use TRY(...) to return NULL for this
expression, or SET error_on_division_by_zero=false to return NULL for all divisions by zero.
```

To get `NULL` for a single expression, wrap it in `TRY`, e.g., `TRY(1 // 0)`. Regular division (`1 / 0`) still returns `inf`.

`MERGE INTO` with Multiple Matching Source Rows

A `MERGE INTO` statement that would update or delete the same target row more than once, because the row matches multiple source rows, now throws an error instead of silently using one of the source rows. Deduplicate the source rows or make the `ON` condition unique.

Untyped `NULL` Columns

An untyped `NULL` now keeps the `NULL` type instead of defaulting to `INTEGER`. For example, `CREATE TABLE t AS SELECT NULL AS x` creates a column of type `"NULL"`, which rejects subsequent inserts of non-`NULL` values. Cast the `NULL` to the intended type, e.g., `NULL::VARCHAR AS x`, or restore the previous behavior with `SET legacy_disable_null_type = true;`{:.language-sql .highlight}.

Identifier Case Setting

The `preserve_identifier_case` setting changed from a `BOOLEAN` to a `VARCHAR` with the values `'preserve_case'` (default), `'lowercase'` and `'uppercase'`. The old values `true` and `false` are still accepted and map to `'preserve_case'` and `'lowercase'`, respectively.

S3 URL Parameters

Passing S3 settings as URL parameters (e.g., `s3://bucket/file.csv?s3_access_key_id=...`) is disabled by default: the `?` and the following text are now treated as part of the object key. Use secrets instead or, if you need the legacy behavior, opt in with `SET s3_enable_deprecated_url_parameters = true` (available after loading the `httpfs` extension).

Removed and Deprecated Settings

- The `custom_profiling_settings` setting was removed. Use `tracked_metrics` instead, see Profiling.
- The `enable_http_logging` and `http_logging_output` settings were removed. Use `CALL enable_logging('HTTP')` instead, see Logging.
- The `profiling_mode` setting is deprecated and its value has no effect: detailed profiling information is always collected. Use `PRAGMA enable_profiling` to enable profiling.

Log Storage in the CLI

Since v1.5.0, the CLI uses the `shell_log_storage` log storage by default, which prints warnings to the console but cannot be queried. To query logs via `duckdb_logs` in the CLI, enable logging with `CALL enable_logging(storage = 'memory')`. Clients other than the CLI still default to the `memory` log storage.

Client APIs

Some client APIs ship their own breaking changes with DuckDB 2.0. In particular, the Python client requires Python 3.11 or newer and changes several APIs, see Migrating Python Code to DuckDB 2.0.

What's New in v2.0

Beyond the breaking changes above, DuckDB 2.0.0 added a large set of new features. Rather than repeat the feature documentation here, this section points to where each new feature is covered in full. For the complete tour, see the preview announcement and the detailed release announcement. The What's New in the Docs for 2.0 page links every high-level new feature to its documentation, organized by theme:

- DuckDB as a server with the `quack` extension and the `CONNECT` statement
- The `VARIANT` type for semi-structured data
- Triggers that run actions automatically in response to table changes
- SQL dialect additions, including data-modifying CTEs and nested schemas
- Asynchronous I/O for faster reads from object storage
- Optimizer and engine improvements that speed up existing queries, including on Windows
- Self-hostable extension repositories for signing and distributing your own extensions

Several of the headline v2.0.0 changes are covered in Breaking Changes above: the new default storage format, the PEG-based SQL parser, the revamped C API v2 and the native timezone, calendar and collation support left after ICU was removed from the core build.

See Also

- Storage Versions and Format
- Operations Manual

Backup Extension Repositories

The core and community extension repositories have backup servers (see duckdb-backup.org):

Repository	Primary server	Backup server
`core`	`http://extensions.duckdb.org`	`http://extensions.duckdb-backup.org`
`community`	`http://community-extensions.duckdb.org`	`http://community-extensions.duckdb-backup.org`

The backup servers serve byte-identical binaries, so signature checks are unaffected.

Automatic Fallback

Fallback is automatic: if the primary server is unreachable or returns a 5xx error, `INSTALL` retries the download from the backup server. No configuration is needed.

- A 4xx error (e.g., a non-existent extension) is reported directly, without contacting the backup server.
- The fallback only applies to the built-in `core` and `community` repositories. Repositories set via `custom_extension_repository` or `INSTALL ... FROM '<url>'` are used as-is.
- The installed extension's origin is still recorded as `core` or `community`.

Using the Backup Server Manually

To always install core extensions from the backup server, run:

```
SET custom_extension_repository = 'http://extensions.duckdb-backup.org';
```

DuckDB's Footprint

Files Created by DuckDB

DuckDB creates several files and directories on disk. This page lists both the global and the local ones.

Global Files and Directories

DuckDB creates the following global files and directories in the user's home directory (denoted with ~):

Location	Description	Shared between versions	Shared between clients
`~/.duckdbrc`	The content of this file is executed when starting the DuckDB CLI client. The commands can be both dot command and SQL statements. The naming of this file follows the `~/.bashrc` and `~/.zshrc` "run commands" files.	Yes	Only used by CLI
`~/.duckdb_history`	History file, similar to `~/.bash_history` and `~/.zsh_history`. Used by the DuckDB CLI client.	Yes	Only used by CLI
`~/.duckdb/extensions`	Binaries of installed extensions.	No	Yes
`~/.duckdb/stored_secrets`	Persistent secrets created by the Secrets manager.	Yes	Yes
`~/.duckdb/extension_repositories`	Definitions of custom extension repositories, including their pinned public keys, stored as one JSON file per repository. Created by `CREATE EXTENSION REPOSITORY`.	Yes (v2.0 and later)	Yes

Local Files and Directories

DuckDB creates the following files and directories in the working directory (for in-memory connections) or relative to the database file (for persistent connections):

Name	Description	Example
`<database_filename>`{:.language-sql .highlight}	Database file. Only created in on-disk mode. The file can have any extension with typical extensions being `.duckdb`, `.db` and `.ddb`.	`weather.duckdb`
`.tmp/`	Temporary directory. Only created in in-memory mode.	`.tmp/`
`<database_filename>.tmp/`{:.language-sql .highlight}	Temporary directory. Only created in on-disk mode.	`weather.duckdb.tmp`

Name	Description	Example
`<database_filename>.wal`{:.language-sql .highlight}	Write-ahead log file. If DuckDB exits normally, the WAL file is deleted upon exit. If DuckDB crashes, the WAL file is required to recover data.	`weather.duckdb.wa`

If you are working in a Git repository and would like to disable tracking these files by Git, see the instructions on using `.gitignore` for DuckDB.

Gitignore for DuckDB

If you work in a Git repository, you may want to configure your Gitignore to disable tracking files created by DuckDB. These potentially include the DuckDB database, write-ahead log, and temporary files.

Sample Gitignore Files

In the following, we present sample Gitignore configuration snippets for DuckDB.

Ignore Temporary Files but Keep Database

This configuration is useful if you would like to keep the database file in the version control system:

```
*.wal
*.tmp/
```

Ignore Database and Temporary Files

If you would like to ignore both the database and the temporary files, extend the Gitignore file to include the database file. The exact Gitignore configuration to achieve this depends on the extension you use for your DuckDB databases (`.duckdb`, `.db`, `.ddb`, etc.). For example, if your DuckDB files use the `.duckdb` extension, add the following lines to your `.gitignore` file:

```
*.duckdb*
*.wal
*.tmp/
```

Reclaiming Space

DuckDB uses a single-file format, which has some inherent limitations w.r.t. reclaiming disk space.

`CHECKPOINT`

To reclaim space after deleting rows, use the `CHECKPOINT` statement.

`VACUUM`

The `VACUUM` statement does *not* trigger vacuuming deletes and hence does not reclaim space.

Compacting a Database by Copying

To compact the database, you can create a fresh copy of the database using the `COPY FROM DATABASE` statement. In the following example, we first connect to the original database `db1`, then the new (empty) database `db2`. Then, we copy the content of `db1` to `db2`.

```
ATTACH 'db1.db' AS db1;
ATTACH 'db2.db' AS db2;
COPY FROM DATABASE db1 TO db2;
```

Installing DuckDB

Install Script

You can install the DuckDB CLI client using an install script.

Linux and macOS

To use the DuckDB install script on Linux and macOS, run:

```
curl https://install.duckdb.org | bash
```

By default, this installs the latest stable version of DuckDB to `~/.duckdb/cli/latest/duckdb`. To add the DuckDB binary to your path, append the following line to your shell profile or RC file (e.g., `~/.bashrc`, `~/.zshrc`):

```
export PATH="~/.duckdb/cli/latest":$PATH
```

You can install past DuckDB releases (all the way back to v1.0.0) using the `DUCKDB_VERSION` variable. For example, to install v1.2.2, run:

```
curl https://install.duckdb.org | DUCKDB_VERSION=1.2.2 bash
```

Installing the C/C++ Libraries

Starting with DuckDB 2.0, the install script can also install the DuckDB C/C++ headers and libraries using the `DUCKDB_INSTALL` variable. It takes a comma-separated list of components: `cli` (default), `static` and `shared`. For example, to install the static library and the CLI, run:

```
curl https://install.duckdb.org | DUCKDB_INSTALL=static,cli bash
```

The libraries and headers are installed to `~/.duckdb/lib/<version>/release_<platform>`, e.g., `~/.duckdb/lib/2.0.0/release_osx_arm64`. For more details, see Linking Extensions.

Windows

On Windows, we provide a beta install script. To use it, run the following command from the Run menu (Win+R) or the Command Prompt:

```
powershell -NoExit -ExecutionPolicy Bypass -Command "irm https://install.duckdb.org/install.ps1 | iex"
```

Requirements

DuckDB is designed to be portable and has very few runtime dependencies. Still, there are a few requirements to be aware of before installing or running DuckDB.

Supported Platforms

DuckDB officially supports Linux, macOS and Windows on both x86_64 (AMD64) and AArch64 (ARM64) CPU architectures. Official binaries are distributed for these platforms for both the latest stable version and the preview version (nightly build).

Platform name	Operating system	Architecture	CPU types	Support level
`linux_amd64`	Linux	x86_64 (AMD64)	Intel, AMD, etc.	Full
`linux_arm64`	Linux	AArch64 (ARM64)	AWS Graviton, Snapdragon, etc.	Full
`osx_amd64`	macOS 12+	x86_64 (AMD64)	Intel	Partial
`osx_arm64`	macOS 12+	AArch64 (ARM64)	Apple Silicon M1, M2, etc.	Full
`windows_amd64`	Windows 10+	x86_64 (AMD64)	Intel, AMD, etc.	Full
`windows_arm64`	Windows 10+	AArch64 (ARM64)	Copilot+ PC with Qualcomm CPU	Full

> **Warning.** macOS on Intel CPUs (`osx_amd64`) is only partially supported. DuckDB binaries and core extensions are still distributed for this platform but, starting with DuckDB 2.0.0, community extensions are no longer built for it. DuckDB 2.1 will stop supporting `osx_amd64`.

DuckDB can also run on other platforms, such as Linux distributions using musl libc (e.g., Alpine Linux), Android and WebAssembly, with varying levels of support. For details, see the list of platforms on the "Building DuckDB" page. For the platforms extensions are distributed for, see the "Extension Distribution" page.

Linux

DuckDB's official Linux binaries require glibc 2.28 or newer. Since DuckDB 1.3.0, the official Linux binaries are built using the `manylinux_2_28` image, which combines an older glibc with a newer compiler. As a result, extensions are no longer distributed for the `linux_amd64_gcc4` platform.

Most mainstream Linux distributions ship with glibc 2.28 or newer, so no action is needed on these systems. If you need to run DuckDB on a system with an older glibc, you can build DuckDB from source against that version.

Logging

Logging

DuckDB implements a logging mechanism that provides users with detailed information about events such as query execution, performance metrics and system events.

> **Warning.** Logs may contain sensitive information, including secrets and credentials. Enable logging with care and consider where logs are stored and who can access them. Review their contents before sharing them.

Basics

The DuckDB logging mechanism can be enabled or disabled using a special function, `enable_logging`. Logs are stored in a special view named `duckdb_logs`, which can be queried like any standard table.

Example:

```
CALL enable_logging(storage = 'memory');
-- Run some queries...
SELECT * FROM duckdb_logs;
```

> In the CLI client, the default log storage is `shell_log_storage`, which prints log messages to the console but does not support querying. Therefore, the example above passes `storage = 'memory'`. In other clients, the default log storage is `memory`, so `CALL enable_logging();` suffices. See Log Storages for details.

To disable logging, run

```
CALL disable_logging();
```

To clear the current log, run

```
CALL truncate_duckdb_logs();
```

Log Level

DuckDB supports different logging levels that control the verbosity of the logs:

- `FATAL`: Only logs fatal errors
- `ERROR`: Logs error messages
- `WARNING`: Logs warnings and errors
- `INFO`: Logs general information, warnings and errors (default)
- `DEBUG`: Logs detailed debugging information
- `TRACE`: Logs very detailed tracing information

The log level can be set using:

```
CALL enable_logging(level = 'debug');
```

Warnings

DuckDB emits warnings, for example when you use deprecated syntax or a deprecated setting, through the logging system at the `WARNING` level. In the CLI, warnings are printed by default. To capture them from any client, or to inspect them after the fact, enable logging and query the `duckdb_logs` view, filtering on the log level:

```
CALL enable_logging(level = 'warning', storage = 'memory');
-- Run some queries that may emit warnings...
SELECT timestamp, message
FROM duckdb_logs
WHERE log_level = 'WARNING';
```

The default log level is `INFO`, which already includes warnings, so calling `enable_logging()` without a `level` also captures warnings. Setting `level = 'warning'` excludes lower-severity `INFO` messages.

> **Note.** Switching the CLI to a log storage other than `shell_log_storage` means that warnings are no longer printed to the console. To restore the default behavior, run `CALL enable_logging(storage = 'shell_log_storage')`.

Log Types

In DuckDB, log messages can have an associated log type. Log types allow two main things:

- Fine-grained control over log message generation
- Support for structured logging

Logging-Specific Types

To log only messages of a specific type:

```
CALL enable_logging('HTTP');
```

The above function will automatically set the correct log level and will add the `HTTP` type to the `enabled_log_types` settings. This ensures only log messages of the `HTTP` type will be written to the log.

> New As of DuckDB 2.0, the dedicated `enable_http_logging` and `http_logging_output` settings have been removed. Use the unified logging framework shown here (`CALL enable_logging('HTTP')` together with `duckdb_logs_parsed('HTTP')`) instead. The `redact_http_logs` setting (default `true`) controls whether HTTP header values are redacted in the logs.

To enable multiple log types, simply pass:

```
CALL enable_logging(['HTTP', 'QueryLog']);
```

Structured Logging

Some log types like HTTP will have an associated message schema. To make DuckDB automatically parse the message, use the `duckdb_logs_parsed()` macro. For example:

```
SELECT request.headers FROM duckdb_logs_parsed('HTTP');
```

To view the schema of each structure log type simply run:

```
DESCRIBE FROM duckdb_logs_parsed('HTTP');
```

List of Available Log Types

This is a (non-exhaustive) list of the available log types in DuckDB.

Log Type	Description	Structured
`QueryLog`	Logs which queries are executed in DuckDB	No
`FileSystem`	Logs all FileSystem interaction with DuckDB's Filesystem	Yes
`HTTP`	Logs all HTTP traffic from DuckDB's internal HTTP client	Yes
`PhysicalOperator`	Logs events emitted by physical operators during query execution	Yes
`Metrics`	Logs profiling metrics collected during query execution	Yes

The structured log types expose the following schemas, which you can inspect at any time with `DESCRIBE FROM duckdb_logs_parsed(<log_type>)`{:.language-sql .highlight}:

Log Type	Schema
`FileSystem`	`fs VARCHAR, path VARCHAR, op VARCHAR, bytes BIGINT, pos BIGINT`
`HTTP`	`request STRUCT(type, url, start_time, duration_ms, headers MAP), response STRUCT(status, reason, headers MAP)`
`PhysicalOperator`	`operator_type VARCHAR, parameters MAP(VARCHAR, VARCHAR), class VARCHAR, event VARCHAR, info MAP(VARCHAR, VARCHAR)`
`Metrics`	`metric VARCHAR, value VARCHAR`

Log Storages

By default, DuckDB logs to an in-memory log storage (`memory`), except in the CLI client, which uses `shell_log_storage`. DuckDB supports different types of log storage. Currently, the following log storage types are implemented in core DuckDB.

Log storage	Description
`memory`	Log to an in-memory buffer (default in clients other than the CLI)
`stdout`	Log to the stdout of the current process (in CSV format)
`file`	Log to (a) csv file(s)
`shell_log_storage`	Print log messages to the console (default in the CLI)

The `shell_log_storage` cannot be queried: `duckdb_logs`, `duckdb_logs_parsed()` and `truncate_duckdb_logs()` all return an error. It also does not accept configuration options such as `storage_config` or `storage_buffer_size`. To query logs in the CLI, switch to a queryable storage such as `memory` or `file`:

```
CALL enable_logging(storage = 'memory');
```

Note that the `duckdb_logs` view is automatically updated to target the currently active log storage. This means that switching the log storage may influence what is returned by the `duckdb_logs` function.

Logging to stdout

```
CALL enable_logging(storage = 'stdout');
```

Logging to File

```
CALL enable_logging(storage = 'file', storage_config = {'path': 'path/to/store/logs'});
```

or using the equivalent shorthand:

```
CALL enable_logging(storage_path = 'path/to/store/logs');
```

Advanced Usage

Normalized vs. Denormalized Logging

DuckDB's log storages can log in two ways: normalized vs. denormalized.

In denormalized logging, the log context information is appended directly to each log entry, while in normalized logging the log entries are stored separately with `context_id`s referencing the context information.

Log storage	Normalized
`memory`	yes
`file`	configurable
`stdout`	no

For file storage, you can switch between normalized and denormalized by providing a path to a directory (for normalized) or a path ending in `.csv` (for denormalized). For file logging, denormalized is generally recommended since this increases performance and reduces the total size of the logs. To configure normalization of `file` log storage:

```
-- normalized: creates `/tmp/duckdb_log_contexts.csv` and `/tmp/duckdb_log_entries.csv`
CALL enable_logging(storage_path = '/tmp');
-- denormalized: creates `/tmp/logs.csv`
CALL enable_logging(storage_path = '/tmp/logs.csv');
```

Note that the difference between normalized and denormalized is typically hidden from users through the `duckdb_logs` function, which automatically joins normalized tables into a single unified result. To illustrate, both configurations above will be queryable using `FROM duckdb_logs;` and will produce identical results.

Buffer Size

The log storage in DuckDB implements a buffering mechanism to optimize logging performance. This implementation introduces a potential delay between message logging and storage writing. This delay can obscure the actual message writing time, which is particularly problematic when debugging crashes, as messages generated immediately before a crash might not be written. To address this, the buffer size can be configured as follows:

```
CALL enable_logging(storage = 'memory', storage_config = {'buffer_size': 0});
```

or using the equivalent shorthand:

```
CALL enable_logging(storage = 'memory', storage_buffer_size = 0);
```

Note that the default buffer size is different for different log storages:

Log storage	Default buffer size
`memory`	`STANDARD_VECTOR_SIZE` (2048)
`file`	`STANDARD_VECTOR_SIZE` (2048)
`stdout`	Disabled (0)

So for example, if you want to increase your `stdout` logging performance, simply enable buffering to greatly (>10x) speed up your logging:

```
CALL enable_logging(storage = 'stdout', storage_buffer_size = 2048);
```

Or imagine you are debugging a crash in DuckDB and you want to use the `file` logger to understand what's going on. Simply disable the buffering using:

```
CALL enable_logging(storage_path = '/tmp/mylogs', storage_buffer_size = 0);
```

Syntactic Sugar

DuckDB contains some syntactic sugar to make common paths easier. For example, the following statements are all equal:

```
-- regular invocation
CALL enable_logging(storage = 'file', storage_config = {'path': 'path/to/store/logs'});
-- using shorthand for common path storage config param
CALL enable_logging(storage = 'file', storage_path = 'path/to/store/logs');
-- omitting `storage = 'file'` -> is implied from presence of `storage_config`
CALL enable_logging(storage_config = {'path': 'path/to/store/logs'});
```

HTTP User-Agent

HTTP User-Agent

Core DuckDB sets the default user-agent as follows:

```
duckdb/v2.0.0(osx_arm64) cli
```

which indicates the DuckDB version, the platform and the client API (the value of the `duckdb_api` setting, e.g., `cli`, `capi` or `cpp`). If the `custom_user_agent` setting is set, its value is appended to the user-agent string. Note that `custom_user_agent` can only be set when starting DuckDB (e.g., via the connection configuration), not while the database is running, see Configuration. The currently generated user-agent string can be seen via `PRAGMA user_agent;`, see Configuration/Pragmas.

In addition, some extensions set their own user agents; notable examples here include the following.

Extensions

Azure

Azure uses the Azure SDK which sets its own user agents. For identity and storage calls you may see respectively strings like these:

- via Azure Identity: `azsdk-cpp-identity/1.11.0 (Darwin 25.2.0 arm64 Darwin Kernel Version 25.2.0: Tue Nov 18 21:07:05 PST 2025; root:xnu-12377.61.12~1/RELEASE_ARM64_T6020 Cpp/201402)`
- via Azure Blob/ADLSv2: `azsdk-cpp-storage-blobs/12.15.0 (Darwin 25.2.0 arm64 Darwin Kernel Version 25.2.0: Tue Nov 18 21:07:05 PST 2025; root:xnu-12377.61.12~1/RELEASE_ARM64_T6020 Cpp/201402)`

Delta (and Unity Catalog)

The Delta extension employs calls from DuckDB core, tagged as the DuckDB default above, and also has calls originating from the Delta Kernel, which may look like:

- `object_store/0.12.5`

Unity Catalog calls also use a mix of DuckDB default user-agents, and the Delta style agent above.

HTTPFS - HTTPS/S3

Calls via HTTPFS the extension use the DuckDB default strings noted above.

Securing DuckDB

Securing DuckDB

DuckDB is a powerful analytical database engine. It can read and write files, access the network, load extensions, and use system resources. Like any powerful tool, these capabilities require appropriate configuration when working with sensitive data or in shared environments.

This page documents DuckDB's security model and security-related settings. The right configuration depends on your use case, environment, and threat model. If you plan to embed DuckDB in your application, also consult the "Embedding DuckDB" page.

Untrusted Input

Untrusted SQL Input

> **Warning.** Treat SQL in DuckDB like code in Bash or Python. Do not execute SQL from untrusted sources without proper sandboxing.

DuckDB executes SQL with the full privileges of the user running it, much like a shell or scripting interpreter (such as bash or Python). Just as you wouldn't run an untrusted shell script or Python program without sandboxing, apply the same caution to SQL in DuckDB.

If your application must execute SQL from untrusted sources, use additional safeguards when running untrusted code such as:

- Use duckdb-wasm for sandboxing
- Run DuckDB in an isolated container (e.g., Docker with restricted capabilities)
- Use a virtual machine or separate process with minimal privileges
- Apply operating system-level sandboxing
- Use network isolation to prevent data exfiltration
- Implement strict query timeouts at the application level

The settings described on this page provide **defense-in-depth** and can limit certain capabilities, but they are not a substitute for proper sandboxing. Also keep in mind that sandboxing should not just be considered for security purposes, but also for preventing denial of service (DoS) attacks: malicious inputs can easily cause DuckDB to consume excessive resources such as memory, disk, CPU, or network.

Untrusted Non-SQL Input

> **Warning.** Even non-SQL input into DuckDB can easily have unintended consequences. When building security-sensitive applications with DuckDB, always make sure you properly understand the impact of feeding untrusted input into DuckDB.

Besides SQL, DuckDB also has several non-SQL APIs that can be used to interact with the database. For example, there is a relational API in Python that allows building queries programmatically.

These APIs accept user input such as file paths, table names, column names, and filter expressions. While they don't execute raw SQL strings, they still trigger DuckDB operations that can read files, access the network, and use system resources.

Example considerations for non-SQL APIs:

- **File paths:** Functions like `duckdb.read_csv(path)` or `duckdb.read_parquet(path)` accept file paths. An attacker-controlled path could read sensitive files (e.g., `/etc/passwd`) or access remote URLs.
- **Table and column names:** While these are typically identifiers rather than executable code, unsanitized input could lead to unexpected behavior or information disclosure.

- **Filter expressions:** Some APIs accept filter expressions that are compiled into DuckDB expressions, which often support subqueries containing arbitrary SQL. Treat these with the same caution as SQL.

Recommendations:

- Validate and sanitize all user-provided inputs before passing them to DuckDB APIs.
- Apply the same sandboxing principles as for untrusted SQL when accepting input from untrusted sources.
- Properly read the documentation of all used functions to ensure you understand whether a function is safe to use with untrusted input under your specific use case.

Extensions

DuckDB has a flexible extension mechanism that adds functionality such as new file formats, functions, and remote file system access. Extensions run with the same privileges as the DuckDB process itself, so they warrant careful consideration in security-sensitive environments.

Autoloading

DuckDB can automatically load core extensions when certain SQL statements require them. To maintain full control over which extensions are loaded, you can disable autoloading:

```
SET autoload_known_extensions = false;
SET autoinstall_known_extensions = false;
```

Core vs. Community Extensions

DuckDB extensions fall into two categories:

- **Core extensions:** Maintained by the DuckDB team with full support. These include extensions like `parquet`, `json`, and `httpfs`.
- **Community extensions:** Contributed by third parties and installed via `INSTALL extension_name FROM community`. These are not maintained by the DuckDB team, so only install community extensions from sources you trust.

To disable community extensions entirely:

```
SET allow_community_extensions = false;
```

Serving DuckDB over the Network

With the quack extension, a DuckDB process can act as a server (`CALL quack_serve(...)` or `duckdb -serve`). A Quack server exposes the full SQL surface of the instance, including read and write access to every table its session can see, to any client holding the token. The server binds to `localhost` by default; for remote access, put it behind a TLS-terminating reverse proxy instead of using `allow_other_hostname`. Note that `duckdb -serve` stores its token in a persistent (unencrypted) `quack` secret. For authentication and authorization hooks, see the Quack security page.

Reporting Vulnerabilities

If you discover a potential vulnerability, please report it confidentially via GitHub.

Settings to Limit DuckDB's Capabilities

The settings documented in this section provide additional hardening for DuckDB deployments. However, they should not be relied upon as comprehensive security mechanisms in all configurations. These settings are designed as defense-in-depth measures to limit the impact of potential security issues, but they cannot provide complete protection against all attack vectors, especially when executing untrusted SQL. For robust security when dealing with untrusted input, combine these settings with proper sandboxing at the operating system or container level, as described in the "Untrusted SQL Input" section.

Safe Mode (CLI)

DuckDB's CLI client supports "safe mode", which prevents DuckDB from accessing external files other than the database file. This can be activated via a command line argument or a dot command:

```
duckdb -safe ...
```

```
.safe_mode
```

Restricting File Access

DuckDB can list directories and read arbitrary files via its CSV parser's `read_csv` function or read text via the `read_text` function. This makes it possible to read from the local file system, for example:

```
SELECT *
FROM read_csv('/etc/passwd', sep = ':');
```

Disabling File Access File access can be disabled in two ways. First, you can disable individual file systems. For example:

```
SET disabled_filesystems = 'LocalFileSystem';
```

Second, you can also completely disable external access by setting the `enable_external_access` option option to `false`.

```
SET enable_external_access = false;
```

This setting implies that:

- ATTACH cannot attach to a database in a file.
- COPY cannot read from or write to files.
- Functions such as `read_csv`, `read_parquet`, `read_json`, etc. cannot read from an external source.

The `allowed_directories` and `allowed_paths` Options You can restrict DuckDB's access to certain directories or files using the `allowed_directories` and `allowed_paths` options (respectively). These options allow fine-grained access control for the file system. For example, you can set DuckDB to only use the `/tmp` directory.

```
SET allowed_directories = ['/tmp'];
SET enable_external_access = false;
FROM read_csv('test.csv');
```

With the setting applied, DuckDB will refuse to read files in the current working directory:

```
Permission Error:
Cannot access file "test.csv" - file system operations are disabled by configuration
```

Locking Configurations

Security-related configuration settings generally lock themselves for safety reasons. For example, while we can disable community extensions using the SET `allow_community_extensions = false`{:.language-sql .highlight}, we cannot re-enable them again after the fact without restarting the database. Trying to do so will result in an error:

```
Invalid Input Error:
Cannot change allow_community_extensions setting while database is running
```

This prevents re-enabling settings that were explicitly disabled.

Nevertheless, many configuration settings do not disable themselves, such as the resource constraints. If you allow users to run SQL statements unrestricted on your own hardware, you might want to consider locking the configuration after your own configuration has finished using the following command:

```sql
SET lock_configuration = true;
```

This prevents any configuration settings from being modified from that point onwards.

To allow specific settings to remain configurable even when `lock_configuration` is enabled, use the `allowed_configs` option:

```sql
SET allowed_configs = ['memory_limit', 'threads'];
SET lock_configuration = true;
```

With this configuration, `memory_limit` and `threads` can still be changed, while all other settings are locked.

Secrets

Secrets are used to manage credentials to log into third party services like AWS or Azure. DuckDB can show a list of secrets using the `duckdb_secrets()` table function. This will redact any sensitive information such as security keys by default. The `allow_unredacted_secrets` option can be set to show all information contained within a security key. It is recommended not to turn on this option if you are running untrusted SQL input.

> The `allow_unredacted_secrets` option can only be set when the database is instantiated. It cannot be changed at runtime. Setting it with SET after the database is running fails with `Cannot change allow_unredacted_secrets setting while database is running`. Provide it as a startup configuration option instead, for example via the `config` argument when connecting from a client.

Queries can access the secrets defined in the Secrets Manager. For example, if there is a secret defined to authenticate with a user, who has write privileges to a given AWS S3 bucket, queries may write to that bucket. This is applicable for both persistent and temporary secrets.

Persistent secrets are stored in unencrypted binary format on the disk. These have the same permissions as SSH keys, `600`, i.e., only the user who is running the DuckDB (parent) process can read and write them.

Prepared Statements to Prevent SQL Injection

Similarly to other SQL databases, it's recommended to use prepared statements in DuckDB to prevent SQL injection.

> Important Prepared statements protect against SQL injection when **you control the query structure** but accept **untrusted data values** (e.g., user-provided search terms or IDs). If users can supply the SQL query itself, this is equivalent to allowing them to run arbitrary code – see "Untrusted SQL Input".

Therefore, avoid concatenating strings for queries:

```python
import duckdb
duckdb.execute("SELECT * FROM (VALUES (32, 'a'), (42, 'b')) t(x) WHERE x = " + str(42)).fetchall()
```

Instead, use prepared statements:

```python
import duckdb
duckdb.execute("SELECT * FROM (VALUES (32, 'a'), (42, 'b')) t(x) WHERE x = ?", [42]).fetchall()
```

Constrain Resource Usage

DuckDB can use quite a lot of CPU, RAM, and disk space. These resources can be limited to control the usage of the DuckDB instance.

The number of CPU threads that DuckDB can use can be set using, for example:

```
SET threads = 4;
```

Where 4 is the number of allowed threads.

The maximum amount of memory (RAM) can also be limited, for example:

```
SET memory_limit = '4GB';
```

The size of the temporary file directory can be limited with:

```
SET max_temp_directory_size = '4GB';
```

Privileges

Avoid running DuckDB as a root user (e.g., using `sudo`). There is no good reason to run DuckDB as root.

Embedding DuckDB

CLI Client

The Command Line Interface (CLI) client is intended for interactive use cases and not for embedding. As a result, it has more features that could be abused by a malicious actor. For example, the CLI client has the `.sh` feature that allows executing arbitrary shell commands. This feature is only present in the CLI client and not in any other DuckDB clients.

```
.sh ls
```

> **Tip.** Calling DuckDB's CLI client via shell commands is **not recommended** for embedding DuckDB. It is recommended to use one of the client libraries, e.g., Python, R, Java, etc.

Securing Extensions

DuckDB has a powerful extension mechanism, which has the same privileges as the user running DuckDB's (parent) process. This introduces security considerations. Therefore, we recommend reviewing the configuration options listed on this page and setting them according to your attack models.

DuckDB Signature Checks

DuckDB extensions are checked on every load using the signature of the binaries. There are currently three categories of extensions:

- Signed with a `core` key. Only extensions vetted by the core DuckDB team are signed with these keys.
- Signed with a `community` key. These are open-source extensions distributed via the DuckDB Community Extensions repository.
- Unsigned.

Overview of Security Levels for Extensions

DuckDB offers the following security levels for extensions.

Usable extensions	Description	Configuration
`core`	Extensions can only be loaded if signed from a `core` key.	`SET allow_community_extensions = false`{:.language-sql .highlight}
`core` and `community`	Extensions can only be loaded if signed from a `core` or `community` key.	This is the default security level.
Any extension including unsigned	Any extensions can be loaded.	Set `allow_unsigned_extensions = true` at startup (e.g., `duckdb -unsigned` in the CLI). It cannot be changed while the database is running.

Security-related configuration settings lock themselves, i.e., it is only possible to restrict capabilities in the current process.

For example, attempting the following configuration changes will result in an error:

```
SET allow_community_extensions = false;
SET allow_community_extensions = true;
```

```
Invalid Input Error:
Cannot change allow_community_extensions setting while database is running
```

Custom Extension Repositories

DuckDB 2.0 introduced trusted custom extension repositories, which can serve extensions signed with user-provided keys. Two settings control them:

- `allow_extension_repositories`: whether custom trusted extension repositories are `'allowed'`, `'forbidden'` (which also distrusts existing repositories) or `'undecided'` (the default: blocks adding new repositories, but keeps trusting existing ones). While the database is running, the setting can only move from `'undecided'` to `'allowed'` or `'forbidden'`, or from `'allowed'` to `'forbidden'`.
- `extension_repository_directory`: the directory in which trusted extension repositories are stored. This is the trust anchor for user-provided repositories, so while signature checking is enabled (`allow_unsigned_extensions = false`), it can only be set at startup, not while the database is running.

If you run DuckDB in an environment where only extensions signed by the DuckDB team should be loaded, disable custom repositories:

```
SET allow_extension_repositories = 'forbidden';
```

Community Extensions

DuckDB has a Community Extensions repository, which allows convenient installation of third-party extensions. Community extension repositories like pip or npm are essentially enabling remote code execution by design. This is less dramatic than it sounds. For better or worse, we are quite used to piping random scripts from the web into our shells, and routinely install a staggering amount of transitive dependencies without thinking twice. Some repositories like CRAN enforce a human inspection at some point, but that's no guarantee for anything either.

We've studied several different approaches to community extension repositories and have picked what we think is a sensible approach: we do not attempt to review the submissions, but require that the *source code of extensions is available*. We do take over the complete build, sign and distribution process. Note that this is a step up from pip and npm that allow uploading arbitrary binaries but a step down from reviewing everything manually. We allow users to report malicious extensions and show adoption statistics like GitHub stars and download count. Because we manage the repository, we can remove problematic extensions from distribution quickly.

Despite this, installing and loading DuckDB extensions from the community extension repository will execute code written by third party developers, and therefore *can* be dangerous. A malicious developer could create and register a harmless-looking DuckDB extension that steals your crypto coins. If you're running a web service that executes untrusted SQL from users with DuckDB, we recommend disabling community extensions. To do so, run:

```
SET allow_community_extensions = false;
```

Disabling Autoinstalling and Autoloading Known Extensions

By default, DuckDB automatically installs and loads known extensions. To disable autoinstalling known extensions, run:

```
SET autoinstall_known_extensions = false;
```

To disable autoloading known extensions, run:

```
SET autoload_known_extensions = false;
```

To lock this configuration, use the `lock_configuration` option:

```
SET lock_configuration = true;
```

Always Require Signed Extensions

By default, DuckDB requires extensions to be either signed as core extensions (created by the DuckDB developers) or community extensions (created by third-party developers but distributed by the DuckDB developers). The `allow_unsigned_extensions` setting can be enabled on start-up to allow loading unsigned extensions. While this setting is useful for extension development, enabling it will allow DuckDB to load *any extensions,* which means more care must be taken to ensure malicious extensions are not loaded.

Non-Deterministic Behavior

Several operators in DuckDB exhibit non-deterministic behavior. Most notably, SQL uses set semantics, which allows results to be returned in a different order. DuckDB exploits this to improve performance, particularly when performing multi-threaded query execution. Other factors, such as using different compilers, operating systems and hardware architectures, can also cause changes in ordering. This page documents the cases where non-determinism is an *expected behavior*. If you would like to make your queries deterministic, see the "Working Around Non-Determinism" section.

Set Semantics

One of the most common sources of non-determinism is the set semantics used by SQL. E.g., if you run the following query repeatedly, you may get two different results:

```
SELECT *
FROM (
    SELECT 'A' AS x
    UNION
    SELECT 'B' AS x
);
```

Both results A, B and B, A are correct.

Different Results on Different Platforms: `array_distinct`

The `array_distinct` function may return results in a different order on different platforms:

```
SELECT array_distinct(['A', 'A', 'B', NULL, NULL]) AS arr;
```

For this query, both `[A, B]` and `[B, A]` are valid results.

Floating-Point Aggregate Operations with Multi-Threading

Floating-point inaccuracies may produce different results when run in multi-threaded configurations: For example, `stddev` and `corr` may produce non-deterministic results:

```
CREATE TABLE tbl AS
    SELECT 'ABCDEFG'[floor(random() * 7 + 1)::INT] AS s, 3.7 AS x, i AS y
    FROM range(1, 1_000_000) r(i);

SELECT s, stddev(x) AS standard_deviation, corr(x, y) AS correlation
FROM tbl
GROUP BY s
ORDER BY s;
```

The expected standard deviations and correlations from this query are 0 for all values of `s`. However, when executed on multiple threads, the query may return small numbers (`0 <= z < 10e-16`) due to floating-point inaccuracies.

Working Around Non-Determinism

For the majority of use cases, non-determinism is not causing any issues. However, there are some cases where deterministic results are desirable. In these cases, try the following workarounds:

1. Limit the number of threads to prevent non-determinism introduced by multi-threading.

   ```
   SET threads = 1;
   ```

2. Enforce ordering. For example, you can use the `ORDER BY ALL` clause:

   ```
   SELECT *
   FROM (
       SELECT 'A' AS x
       UNION
       SELECT 'B' AS x
   )
   ORDER BY ALL;
   ```

   You can also sort lists using `list_sort`:

   ```
   SELECT list_sort(array_distinct(['A', 'A', 'B', NULL, NULL])) AS i
   ORDER BY i;
   ```

   It's also possible to introduce a deterministic shuffling.

Limits

This page contains DuckDB's built-in limit values. To check the value of a setting on your system, use the `current_setting` function.

Limit Values

Limit	Default value	Configuration option	Comment
Array size	100000	-	
BLOB size	4 GB	-	
Expression depth	1000	`max_expression_depth`	
Memory allocation for a vector	128 GB	-	
Memory use	80% of RAM	`memory_limit`	Note: This limit only applies to the buffer manager.
String size	4 GB	-	
Temporary directory size	90% of available disk space	`max_temp_directory_size`	

Size of Database Files

DuckDB doesn't have a practical limit for the size of a single DuckDB database file. We have database files using 15 TB+ of disk space and they work fine. However, connecting to such a huge database may take a few seconds and checkpointing can be slower.

DuckDB Docker Container

DuckDB has an official Docker image, which supports both the ARM64 (AArch64) and x86_64 (AMD64) architectures.

Usage

To use the DuckDB Docker image, run:

```
docker run --rm -it -v "$(pwd):/workspace" -w /workspace duckdb/duckdb
```

Using the DuckDB UI with Docker

To use the DuckDB UI with Docker, enable host networking.

> This setting forwards all ports from the container, so exercise caution and avoid it in secure environments.

```
docker run --rm -it -v "$(pwd):/workspace" -w /workspace --net host duckdb/duckdb
```

Then, launch the UI as follows:

```
CALL start_ui();
```

To enable host networking in Docker Desktop, follow the instructions on the Host network driver page.

Development

Overview

The pages in this section are intended for developers who contribute to DuckDB or build extensions for it. If you would like to contribute, please also read the contributing guidelines.

Project Structure and Processes

- DuckDB Repositories lists the repositories that host DuckDB's components, such as clients and extensions.
- Release Cycle describes the release process of DuckDB and its core extensions.

Building DuckDB

- Building DuckDB from Source explains how to compile DuckDB on different platforms and how to build extensions.
- Build Configuration lists the build types and options.
- Troubleshooting covers common build issues.

Testing

- Testing gives an overview of how DuckDB is tested.
- sqllogictest Introduction introduces the sqllogictest format used for most of DuckDB's tests.
- Writing Tests describes how to write new tests.
- Internal Errors explains what internal errors are and how to debug them.

Performance

- Profiling shows how to use DuckDB's built-in query profiler.
- Metrics lists the metrics collected by the profiler.
- Benchmark Suite describes how to run DuckDB's benchmarks to detect performance regressions.

DuckDB Repositories

Several components of DuckDB are maintained in separate repositories.

Main Repositories

- `duckdb`: core DuckDB project
- `duckdb-web`: documentation and blog
- `duckdb-quack`: Quack remote protocol

Clients

- `duckdb-go`: Go client
- `duckdb-java`: Java (JDBC) client
- `duckdb-node-neo`: Node.js client
- `duckdb-odbc`: ODBC client
- `duckdb-pyodide`: Pyodide client
- `duckdb-python`: Python client
- `duckdb-r`: R client
- `duckdb-rs`: Rust client
- `duckdb-swift`: Swift client
- `duckdb-wasm`: WebAssembly client
- `duckplyr`: a drop-in replacement for dplyr in R
- `DuckDB.jl`: Julia client

Connectors

- `dbt-duckdb`: dbt
- `duckdb-mysql`: MySQL connector
- `duckdb-postgres`: PostgreSQL connector (connect to PostgreSQL from DuckDB)
- `duckdb-sqlite`: SQLite connector
- `pg_duckdb`: official PostgreSQL extension for DuckDB (run DuckDB in PostgreSQL)

Extensions

- `duckdb-ui`: web UI for DuckDB
- Core extension repositories are linked in the Official Extensions page
- Community extensions are served from the Community Extensions repository

Specifications

- DuckLake specification

Release Cycle

This document outlines the DuckDB and core extension release cycle framework. It is intended for developers working on DuckDB extensions to better understand the underlying processes.

Overview

- DuckDB follows Semantic Versioning (`v<MAJOR>.<MINOR>.<PATCH>`)
- Feature releases (new minor or major versions) are released approximately every 6 months, e.g., v1.4.0 in September 2025, v1.5.0 in March 2026 and v2.0.0 in October 2026
- Patch releases are issued as needed for all supported versions:
  - The latest stable version
  - Long Term Support (LTS) versions within their support period
  - Versions with extended support (e.g., v1.5, which is supported until March 2027)
- All releases and their end-of-life dates are listed in the Release Calendar

Terminology

In the release docs we use some basic terminology to describe versions and branches. We briefly go over them here.

- **`vx.y.z`**: The latest stable release
- **`vx.y-codename`**: The name of the branch that will produce `vx.y.<n>` releases
- **`vx.<y+1>-codename`**: The branch name that is used for the branch that will produce the next minor release
- **`Main release cycle`**: The branches, commits, and PRs related to producing `vx.<y+1>.0` and `vx.y.<z+1>` releases
- **`Active branch`**: A branch that is part of the main release cycle. Either main or vx.<y+n>-codename where n >= 0
- **`Single branch extension`**: Extension with 1 active branch. Since main is always an active branch this is always main. This means all other branches of format `vx.y-codename` must be `vx.<y-n>-codename` where `n >= 1`
- **`Multi branch extension`**: Extension with more than 1 active branch
- **`Two branch extension`**: Extension with two active branches: main and `vx.y-codename`
- **`Three branch extension`**: Extension with three active branches: main, `vx.y-codename`, and `vx.<y+1>-codename`
- **`LTS release`**: Long term support release. These releases will receive support (patch releases) beyond their lifetime in the active release cycle. Currently LTS releases will receive 1 year of support
- **`Unstable API extension`**: An extension targeting the *unstable* extension API. This can be both the C++ API or the unstable C API. These extensions are not binary-compatible across multiple DuckDB versions
- **`Stable API extension`**: An extension targeting the *stable* C API of DuckDB. These extensions are binary-compatible across multiple DuckDB versions
- **`In-tree extensions`**: Extensions that live inside the `duckdb/duckdb` source tree

Main Branches and Tags

In git-based version control, branches are used to allow multiple versions of the same codebase to co-exist. At DuckDB, there are two core branches that play the main role in the DuckDB (and extensions) release cycle. We will start off by listing the format these core branches come in.

- **`main`** branch: the main branch can mean various things, but can generally be considered the catch-all branch
- **`vx.y-codename`** branch: the branch used to produce all `vx.y.z` releases
- **`vx.y.z`** tag: a stable release of DuckDB. These tags are write-only and will always be tied to the same commit

The Main DuckDB Release Cycle

> LTS (Long-Term Support) releases follow a separate maintenance cycle to provide extended support and stability.

The main DuckDB release cycle consists of 3 main phases: *Mid-cycle*, *Pre-release* and *Feature freeze*. These phases are clearly defined and communicated to ensure the whole team is synchronized and working together towards the next release.

Phase 1: Mid-Cycle

Active DuckDB Branches

- `main`
- `vx.y-codename`

Description The *mid-cycle* phase is the most common phase of the release cycle, with about 75% of the time being spent in this phase. It can be seen as *business-as-usual*, where the upcoming release is still far away and the team is working hard on merging a variety of features and bug-fixes. During this phase, patch releases (`vx.y.<z+n>`) may be created from the `vx.y-codename` branch. The patches are merged into the `vx.y-codename` branch, and the `vx.y-codename` branch is frequently merged into main to keep the two in sync.

PRs into DuckDB

- Bug-fixes for `vx.y.<z+n>` patch releases are merged into `vx.y-codename`
- Features and bug-fixes for `vx.<y+1>.0` are merged into `main`

Phase 2: Pre-Release

Active Branches

- `main`
- `vx.y-codename`
- `vx.<y+1>-codename`

Description The pre-release phase is intended to prepare for the upcoming `vx.<y+1>.0` minor release. At the start of this phase, the `vx.<y+1>-codename` branch is created. This branch will be used to produce the upcoming minor release and is the branch from which all subsequent `vx.<y+1>.<n>` patch releases are released.

PRs into DuckDB

- Bug-fixes for `vx.y.<z+1>` patch releases are merged into `vx.y-codename`
- Features and bug-fixes for `vx.<y+1>.0` are merged into `vx.<y+1>-codename`
- Features for `vx.<y+2>.0` are merged into `main`

Phase 3: Feature Freeze

Active Branches

- `main`
- `vx.y-codename`
- `vx.<y+1>-codename`

Description The feature freeze phase is the phase closest to release. During this phase features are no longer allowed to be merged into `vx.<y+1>-codename` and only bug fixes are merged. This phase is intended to ensure the quality of the upcoming release. During this phase additional testing and benchmarking is performed while reducing the risk of introducing last-minute bugs by disallowing feature merges.

PRs into DuckDB

- Bug-fixes for `vx.y.<z+1>` are no longer allowed, should target `vx.<y+1>.0` instead
- Bug-fixes for `vx.<y+1>.0` are merged into `vx.<y+1>-codename`
- Features for `vx.<y+1>.0` are no longer allowed, should target `vx.<y+2>.0` instead
- Features for `vx.<y+2>.0` are merged into `main`

Main Extension Release Cycle

Most DuckDB extensions are completely separate from the main `duckdb/duckdb` repository and are free to follow their own release cycle. In this section we categorize different types of DuckDB extensions and go over their release cycles.

To describe the release cycle of extensions, we need to first categorize extensions in three different groups, as extensions share the same release cycle based on which of these three categories they belong to.

- In-tree extensions
- Unstable API extensions
- Stable API extensions

We will now go over the release cycles of the three different categories, in order of increasing complexity.

In-Tree Extensions

For *in-tree extensions*, the release cycle is very simple. Since their code lives in the `duckdb/duckdb` repository, they move in complete lock-step with DuckDB. This means they share the same versioning and branching. In this sense they are not really extensions, but more lazy-loadable parts of the `duckdb/duckdb` codebase.

Stable API Extensions

Stable API extensions in DuckDB are a relatively new concept, but are planned to form the majority of extensions in the future. Stable API extensions are built on the stable C extension API, making them binary compatible with multiple versions of DuckDB. This means that their release cycle can/should also be completely separate from the DuckDB release cycle.

While the release cycle for stable API extensions is still work in progress, the basic idea is that the release cycle of stable API extensions consists of a similar but separate cycle to that of `duckdb/duckdb`, where every version will target 1 or more versions of DuckDB.

Unstable API Extensions

Unstable API extensions currently make up the majority of DuckDB extensions. These extensions either target the C++ extension API, or the unstable C extension API. They are, from a release cycle point of view, the most complex. Every version of an unstable API extension only targets a single DuckDB version. This 1:1 tie means that the release cycle of these extensions tends to form a sometimes intricate dance around the main DuckDB release cycle. While the goal is to move as many extensions over to stable APIs, we expect unstable API extensions to be around for quite some time so there remains a need to clearly define their lifecycle. Therefore we will use the remainder of this section to describe it.

Categorizing by Branching To start, we will divide the unstable API extensions into different subcategories. Just like DuckDB itself, these extensions follow the same branching scheme as DuckDB where a combination of `main` and `vx.y-codename` play the main role. We will now define the three types of unstable extensions by looking at their **number of active branches**.

- **Single branch extensions** have only the `main` *active* branch
- **Two branch extensions** have two *active* branches: `main` and `vx.y-codename`
- **Three branch extensions** have three *active* branches: `main`, `vx.y-codename`, and `vx.<y+1>-codename`

DuckDB Targets Every unstable API extension should target a single version of DuckDB. This target version is defined by a combination of **the duckdb submodule** and the target version in the `MainDistributionPipeline` workflow. Which version an extension targets depends on the release cycle phase and the branch. We will now go over all combinations

- Phase: **Mid-cycle**
  - Type: **Single branch**
    - Extension **`main`** -> DuckDB **`vx.y.z`** or **`main`**
  - Type: **Two branch**
    - Extension **`main`** -> DuckDB **`vx.y.z`** or **`main`**
    - Extension **`vx.y-codename`** -> DuckDB **`vx.y.z`** or **`vx.y-codename`**
  - Type: **Three branch**: should not exist
- Phase: **Pre-release** / **Patch**
  - Type: **Single branch**
    - Extension **`main`** -> DuckDB **`vx.y.z`** or **`vx.<y+1>-codename`**
  - Type: **Two branch**
    - Extension **`main`** -> DuckDB **`vx.y.z`** or **`vx.<y+1>-codename`**
    - Extension **`vx.y-codename`** -> DuckDB **`vx.y.z`** or **`vx.y-codename`**
  - Type: **Three branch**
    - Extension **`main`** -> DuckDB **`main`**
    - Extension **`vx.y-codename`** -> DuckDB **`vx.y.z`** or **`vx.y-codename`**
    - Extension **`vx.<y+1>-codename`** -> DuckDB **`vx.<y+1>-codename`**

Where to Merge PRs To know where to merge a PR into an unstable API extension depends on two things: the current release phase and the type of extensions. We will now go over all combinations.

- Phase: **Mid-cycle**
  - Type: **Single branch**
    - if DuckDB target: `vx.y.z`:
      - PR for **`vx.y.<z+1>`** into **`main`**[1]
      - PR for **`vx.<y+1>.0`** merges into **`main`**
    - if DuckDB target: `main`:
      - PR for **`vx.y.<z+1>`** are **impossible**
      - PR for **`vx.<y+1>.0`** merges into **`main`**
  - Type: **Two branch**
    - PR for **`vx.y.<z+1>`** merges into **`vx.y-codename`**
    - PR for **`vx.<y+1>.0`** merges into **`main`**
  - Type: **Three branch**
    - PR for **`vx.y.<z+1>`** merges into **`vx.y-codename`**

[1] Single branch extensions require manual version updates to ensure changes are included in the targeted release.

        * PR for **`vx.<y+1>.0`** merges into **`vx.<y+1>-codename`**
        * PR for **`vx.<y+2>.0`** merges into **`main`**

- Phase: **Pre-release** / **Patch**
  - Type: **Single branch**
    * if DuckDB target: `vx.y.z`:
      · PR for **`vx.y.<z+1>`** into **`main`**[2] [3]
      · PR for **`vx.<y+1>.0`** merges into **`main`**
    * if DuckDB target: `main`:
      · PR for **`vx.y.<z+1>`** are **impossible**
      · PR for **`vx.<y+1>.0`** merges into **`main`**
  - Type: **Two branch**
    * PR for **`vx.y.<z+1>`** merges into **`vx.y-codename`** [4]
    * PR for **`vx.<y+1>.0`** merges into **`main`**
  - Type: **Three branch**
    * PR for **`vx.y.<z+1>`** merges into **`vx.y-codename`**[5]
    * PR for **`vx.<y+1>.0`** merges into **`vx.<y+1>-codename`**
    * PR for **`vx.<y+2>.0`** merges into **`main`**

What Extension Version Will Be Released? Every DuckDB release, a complete set of all core extensions should be available. For unstable API extensions, this means a rebuild of the binaries. For the core extensions, this build generally happens through the `duckdb/duckdb` CI. This means that the list of extensions that will be available on release is documented in the extension config files. However, this config file may not always be up to date. To decide which version of an extension should be part of the upcoming release, we define the following sources-of-truth for latest extension version based on the release type (major/minor) and extension type (single/multi branch):

- Release type: **Patch**
  - Extension type: **Single branch**
    * Latest version: **commit in config files**
  - Extension type: **Multi branch**
    * Latest version: Extension **`vx.y-codename`** branch
- Release type: **Minor**
  - Extension type: **Single branch**
    * Latest version: Extension **`main`** branch
  - Extension type: **Two branch**
    * Latest version: Extension **`main`** branch
  - Extension type: **Three branch**
    * Latest version: Extension **`vx.<y+1>-codename`** branch

Switching between Single Branch, Two Branch and Three Branch Switching between the different branch types for extensions is a fairly straightforward process and should be done as follows:

- Switch: **Single branch** -> **Two branch**
  - When: during **any** phase
  - Reasons:

[2] Single branch extensions require manual version updates to ensure changes are included in the targeted release.
[3] Patch releases during pre-release or feature-freeze phases are uncommon. Consider targeting changes for the next minor release instead.
[4] Patch releases during pre-release or feature-freeze phases are uncommon. Consider targeting changes for the next minor release instead.
[5] Patch releases during pre-release or feature-freeze phases are uncommon. Consider targeting changes for the next minor release instead.

        * When desire arises to merge features not eligible for `vx.y.<z+1>` while also maintaining ability to do releases for `vx.y.<z+n>`
        * To be able to test with latest DuckDB main while maintaining ability to do releases for `vx.y.<z+n>` (including `vx.y.z` itself)
    - Actions:
        * Create branch `vx.y-codename` from a commit on main between HEAD of `main` and the commit in the DuckDB `vx.y.z` config file.

- Switch: **Two branch** -> **Three branch**
    - When: during **Pre-release** or **Feature-freeze** phase
    - Reasons:
        * Whenever a feature needs to be merged that is not eligible for merging into `vx.<y+1>.0`.
    - Actions:
        * Create `vx.<y+1>-codename` branch from main

- Switch **Three branch** -> **Two branch** or **Two branch** -> **Single branch**
    - When: part of transition from **Feature Freeze** -> **Mid-cycle**
    - Action: happens automatically (`vx.y-codename` becomes *inactive* by definition)

Metrics

DuckDB provides a set of metrics that can be used to monitor the performance and health of the database. Metrics are named `<group>.<metric>`, e.g., `query.total_time` or `operator.timing`.

The set of collected metrics is controlled with the `tracked_metrics` setting, which takes a list of glob patterns, e.g., `['query.*', 'operator.timing']`. By default, all metrics are collected (`['*']`). The available metrics, along with their types and units, can be listed with the `duckdb_available_metrics()` table function:

```
SELECT metric_name, metric_type, unit
FROM duckdb_available_metrics();
```

All Metrics

The metrics are organized into the following groups:

- `query`
- `system`
- `io`
- `storage`
- `operator`
- `optimizer`
- `parser`
- `planner`
- `physical_planner`

`query` Metrics

Metrics measured over the entire query.

Name	Type	Unit	Description
`query.cpu_time`	`double`	seconds	CPU time spent on the query
`query.sql`	`string`		The SQL string of the query
`query.total_intermediate_rows`	`uint64`	rows	Cumulative number of intermediate rows produced by the query
`query.total_intermediate_size_bytes`	`uint64`	bytes	Cumulative size in bytes of all intermediate results produced by the query
`query.total_row_groups_scanned`	`uint64`	row_groups	Cumulative number of row groups scanned by the query
`query.total_row_groups_to_scan`	`uint64`	row_groups	Cumulative total number of row groups available to scan
`query.total_rows_scanned`	`uint64`	rows	Cumulative number of rows scanned by the query
`query.total_time`	`double`	seconds	Time spent executing the entire query

`system` Metrics

Metrics on system resources, such as memory and threads, used while running the query.

Name	Type	Unit	Description
`system.blocked_thread_time`	`double`	seconds	Time spent waiting for a thread to become available
`system.peak_buffer_memory`	`uint64`	bytes	Peak memory usage of the buffer manager
`system.peak_streaming_buffer_size`	`uint64`	bytes	Peak number of bytes a streaming query result held: queued bytes plus the units blocked workers hold, both exact. Units still under construction are not counted
`system.peak_temp_dir_size`	`uint64`	bytes	Peak size of the temporary directory
`system.total_memory_allocated`	`uint64`	bytes	The total memory allocated by the buffer manager

`io` Metrics

Metrics on the data read from and written to storage.

Name	Type	Unit	Description
`io.total_bytes_read`	`uint64`	bytes	The total amount of bytes read from storage
`io.total_bytes_written`	`uint64`	bytes	The total amount of bytes written to storage

`storage` Metrics

Metrics on storage operations, such as attaching databases, writing to the WAL and checkpointing.

Name	Type	Unit	Description
`storage.attach_load_storage_latency`	`double`	seconds	Time spent loading from storage
`storage.attach_replay_wal_latency`	`double`	seconds	Time spent replaying the WAL file
`storage.checkpoint_latency`	`double`	seconds	Time spent running checkpoints
`storage.commit_local_storage_latency`	`double`	seconds	Time spent committing the transaction-local storage
`storage.total_vacuum_time`	`double`	seconds	Total time spent running vacuum tasks during checkpointing
`storage.waiting_to_attach_latency`	`double`	seconds	Time spent waiting to ATTACH a file
`storage.wal_replay_entry_count`	`uint64`	rows	The total number of entries to replay in the WAL
`storage.write_to_wal_latency`	`double`	seconds	Time spent writing to the WAL

`operator` Metrics

Metrics collected for each individual operator in the query plan. These are reported in the `operator` tree of the profiling output.

Name	Type	Unit	Description
`operator.cpu_time`	`double`	seconds	CPU time spent in the operator
`operator.extra_info`	`map`		Unique operator metrics
`operator.intermediate_rows`	`uint64`	rows	Number of intermediate rows produced by the operator
`operator.intermediate_size_bytes`	`uint64`	bytes	Intermediate size in bytes produced by the operator
`operator.row_groups_scanned`	`uint64`	row_groups	Number of row groups scanned by the operator
`operator.rows_scanned`	`uint64`	rows	Number of rows scanned by the operator
`operator.timing`	`double`	seconds	Time spent in the operator
`operator.total_row_groups_to_scan`	`uint64`	row_groups	Total number of row groups available for the operator to scan
`operator.type`	`string`		Type of the operator

`optimizer` Metrics

Metrics measuring the time spent in the optimizers.

Name	Type	Unit	Description
`optimizer.total_time`	`double`	seconds	Time spent in all optimizers combined

Additionally, each optimizer has a corresponding metric that follows the template `optimizer.<optimizer_name>` and measures the time (in seconds) spent in that optimizer. These metrics are only collected when the specific optimizer is enabled. The available optimizers can be queried using the `duckdb_optimizers()`{:.language-sql .highlight} table function:

`optimizer.expression_rewriter`, `optimizer.filter_pullup`, `optimizer.filter_pushdown`, `optimizer.empty_result_pullup`, `optimizer.cte_filter_pusher`, `optimizer.regex_range`, `optimizer.in_clause`, `optimizer.join_order`, `optimizer.deliminator`, `optimizer.unnest_rewriter`, `optimizer.unused_columns`, `optimizer.statistics_propagation`, `optimizer.common_subexpressions`, `optimizer.common_aggregate`, `optimizer.column_lifetime`, `optimizer.build_side_probe_side`, `optimizer.limit_pushdown`, `optimizer.top_n`, `optimizer.compressed_materialization`, `optimizer.duplicate_groups`, `optimizer.reorder_filter`, `optimizer.sampling_pushdown`, `optimizer.join_filter_pushdown`, `optimizer.extension`, `optimizer.materialized_cte`, `optimizer.aggregate_function_rewriter`, `optimizer.late_materialization`, `optimizer.cte_inlining`, `optimizer.row_group_pruner`, `optimizer.top_n_window_elimination`, `optimizer.common_subplan`, `optimizer.join_elimination`, `optimizer.window_self_join`, `optimizer.projection_pullup`, `optimizer.outer_join_simplification`, `optimizer.row_number_rewriter`, `optimizer.partitioned_execution`, `optimizer.partial_aggregate_pushdown`.

`parser` Metrics

Metrics on parsing the SQL query.

Name	Type	Unit	Description
parser.total_time	double	seconds	Time spent parsing the SQL query

`planner` Metrics

Metrics on generating the logical plan.

Name	Type	Unit	Description
planner.binding_time	double	seconds	Time spent binding the logical plan
planner.total_time	double	seconds	Time spent generating the logical plan

`physical_planner` Metrics

Metrics on generating the physical plan.

Name	Type	Unit	Description
physical_planner.column_binding	double	seconds	Time spent binding the columns in the logical plan to physical columns
physical_planner.create_plan	double	seconds	Time spent creating the physical plan
physical_planner.resolve_types	double	seconds	Time spent resolving the types in the logical plan to physical types
physical_planner.total_time	double	seconds	Time spent generating the physical plan

Examples

The following example enables profiling, sets the output to a file and only collects the total query time along with the operator metrics.

```
CREATE TABLE students (name VARCHAR, sid INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);

SET enable_profiling = 'json';
SET profiling_output = '/path/to/file.json';
SET tracked_metrics = ['query.total_time', 'operator.*'];

SELECT name
FROM students
JOIN exams USING (sid)
WHERE name LIKE 'Ma%';
```

The file's content after executing the query:

```
{
    "operator": [
        {
            "extra_info": {},
            "intermediate_rows": 0,
```

```
            "intermediate_size_bytes": 0,
            "timing": 5.458e-06,
            "type": "RESULT_COLLECTOR",
            "children": [
                {
                    "extra_info": {
                        "Projections": "name",
                        "Estimated Cardinality": "1"
                    },
                    "intermediate_rows": 2,
                    "intermediate_size_bytes": 40,
                    "timing": 8.3e-08,
                    "type": "PROJECTION",
                    "children": [
                        {
                            "extra_info": {
                                "Join Type": "INNER",
                                "Conditions": "sid = sid",
                                "Estimated Cardinality": "1"
                            },
                            "intermediate_rows": 2,
                            "intermediate_size_bytes": 48,
                            "timing": 4.4209e-05,
                            "type": "HASH_JOIN",
                            "children": [
...
    "query": {
        "total_time": 0.001491167
    }
}
```

The second example collects all `query` and `planner` metrics, as well as the total time spent in the optimizers.

```
SET tracked_metrics = ['query.*', 'planner.*', 'optimizer.total_time'];

SELECT name
FROM students
JOIN exams USING (sid)
WHERE name LIKE 'Ma%';
```

The contents of the output file:

```
{
    "optimizer": {
        "total_time": 0.000179833
    },
    "planner": {
        "binding_time": 5e-05,
        "total_time": 6.8959e-05
    },
    "query": {
        "cpu_time": 5.8541999999999996e-05,
        "sql": "SELECT name FROM students JOIN exams USING (sid) WHERE name LIKE 'Ma%';\n",
        "total_intermediate_rows": 8,
        "total_intermediate_size_bytes": 136,
        "total_row_groups_scanned": 2,
        "total_row_groups_to_scan": 2,
        "total_rows_scanned": 6,
        "total_time": 0.001529
    }
}
```

Profiling

Profiling is essential to help understand why certain queries exhibit specific performance characteristics. DuckDB contains several built-in features to enable query profiling, which this page covers. For a high-level example of using `EXPLAIN`, see the "Inspect Query Plans" page.

Statements

The `EXPLAIN` Statement

The first step to profiling a query can include examining the query plan. The `EXPLAIN` statement shows the query plan and describes what is going on under the hood.

The `EXPLAIN ANALYZE` Statement

The query plan helps developers understand the performance characteristics of the query. However, it is often also necessary to examine the performance numbers of individual operators and the cardinalities that pass through them. The `EXPLAIN ANALYZE` statement enables obtaining these, as it pretty-prints the query plan and also executes the query. Thus, it provides the actual run-time performance numbers.

The `FORMAT` Option

The `EXPLAIN [ANALYZE]` statement allows exporting to several formats:

- `text` – default ASCII-art style output
- `graphviz` – produces a DOT output, which can be rendered with Graphviz
- `html` – produces an HTML output, which can be rendered with treeflex
- `json` – produces a JSON output
- `mermaid` – produces a Mermaid flowchart
- `web` – opens the query graph in a web browser (command line client only)

To specify a format, use the `FORMAT` tag:

```
EXPLAIN (FORMAT html) SELECT 42 AS x;
```

> New As of DuckDB 2.0, the web format is available in the command line client. `EXPLAIN ANALYZE (FORMAT WEB)` renders the query graph and opens it in a web browser, and the `.web` dot command does the same for the previous query. Both are disabled in safe mode.

Pragmas

DuckDB supports several pragmas for turning profiling on and off and controlling the level of detail in the profiling output.

The following pragmas are available and can be set using either `PRAGMA` or `SET`. They can also be reset using `RESET`, followed by the setting name. For more information, see the "Profiling" section of the pragmas page.

Setting	Description	Default	Options
`enable_profiling`, `enable_profile`	Turn on profiling	`query_tree`	`query_tree`, `json`, `query_tree_optimizer`, `no_output`
`profiling_coverage`	Set the operators to profile	`SELECT`	`SELECT`, `ALL`
`profiling_output`	Set a profiling output file	Console	A filepath
`profiling_mode`	Deprecated, has no effect	`standard`	`standard`, `detailed`, `all`
`tracked_metrics`	Select which metrics to collect	`['*']` (all metrics)	A list of metric glob patterns, e.g. `['query.*', 'optimizer.*']`. (List of all available metrics)
`disable_profiling`, `disable_profile`	Turn off profiling		

Table Functions

> These table functions were introduced in DuckDB 1.5.0.

DuckDB provides table functions to enable and disable profiling, consolidating multiple settings into a single call.

`enable_profiling()`

The `enable_profiling()` function configures profiling with the specified options.

```
CALL enable_profiling(
    format := 'json',
    save_location := '/path/to/output.json',
    coverage := 'select',
    metrics := ['query.sql', 'query.total_time', 'operator.timing']
);
```

Parameter	Type	Description
`metrics`	`LIST`, `STRUCT`, or `JSON`	Specifies which metrics to enable
`mode`	`VARCHAR`	Deprecated, has no effect (detailed profiling information is always collected)
`save_location`	`VARCHAR`	File path for profiling output
`coverage`	`VARCHAR`	Query coverage: `'select'` or `'all'`
`format`	`VARCHAR`	Output format: `'query_tree'`, `'json'`, `'query_tree_optimizer'`, `'no_output'`

All parameters are optional and named. You can also pass metrics as an unnamed parameter:

```
CALL enable_profiling(['query.total_time', 'query.total_intermediate_size_bytes']);
```

`disable_profiling()`

The `disable_profiling()` function turns off profiling.

```
CALL disable_profiling();
```

Metrics

DuckDB supports a wide range of metrics that can be enabled or disabled independently. To learn more and to see the full list of available metrics, refer to the metrics documentation.

> New In DuckDB 2.0, the set of collected metrics is controlled with the `tracked_metrics` setting, which takes a list of metric glob patterns such as `['query.*', 'optimizer.*']`. This replaces the `custom_profiling_settings` PRAGMA used in earlier versions. The available metric names, types, and units can be listed with the `duckdb_available_metrics()` table function:
>
> ```
> SELECT metric_name, metric_type, unit
> FROM duckdb_available_metrics();
> ```

Detailed Profiling

> **Deprecated.** In DuckDB 2.0, the `profiling_mode` setting is deprecated and has no effect: detailed profiling information, such as the `planner` and `optimizer` metrics, is always collected. To restrict the collected metrics, use the `tracked_metrics` setting.

Query Graphs

It is also possible to render the profiling output as a query graph. The query graph visually represents the query plan, showing the operators and their relationships. The query plan must be output in the `json` format and stored in a file. After writing a profiling output to its designated file, the Python script can render it as a query graph. The script requires the `duckdb` Python module to be installed. It generates an HTML file and opens it in your web browser.

```
python -m duckdb.query_graph /path/to/file.json
```

Notation in Query Plans

In query plans, the hash join operators adhere to the following convention: the *probe side* of the join is the left operand, while the *build side* is the right operand.

Join operators in the query plan show the join type used:

- Inner joins are denoted as `INNER`.
- Left outer joins and right outer joins are denoted as `LEFT` and `RIGHT`, respectively.
- Full outer joins are denoted as `FULL`.

> **Tip.** To visualize query plans, consider using the DuckDB execution plan visualizer developed by the Database Systems Research Group at the University of Tübingen.

Building DuckDB

Building DuckDB from Source

When Should You Build DuckDB?

DuckDB binaries are available for *stable* and *preview* builds on the installation page. In most cases, it's recommended to use these binaries. When you are running on an experimental platform (e.g., Raspberry Pi) or you would like to build the project for an unmerged pull request, you can build DuckDB from source based on the `duckdb/duckdb` repository hosted on GitHub. This page explains the steps for building DuckDB.

Build Instructions

Prerequisites

> New DuckDB 2.0 requires a C++17 compiler, while earlier versions required a C++11 compiler.

DuckDB needs CMake and a C++17-compliant compiler (e.g., GCC, Clang, Apple-Clang, MSVC). Additionally, we recommend using the Ninja build system, which automatically parallelizes the build process.

Building with the Makefile

A `Makefile` wraps the build process. See Build Configuration for targets and configuration flags.

```
make
make release # same as plain make
make relassert
make debug
GEN=ninja make # for use with Ninja
BUILD_BENCHMARK=1 make # build with benchmarks
```

> `debug` builds use a lot of disk space – make sure you have at least 25 GB available.

Speeding up Subsequent Builds

You can speed up subsequent builds using ccache. On macOS, you can install and configure it as follows:

```
brew install ccache
ccache -M 50G
```

Platforms

Platforms with Full Support

DuckDB fully supports Linux, macOS and Windows. Both x86_64 (amd64) and AArch64 (ARM64) builds are available for these platforms (except for macOS on x86_64, which only has partial support, see below), and almost all extensions are distributed for these platforms.

Platform name	Description	Comment
`linux_amd64`	Linux x86_64 (AMD64) with glibc	
`linux_arm64`	Linux AArch64 (ARM64) with glibc	
`osx_arm64`	macOS 12+ ARM64 (Apple Silicon CPUs)	
`windows_amd64`	Windows 10+ x86_64 (AMD64)	
`windows_arm64`	Windows 10+ AArch64 (ARM64)	

For these platforms, builds are available for both the latest stable version and the preview version (nightly build). In some circumstances, you may still want to build DuckDB from source, e.g., to test an unmerged pull request. For build instructions on these platforms, see:

- Linux
- macOS
- Windows

Platforms with Partial Support

There are several partially supported platforms. For some platforms, DuckDB binaries and extensions (or a subset of extensions) are distributed. For others, building from source is possible.

Platform name	Description
`osx_amd64`	macOS 12+ x86_64 (AMD64) (Intel CPUs). DuckDB binaries and core extensions are distributed, but community extensions are no longer built for this platform. DuckDB 2.1 will stop supporting `osx_amd64`.
`linux_amd64_musl`	Linux x86_64 (AMD64) with musl libc, e.g., Alpine Linux
`linux_arm64_musl`	Linux AArch64 (ARM64) with musl libc, e.g., Alpine Linux
`linux_arm64_android`	Android AArch64 (ARM64)
`wasm_eh`	WebAssembly Exception Handling

Below, we provide detailed build instructions for some platforms:

- Android
- Raspberry Pi

Platforms with Best Effort Support

Platform name	Description
`freebsd_amd64`	FreeBSD x86_64 (AMD64)
`freebsd_arm64`	FreeBSD AArch64 (ARM64)
`openharmony_arm64`	OpenHarmony AArch64 (ARM64)
`wasm_mvp`	WebAssembly Minimum Viable Product
`windows_amd64_mingw`	Windows 10+ x86_64 (AMD64) with MinGW

Platform name	Description
`windows_arm64_mingw`	Windows 10+ AArch64 (ARM64) with MinGW

> These platforms are not covered by DuckDB's community support. For details on commercial support, see the support policy page.

See the OpenHarmony page for build instructions.

See also the "Unofficial and Unsupported Platforms" page for details.

Outdated Platforms

Some platforms were supported in older DuckDB versions but are no longer supported.

Platform name	Description
`linux_amd64_gcc4`	Linux x86_64 (AMD64) with GCC 4, e.g., CentOS 7
`linux_arm64_gcc4`	Linux AArch64 (ARM64) with GCC 4, e.g., CentOS 7
`windows_amd64_rtools`	Windows 10+ x86_64 (AMD64) for RTools

DuckDB can also be built for end-of-life platforms such as macOS 11 and CentOS 7/8 using the instructions provided for macOS and Linux.

Troubleshooting Guides

We provide troubleshooting guides for building DuckDB:

- Generic issues
- Python
- R

Building Configuration

Build Types

DuckDB can be built in many different settings, most of these correspond directly to CMake but not all of them.

`release`

This build has been stripped of all the assertions and debug symbols and code, optimized for performance.

`debug`

This build runs with all the debug information, including symbols, assertions and `#ifdef DEBUG` blocks. Due to these, binaries of this build are expected to be slow. Note: the special debug defines are not automatically set for this build.

`relassert`

This build does not trigger the `#ifdef DEBUG` code blocks but it still has debug symbols that make it possible to step through the execution with line number information and `D_ASSERT` lines are still checked in this build. Binaries of this build mode are significantly faster than those of the `debug` mode.

`reldebug`

This build is similar to `relassert` in many ways, only assertions are also stripped in this build.

`benchmark`

This build is a shorthand for `release` with `BUILD_BENCHMARK=1` set.

`tidy-check`

This creates a build and then runs Clang-Tidy to check for issues or style violations through static analysis. The CI will also run this check, causing it to fail if this check fails.

`format-fix` | `format-changes` | `format-main`

This doesn't actually create a build, but uses the following format checkers to check for style issues:

- clang-format to fix format issues in the code.
- cmake-format to fix format issues in the `CMakeLists.txt` files.

The CI will also run this check, causing it to fail if this check fails.

Extension Selection

Core DuckDB extensions are the ones maintained by the DuckDB team. These are hosted in the `duckdb` GitHub organization and are served by the `core` extension repository.

Additional extensions can be built as part of DuckDB via the `BUILD_EXTENSIONS` flag, then listing the names of the extensions that are to be built.

```
BUILD_EXTENSIONS='tpch;httpfs;fts;json;parquet' make
```

More on this topic at building DuckDB extensions.

Package Flags

The Makefile has flags to control which components are built. These can be enabled by either setting them in the current `env`, through set up files like `bashrc` or `zshrc`, or by setting them before the call to `make`, for example:

```
BUILD_BENCHMARK=1 make debug
```

> The Python, Java (JDBC) and ODBC clients are maintained in separate repositories and are no longer built from the main DuckDB repository. See the Python, Java and ODBC repositories for their build instructions.

`BUILD_BENCHMARK`

When this flag is set, DuckDB's in-house benchmark suite is built. More information about this can be found in the README.

`DISABLE_SHELL`

The CLI is built by default. When this flag is set, the CLI is not built.

Miscellaneous Flags

`DISABLE_UNITY`

To improve compilation time, we use Unity Build to combine translation units. This can however hide include bugs, this flag disables using the unity build so these errors can be detected.

`DISABLE_SANITIZER`

In some situations, running an executable that has been built with sanitizers enabled is not supported / can cause problems. Julia is an example of this. With this flag enabled, the sanitizers are disabled for the build.

Overriding Git Hash and Version

It is possible to override the Git hash and version when building from source using the `OVERRIDE_GIT_DESCRIBE` environment variable. This is useful when building from sources that are not part of a complete Git repository (e.g., an archive file with no information on commit hashes and tags). For example:

```
OVERRIDE_GIT_DESCRIBE=v0.10.0-843-g09ea97d0a9 GEN=ninja make
```

Will result in the following output when running `./build/release/duckdb`:

```
v0.10.1-dev843 09ea97d0a9
...
```

Building without Git

Git and Python are optional dependencies for building DuckDB from source. DuckDB normally derives its version and commit hash from Git, so `OVERRIDE_GIT_DESCRIBE` is primarily useful when building from sources that are not a complete Git checkout. The version and commit hash are resolved as follows:

- If `OVERRIDE_GIT_DESCRIBE` is set to a full `git describe --tags --long` string (for example, `v1.0.0-843-g09ea97d0a9`), the version, the development iteration, and the commit hash are all extracted from it.
- If `OVERRIDE_GIT_DESCRIBE` is set to a version only (for example, `v1.0.0`), the remaining details are filled in from the current Git commit when Git is available.
- If `OVERRIDE_GIT_DESCRIBE` is not set and Git is available, DuckDB runs `git describe --tags --long` to determine the version and commit hash.
- If `OVERRIDE_GIT_DESCRIBE` is not set and Git is not available, DuckDB falls back to a placeholder version and commit hash, and logs a message during the CMake configuration step. The build still succeeds.

Building Extensions

Extensions can be built from source and installed from the resulting local binary.

Building Extensions

To build using extension flags, set the `BUILD_EXTENSIONS` flag to the list of extensions that you want to be built. For example:

```
BUILD_EXTENSIONS='autocomplete;httpfs;icu;json;tpch' GEN=ninja make
```

This option also accepts out-of-tree extensions such as `delta`:

```
BUILD_EXTENSIONS='autocomplete;httpfs;icu;json;tpch;delta' GEN=ninja make
```

Since DuckDB 2.0, extensions listed in `BUILD_EXTENSIONS` are built but not statically linked into the DuckDB executable by default: they are installed from the build's own extension repository and loaded on first use. To link them into the executable, set `LINK_CORE_EXTENSIONS=1` or list them in `STATICALLY_LINK_EXTENSIONS` (see Linking Extensions).

Special Extension Flags

`BUILD_JEMALLOC`

This flag controls whether the `jemalloc` memory allocator is linked into DuckDB. Unlike TPC-H and TPC-DS, `jemalloc` is not an extension. It is only supported on 64-bit Linux, where it is enabled by default. To disable it, set `BUILD_JEMALLOC=0`.

`BUILD_TPCE`

When this flag is set, the TPCE library is built. Unlike TPC-H and TPC-DS this is not a proper extension and it's not distributed as such. Enabling this allows TPC-E enabled queries through our test suite.

Debug Flags

`CRASH_ON_ASSERT`

`D_ASSERT(condition)` is used all throughout the code, these will throw an InternalException in debug builds. With this flag enabled, when the assertion triggers it will instead directly cause a crash.

`DISABLE_STRING_INLINE`

In our execution format `string_t` has the feature to "inline" strings that are under a certain length (12 bytes), this means they don't require a separate allocation. When this flag is set, we disable this and don't inline small strings.

`DISABLE_MEMORY_SAFETY`

Our data structures that are used extensively throughout the non-performance-critical code have extra checks to ensure memory safety, these checks include:

- Making sure `nullptr` is never dereferenced.
- Making sure index out of bounds accesses don't trigger a crash.

With this flag enabled we remove these checks, this is mostly done to check that the performance hit of these checks is negligible.

`DESTROY_UNPINNED_BLOCKS`

When previously pinned blocks in the BufferManager are unpinned, with this flag enabled we destroy them instantly to make sure that there aren't situations where this memory is still being used, despite not being pinned.

DEBUG_STACKTRACE

When a crash or assertion hit occurs in a test, print a stack trace. This is useful when debugging a crash that is hard to pinpoint with a debugger attached.

Using a CMake Configuration File

To build using a CMake configuration file, create an extension configuration file named `extension_config.cmake` with e.g., the following content:

```
duckdb_extension_load(autocomplete)
duckdb_extension_load(tpch)
duckdb_extension_load(tpcds)
duckdb_extension_load(icu)
duckdb_extension_load(json)
duckdb_extension_load(parquet)
```

Build DuckDB as follows:

```
GEN=ninja EXTENSION_CONFIGS="extension_config.cmake" make
```

Then, to install the extensions in one go, run:

```
# for release builds
cd build/release/extension/
# for debug builds
cd build/debug/extension/
# install extensions
for EXTENSION in *; do
    ../duckdb -c "INSTALL '${EXTENSION}/${EXTENSION}.duckdb_extension';"
done
```

Statically Linking Extensions

This page is for client and binding authors, and for anyone who embeds `libduckdb` in their own application. It describes how statically linking extensions changed in DuckDB 2.0 and what builders on top of DuckDB need to do.

> This change is independent of the new C API. It applies even if you continue to build against the existing (v1) C API: the difference is in how extension archives are linked and registered, not in which API your bindings call.

What Changed in DuckDB 2.0

> New Link-time static linking of extensions was introduced in DuckDB 2.0.

Before v2.0, the set of statically linked extensions was a *compile-time* decision. The sources were compiled with knowledge of which extensions were included, so a build with `CORE_EXTENSIONS='json;parquet;icu'` could not reuse the object files of a build with `CORE_EXTENSIONS='json;parquet;icu;tpch'`, even though one is a subset of the other.

DuckDB 2.0 modularized this into a *link-time* decision (duckdb#25814). Each extension is compiled once into its own static archive, for example:

```
libduckdb_static.a
libcore_functions_extension.a
libparquet_extension.a
libjson_extension.a
libicu_extension.a
```

The list of extensions to link in is now chosen when linking, not when compiling. As a follow-up, the extension loader and the built-in `httplib` also became opt-in at link time (duckdb#26188).

Two aspects that earlier builds relied on are gone:

- The `bundle-library` Makefile target that merged everything into a single `libduckdb_bundle.a`. Extensions, `httplib`, and loadable extensions are now separate archives.
- The generated loader that `libduckdb_static.a` used to register extensions automatically. `libduckdb_static.a` no longer registers extensions on its own.

Registering Statically Linked Extensions

Because extension selection moved to link time, a client that statically links extensions must now *register* them with the engine before any database is opened. Each extension archive carries a generated describe function, `duckdb_extension_<name>_describe`, that fills a `duckdb_extension_descriptor` declared in the new public header `duckdb_static_extension.h`. The descriptor carries a layout version, so the engine refuses an archive whose descriptor layout it does not support, with a clear message instead of misbehaving.

The engine exposes a C API call, `duckdb_register_static_extension(describe)`, which validates the descriptor and registers the extension for every database opened afterwards. A generated loader (`static_extension_loader.c`) defines `duckdb_register_static_extensions()`, which passes every linked extension's describe function to `duckdb_register_static_extension` in turn. Call it once at startup, before any database is opened, from any language that can call C:

```
// Register the statically linked extensions before any database can be opened
if (duckdb_register_static_extensions() != 0) {
    throw std::runtime_error("Failed to register the statically linked DuckDB extensions");
}
```

If registration is skipped, the linked extensions (such as `core_functions`, `icu`, `json`, and `parquet`) are not loaded. Queries that depend on them then fail at runtime. For example, `SELECT to_minutes(5)` reports that the function exists in the `core_functions` extension, even though that extension was linked in.

A program can also skip the generated loader and call `duckdb_register_static_extension` directly on the describe functions it declares itself. An archive whose describe function is never registered contributes nothing, and a registered extension whose archive is missing fails the link.

Calling the Loader Before `main`

A C++ program that wants registration to happen automatically before `main` compiles `extension/loader/static_extension_autoregister.cpp` alongside the loader. Compile it into the program itself, not into an archive, or the linker drops it.

If you do package the loader and the auto-register object into an archive, keep the registration symbol from being discarded by forcing it onto the link line:

```
-Wl,-u,duckdb_register_static_extensions
```

Build Options

When building DuckDB or `libduckdb`, select the extensions to compile with `BUILD_EXTENSIONS` (or `CORE_EXTENSIONS`):

```
BUILD_EXTENSIONS='core_functions;json;parquet;icu' GEN=ninja make
```

Which of the built extensions are statically linked is a separate decision. By default the DuckDB targets link the extensions that the build configs mark for static linking, plus the ones named in `BUILD_EXTENSIONS` when you build with `LINK_CORE_EXTENSIONS=1`. To link an explicit set instead, set `STATICALLY_LINK_EXTENSIONS` (the older name for this flag is `LINK_EXTENSIONS`):

```
STATICALLY_LINK_EXTENSIONS='core_functions;parquet;json;icu;tpch' make
```

Changing the linked set only re-generates the loader, compiles it, and re-links; it does not recompile the extensions themselves. An extension that is built but not statically linked is still available: it is installed from the build's own extension repository and loaded the first time a query needs it.

Linking the Archives by Hand

Outside CMake, link one archive per extension, compile the generated loader, and put the engine archive last:

```
LINK_EXTENSIONS='parquet;json' make static_extension_loader
cc -I src/include main.c build/release/static_extension_loader.c libparquet_extension.a libjson_extension.a
libduckdb_static.a -lstdc++
```

The order matters: each extension archive must come before `libduckdb_static.a` on the link line.

See Build Configuration and Building Extensions for the full set of flags.

Obtaining Prebuilt Archives

Rather than building `libduckdb` yourself, you can download the static and shared archives for DuckDB 2.0 and later using the `duckdb-install-scripts` tooling (`duckdb-install-scripts#5`). Setting `DUCKDB_INSTALL=static` installs the archives and headers under `~/.duckdb/lib/<version>/<build_type>_<platform>` (e.g., `~/.duckdb/lib/2.0.0/release_osx_arm64`):

```
curl https://install.duckdb.org | DUCKDB_INSTALL=static bash
```

Use `DUCKDB_INSTALL=shared` for the shared library, or combine components, e.g., `DUCKDB_INSTALL=static,shared,cli`. See the install script page for details.

This installs, for example:

```
duckdb.h                        duckdb_v2.h
duckdb_extension.h              duckdb_extension_v2.h
duckdb_static_extension.h
libduckdb_static.a              libcore_functions_extension.a
libparquet_extension.a          libjson_extension.a
libicu_extension.a              libautocomplete_extension.a
libtpch_extension.a             libtpcds_extension.a
```

A specific release can be installed by setting `DUCKDB_VERSION` (e.g., `DUCKDB_VERSION=2.0.0`). Static and shared libraries are only available for DuckDB 2.0 and later.

> For DuckDB 2.0 and later, most artifacts are distributed as `.tar.gz` files instead of `.zip` files (`duckdb#24773`).

Reference Implementations for Clients

The following pull requests adapt the official clients to the v2.0 linking model. They provide a path forward for anyone maintaining a binding or an application that links `libduckdb`.

Client	Status	Reference pull requests
Python	Migrated	`duckdb-python#634`
Rust	Migrated	`duckdb-rs#834`, `duckdb-rs#839`, `duckdb-rs#883`, `duckdb-rs#835`
Node.js	In progress	`duckdb-node-neo#476`, `duckdb-node-neo#508`, `duckdb-node-neo#512`
Go	In progress	`duckdb-go-bindings#112`, `duckdb-go-bindings#116`
Wasm	In progress	`duckdb-wasm#2281`

> The Python pull request is the clearest end-to-end example: it restores static linking of the default `BUILD_EXTENSIONS` set by calling `duckdb_register_static_extensions()` and shows how to override the linked set.

See Also

- Building Extensions
- Build Configuration
- C API Overview
- Extensions Overview

Android

DuckDB has experimental support for Android. Please use the latest `main` branch of DuckDB instead of the stable versions.

Building the DuckDB Library Using the Android NDK

We provide build instructions for setups using macOS and Android Studio. For other setups, please adjust the steps accordingly.

1. Open Android Studio. Select the **Tools** menu and pick **SDK Manager**. Select the SDK Tools tab and tick the **NDK (Side by side)** option. Click **OK** to install.
2. Set the Android NDK's location. For example:

   ```
   ANDROID_NDK=~/Library/Android/sdk/ndk/28.0.12433566/
   ```

3. Set the Android ABI. For example:

   ```
   ANDROID_ABI=arm64-v8a
   ```

   Or:

   ```
   ANDROID_ABI=x86_64
   ```

4. If you would like to use the Ninja build system, make sure it is installed and available on the PATH.
5. Set the list of DuckDB extensions to build. These will be statically linked in the binary. For example:

   ```
   DUCKDB_EXTENSIONS="icu;json;parquet"
   ```

6. Navigate to DuckDB's directory and run the build as follows:

   ```
   PLATFORM_NAME="android_${ANDROID_ABI}"
   BUILDDIR=./build/${PLATFORM_NAME}
   mkdir -p ${BUILDDIR}
   cd ${BUILDDIR}
   cmake \
       -G "Ninja" \
       -DEXTENSION_STATIC_BUILD=1 \
       -DDUCKDB_EXTRA_LINK_FLAGS="-llog" \
       -DBUILD_EXTENSIONS=${DUCKDB_EXTENSIONS} \
       -DENABLE_EXTENSION_AUTOLOADING=1 \
       -DENABLE_EXTENSION_AUTOINSTALL=1 \
       -DCMAKE_VERBOSE_MAKEFILE=on \
       -DANDROID_PLATFORM=${ANDROID_PLATFORM} \
       -DLOCAL_EXTENSION_REPO="" \
       -DOVERRIDE_GIT_DESCRIBE="" \
       -DDUCKDB_EXPLICIT_PLATFORM=${PLATFORM_NAME} \
       -DBUILD_UNITTESTS=0 \
       -DBUILD_SHELL=1 \
       -DANDROID_ABI=${ANDROID_ABI} \
   ```

```
    -DCMAKE_TOOLCHAIN_FILE=${ANDROID_NDK}/build/cmake/android.toolchain.cmake \
    -DCMAKE_BUILD_TYPE=Release ../..
cmake \
    --build . \
    --config Release
```

7. For the `arm64-v8a` ABI, the build will produce the `build/android_arm64-v8a/duckdb` and `build/android_arm64-v8a/src/libduckdb.so` binaries.

Building the CLI in Termux

1. To build the command line client in the Termux application, install the following packages:

   ```
   pkg install -y git ninja clang cmake python3
   ```

2. Set the list of DuckDB extensions to build. These will be statically linked in the binary. For example:

   ```
   DUCKDB_EXTENSIONS="icu;json"
   ```

3. Build DuckDB as follows:

   ```
   mkdir build
   cd build
   export LDFLAGS="-llog"
   cmake \
      -G "Ninja" \
      -DBUILD_EXTENSIONS="${DUCKDB_EXTENSIONS}" \
      -DDUCKDB_EXPLICIT_PLATFORM=linux_arm64_android \
      -DCMAKE_BUILD_TYPE=Release \
      ..
   cmake --build . --config Release
   ```

Note that you can also use the Python client on Termux:

```
pip install --pre --upgrade duckdb
```

Troubleshooting

Log Library Is Missing

Problem: The build throws the following error:

```
ld.lld: error: undefined symbol: __android_log_write
```

Solution: Make sure the log library is linked:

```
export LDFLAGS="-llog"
```

OpenHarmony

DuckDB supports building for OpenHarmony (OHOS) AArch64 (ARM64) on a best-effort basis. On this platform, DuckDB reports the platform name `openharmony_arm64`. Prebuilt binaries and extensions are not distributed for this platform.

Prerequisites

DuckDB needs CMake, Ninja, and a C++17 compiler on the build host, plus the OpenHarmony SDK. The Linux x64 SDK from the OpenHarmony release page can be used for cross-compilation. The repository ships a toolchain file that handles both the Linux x64 SDK layout and the on-device SDK layout.

Set `OHOS_SDK` to the `native` directory of the SDK:

```
export OHOS_SDK=/path/to/ohos-sdk/linux/native
```

Cross-Compiling

Configure with the OpenHarmony toolchain file:

```
cmake -G Ninja -B build/openharmony \
    -DCMAKE_TOOLCHAIN_FILE="$PWD/scripts/ohos-toolchain.cmake" \
    -DCMAKE_BUILD_TYPE=Release \
    -DDUCKDB_EXPLICIT_PLATFORM=openharmony_arm64

cmake --build build/openharmony --parallel
```

`DUCKDB_EXPLICIT_PLATFORM` is required when cross-compiling because the platform probe binary cannot run on the build host. The resulting `build/openharmony/duckdb` is an AArch64 musl binary.

Building on an OpenHarmony Device

On a device, CMake detects the OpenHarmony toolchain and the platform probe runs natively, so `DUCKDB_EXPLICIT_PLATFORM` is not required:

```
cmake -G Ninja -B build/openharmony \
    -DCMAKE_TOOLCHAIN_FILE="$PWD/scripts/ohos-toolchain.cmake" \
    -DCMAKE_BUILD_TYPE=Release

cmake --build build/openharmony --parallel
./build/openharmony/duckdb -c "PRAGMA platform;"
```

Notes

- The shell must not be stripped on OpenHarmony: the OS verifies the ELF `.codesign` section on exec.
- No prebuilt extensions are distributed for `openharmony_arm64`; build the extensions you need from source. `INSTALL <extension>` reports that no extension is available for this platform.
- Loadable extensions built with the CMake build are re-signed after DuckDB appends its metadata footer, so they can be `dlopen`ed on OpenHarmony.
- The unittest binary is linked against the static DuckDB library on OpenHarmony because the shared library does not export the internal symbols used by the test helpers.

Linux

Prerequisites

On Linux, install the required packages with the package manager of your distribution.

Ubuntu and Debian

CLI Client On Ubuntu and Debian (and also MX Linux, Linux Mint, etc.), the requirements for building the DuckDB CLI client are the following:

```
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build libssl-dev libcurl4-openssl-dev
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Fedora, CentOS and Red Hat

CLI Client The requirements for building the DuckDB CLI client on Fedora, CentOS, Red Hat, AlmaLinux, Rocky Linux, etc. are the following:

```
sudo yum install -y git g++ cmake ninja-build openssl-devel
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Note that on older Red Hat-based distributions, you may have to change the package name for `g++` to `gcc-c++`, skip Ninja and manually configure the number of Make jobs:

```
sudo yum install -y git gcc-c++ cmake openssl-devel
git clone https://github.com/duckdb/duckdb
cd duckdb
mkdir build
cd build
cmake ..
make -j`nproc`
```

Arch Linux

The following instructions are intended for Arch Linux and Arch-based distributions (e.g., Manjaro, Omarchy).

CLI Client DuckDB is available in Arch's Extra package repository. To install it, run:

```
sudo pacman -S duckdb
```

The requirements for building the DuckDB CLI client on Arch, Manjaro, etc. are the following:

```
sudo pacman -S git gcc cmake ninja openssl
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Alpine Linux

CLI Client The requirements for building the DuckDB CLI client on Alpine Linux are the following:

```
apk add g++ git make cmake ninja
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Performance with musl libc Note that Alpine Linux uses musl libc as its C standard library. DuckDB binaries built with musl libc have lower performance compared to the glibc variants: for some workloads, the slowdown can be more than 5×. Therefore, it's recommended to use glibc for performance-oriented workloads.

Distribution for the `linux_*_musl` Platforms Starting with DuckDB 1.2.0, *DuckDB extensions* are distributed for the `linux_amd64_musl` platform, and starting with DuckDB 1.5.0 also for the `linux_arm64_musl` platform. Since DuckDB 1.5.0, official CLI binaries and libraries are also distributed for both musl platforms (e.g., `duckdb-cli-linux-amd64-musl.tar.gz` on the GitHub release page). You can also build DuckDB with musl libc manually following the instructions on this page.

Python Client on Alpine Linux Currently, installing the DuckDB Python on Alpine Linux requires compilation from source. To do so, install the required packages before running `pip`:

```
apk add g++ py3-pip python3-dev
pip install duckdb
```

Using the DuckDB CLI Client on Linux

Once the build finishes successfully, you can find the `duckdb` binary in the `build` directory:

```
build/release/duckdb
```

For different build configurations (`debug`, `relassert`, etc.), please consult the "Build Configurations" page.

Building Extensions

To build extensions, set the `BUILD_EXTENSIONS` flag to the list of extensions that you want to be built. For example:

```
BUILD_EXTENSIONS='autocomplete;httpfs;icu;json;tpch' GEN=ninja make
```

Troubleshooting

R Package on Linux AArch64: `too many GOT entries` Build Error

Problem: Building the R package on Linux running on an ARM64 architecture (AArch64) may result in the following error message:

```
/usr/bin/ld: /usr/include/c++/10/bits/basic_string.tcc:206:
warning: too many GOT entries for -fpic, please recompile with -fPIC
```

Solution: Create or edit the `~/.R/Makevars` file. This example also contains the `MAKEFLAGS` setting to parallelize the build:

```
ALL_CXXFLAGS = $(PKG_CXXFLAGS) -fPIC $(SHLIB_CXXFLAGS) $(CXXFLAGS)
MAKEFLAGS = -j$(nproc)
```

Building the httpfs Extension Fails

Problem: When building the `httpfs` extension on Linux, the build may fail with the following error.

```
CMake Error at /usr/share/cmake-3.22/Modules/FindPackageHandleStandardArgs.cmake:230 (message):
  Could NOT find OpenSSL, try to set the path to OpenSSL root folder in the
  system variable OPENSSL_ROOT_DIR (missing: OPENSSL_CRYPTO_LIBRARY
  OPENSSL_INCLUDE_DIR)
```

Solution: Install the `libssl-dev` library.

```
sudo apt-get install -y libssl-dev
```

Then, build with:

```
GEN=ninja BUILD_EXTENSIONS="httpfs" make
```

macOS

Prerequisites

Install Xcode and Homebrew. Then, install the required packages with:

```
brew install git cmake ninja
```

Building DuckDB

Clone and build DuckDB as follows.

```
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Once the build finishes successfully, you can find the `duckdb` binary in the `build` directory:

```
build/release/duckdb
```

For different build configurations (`debug`, `relassert`, etc.), please consult the Build Configurations page.

Troubleshooting

Build Failure: `'string' file not found`

Problem: The build fails on macOS with the following error:

```
FAILED: third_party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o
/Library/Developer/CommandLineTools/usr/bin/c++ -DDUCKDB_BUILD_LIBRARY -DEXT_VERSION_
PARQUET=\"9cba6a2a03\" -I/Users/builder/external/duckdb/src/include
-I/Users/builder/external/duckdb/third_party/fsst -I/Users/builder/external/duckdb/third_party/fmt/include
-I/Users/builder/external/duckdb/third_party/hyperloglog -I/Users/builder/external/duckdb/third_
party/fastpforlib -I/Users/builder/external/duckdb/third_party/skiplist
-I/Users/builder/external/duckdb/third_party/fast_float -I/Users/builder/external/duckdb/third_party/re2
-I/Users/builder/external/duckdb/third_party/miniz -I/Users/builder/external/duckdb/third_
party/utf8proc/include -I/Users/builder/external/duckdb/third_party/concurrentqueue
-I/Users/builder/external/duckdb/third_party/pcg -I/Users/builder/external/duckdb/third_party/tdigest
-I/Users/builder/external/duckdb/third_party/mbedtls/include -I/Users/builder/external/duckdb/third_
party/jaro_winkler -I/Users/builder/external/duckdb/third_party/yyjson/include
-I/Users/builder/external/duckdb/third_party/libpg_query/include -O3 -DNDEBUG -O3 -DNDEBUG   -std=c++11
-arch arm64 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX15.1.sdk -fPIC -fvisibility=hidden
-fcolor-diagnostics -w -MD -MT third_party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_
list.cpp.o -MF third_party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o.d -o
third_party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o -c
/Users/builder/external/duckdb/third_party/libpg_query/src_backend_nodes_list.cpp
In file included from /Users/builder/external/duckdb/third_party/libpg_query/src_backend_nodes_list.cpp:35:
/Users/builder/external/duckdb/third_party/libpg_query/include/pg_functions.hpp:4:10: fatal error:
'string' file not found
    4 | #include <string>
```

Solution: Users report that reinstalling Xcode fixed their problem. See related discussions on the DuckDB GitHub issues and on Stack Overflow.

> **Warning.** Attempting to reinstall your Xcode suite may impact other applications on your system. Proceed with caution.

```
sudo rm -rf /Library/Developer/CommandLineTools
xcode-select --install
```

Debug Build Prints malloc Warning

Problem: The debug build on macOS prints a `malloc` warning, e.g.:

```
duckdb(83082,0x205b30240) malloc: nano zone abandoned due to inability to reserve vm space.
```

Solution: To prevent this, set the `MallocNanoZone` flag to 0:

```
MallocNanoZone=0 make debug
```

To apply this change for your future terminal sessions, you can add the following to your `~/.zshrc` file:

```
export MallocNanoZone=0
```

Raspberry Pi

DuckDB is not officially distributed for the Raspberry Pi OS (previously called Raspbian). You can build it following the instructions on this page.

Raspberry Pi (64-bit)

First, install the required build packages:

```
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build
```

Then, clone and build it as follows:

```
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja BUILD_EXTENSIONS="icu;json" make
```

Finally, run it:

```
build/release/duckdb
```

Raspberry Pi (32-bit)

On 32-bit Raspberry Pi boards, you need to add the `-latomic` link flag. As extensions are not distributed for this platform, it's recommended to also include them in the build. For example:

```
mkdir build
cd build
cmake .. \
    -DBUILD_EXTENSIONS="httpfs;json;parquet" \
    -DDUCKDB_EXTRA_LINK_FLAGS="-latomic"
make -j4
```

Windows

On Windows, DuckDB requires the Microsoft Visual C++ Redistributable package both as a build-time and runtime dependency. Note that unlike the build process on UNIX-like systems, the Windows builds directly call CMake.

Clang-cl

To build DuckDB on Windows, we recommend using the `clang-cl` compiler with the Visual Studio build tools (MSVC toolchain, Windows SDK and linker). This is the setup used by DuckDB's CI workflow.

1. Install Visual Studio (or the Build Tools for Visual Studio) with the "Desktop development with C++" workload and the "C++ Clang Compiler for Windows" component. You also need Python, CMake, Ninja and a Bash shell with GNU Make (e.g., Git Bash with `make` installed).
2. Open a shell where the Visual Studio environment variables are set (e.g., the "x64 Native Tools Command Prompt" or a shell started from it).
3. Run the following commands in a Bash shell from the root of the DuckDB repository:

   ```
   export GEN=ninja
   export CC=clang-cl
   export CXX=clang-cl
   python scripts/windows_ci.py
   make windows_release
   ```

The `make windows_release` target invokes CMake with the bundled extensions configuration (`.github/config/bundled_extensions.cmake`) and builds DuckDB in `Release` mode.

MSYS2 and MinGW64

DuckDB on Windows can also be built with MSYS2 and MinGW64. Note that this build is only supported for compatibility reasons and should only be used if the `clang-cl` build is not feasible on a given platform. To build DuckDB with MinGW64, install the required dependencies using Pacman. When prompted with `Enter a selection (default=all)`, select the default option by pressing `Enter`.

```
pacman -Syu git mingw-w64-x86_64-toolchain mingw-w64-x86_64-cmake mingw-w64-x86_64-ninja
git clone https://github.com/duckdb/duckdb
cd duckdb
cmake -G "Ninja" -DCMAKE_BUILD_TYPE=Release -DBUILD_EXTENSIONS="icu;parquet;json"
cmake --build . --config Release
```

Once the build finishes successfully, you can find the `duckdb.exe` binary in the repository's directory:

```
./duckdb.exe
```

Building the Go Client

Building on Windows may result in the following error:

```
go build
```

```
collect2.exe: error: ld returned 5 exit status
```

GitHub user vdmitriyev shared instructions for building the DuckDB Go client on Windows:

1. Get four files (`.dll`, `.lib`, `.hpp`, `.h`) from the `duckdb-shared-libs-windows-amd64.tar.gz` archive (called `libduckdb-windows-amd64.zip` before DuckDB 2.0).
2. Place them to, e.g.,: `C:\duckdb-go\libs\`.
3. Install the dependencies following the `duckdb-go` project.
4. Build your project using the following instructions:

   ```
   set PATH=C:\duckdb-go\libs\;%PATH%
   set CGO_CFLAGS=-IC:\duckdb-go\libs\
   set CGO_LDFLAGS=-LC:\duckdb-go\libs\ -lduckdb
   go build
   ```

Python

The DuckDB Python package has its own repository at `duckdb/duckdb-python` and uses nanobind 3 to create Python bindings with DuckDB.

Prerequisites

This guide assumes:

1. You have a full clone of the DuckDB Python package source, including its git submodule
2. You have Astral UV version >= 0.8.0 installed
3. You run commands from the root of the `duckdb-python` source

We are opinionated about using **Astral UV** for Python environment and dependency management. While using pip for a development environment with an editable install without build isolation is possible, we don't provide guidance for that approach in this guide.

We use **CLion** as our IDE. This guide doesn't include specific instructions for other IDEs, but the setup should be similar.

1. DuckDB Python Repository

Start by forking `duckdb-python` into a personal repository, then clone your fork:

```
git clone --recurse-submodules YOUR_FORK_URL
cd duckdb-python
git remote add upstream https://github.com/duckdb/duckdb-python.git
git fetch --all
```

If you've already cloned without submodules:

```
git submodule update --init --recursive
git remote add upstream https://github.com/duckdb/duckdb-python.git
git fetch --all
```

Important notes:

- DuckDB is vendored as a git submodule and must be initialized, otherwise the build fails with `RuntimeError: Duckdb submodule not initialized`
- The package version is the `release_version` in `pyproject.toml` followed by `.dev` and the time of the latest commit across the repository and the submodule, as `yymmddhhmm` in UTC, for example `2.0.0.dev2610201323`. No git tags are read. The DuckDB version is computed from the number of commits in the submodule, so do not make a shallow clone

2. Install Astral uv

Install uv version >= 0.8.0.

Development Environment Setup

1. Platform-Specific Setup

All Platforms

- Python 3.11 or later; free-threaded builds of CPython are not supported
- uv >= 0.8.0 required
- CMake, Ninja, nanobind and the NumPy headers (installed via UV)
- C++ compiler toolchain

Linux On Ubuntu, install the following packages:

```
sudo apt-get update
sudo apt-get install ccache gcc-16 g++-16
```

On Fedora, Red Hat, Amazon Linux, etc., run:

```
sudo dnf install ccache gcc gcc-c++
```

Make sure the compiler are set using:

```
export CC=gcc-16
export CXX=g++-16
```

macOS Make sure that you have Xcode command line tools installed:

```
xcode-select --install
```

Windows Ensure you have the following installed:

- Visual Studio 2019+ with C++ support
- Git for Windows

2. Install Dependencies and Build

Set up the development environment in two steps:

```
# Install the build dependencies without building the project
uv sync --only-group build --no-install-project -p 3.13

# Build and install the project without build isolation
uv sync --no-build-isolation -v --reinstall -p 3.13
```

The result is an editable Debug build in `build/debug/`. Editable builds use the full CPython API; only wheels use the stable ABI.

Why two steps?

- `uv sync` performs editable installs by default with scikit-build-core using a persistent build-dir
- The build happens in an isolated, ephemeral environment where cmake's paths point to non-existing directories
- Installing dependencies first, then building without isolation ensures proper cmake integration

3. Enable Pre-Commit Hooks

We run a number of linting, formatting and type-checking in CI. You can run all of these manually, but to make your life easier you can install the exact same checks we run in CI as git hooks with pre-commit. Install it pinned to Python 3.12, so the hooks do not depend on your build interpreter:

```
uv tool install --python 3.12 pre-commit
pre-commit install
```

This will run all required checks before letting your commit pass. To run them across the tree, use `pre-commit run --all-files`.

You can also install a post-checkout hook that always runs `git submodule update --init --recursive`. When you change branches between main and a bugfix branch, this makes sure the `duckdb` submodule is always correctly initialized:

```
pre-commit install --hook-type post-checkout
```

4. Verify Installation

```
uv run python -c "import duckdb; print(duckdb.sql('SELECT 42').fetchall())"
```

Development Workflow

Running Tests

Run all tests:

```
uv run --no-build-isolation pytest ./tests --verbose
```

Run fast tests only (excludes slow directory):

```
uv run --no-build-isolation pytest ./tests --verbose --ignore=./tests/slow
```

Test Coverage

Build with coverage (compiles the extension with `--coverage` into `build/coverage/` for C++ coverage), then run the tests under coverage:

```
COVERAGE=1 uv sync --no-build-isolation -v --reinstall -p 3.13
COVERAGE=1 uv run --no-build-isolation coverage run -m pytest ./tests --verbose
```

Check Python coverage:

```
uv run coverage html -d htmlcov-python
uv run coverage report --format=markdown
```

Check C++ coverage:

```
uv run gcovr \
  --gcov-ignore-errors all \
  --gcov-ignore-parse-errors=negative_hits.warn_once_per_file \
  --root "$PWD" \
  --filter "${PWD}/src" \
  --exclude '.*/\.cache/.*' \
  --gcov-exclude '.*/\.cache/.*' \
  --gcov-exclude '.*/external/.*' \
  --gcov-exclude '.*/site-packages/.*' \
  --exclude-unreachable-branches \
  --exclude-throw-branches \
  --html --html-details -o coverage-cpp.html \
  build/coverage/src \
  --print-summary
```

Building Wheels

Build an sdist and a wheel for your system:

```
uv build
```

On CPython 3.12 and later, the wheel is tagged `cp312-abi3`: it uses the stable ABI and serves 3.12 and every later version. On CPython 3.11, the wheel is specific to that version. Build for a specific Python version:

```
uv build -p 3.11
```

To install the wheel, run:

```
uv pip install dist/duckdb-*.whl
```

Cleaning Build Artifacts

```
uv cache clean
rm -rf build .venv uv.lock
```

IDE Setup (CLion)

For CLion users, the project can be configured for C++ debugging of the Python extension:

CMake Profile Configuration

In **Settings** → **Build, Execution, Deployment** → **CMake**, create a Debug profile:

- **Name:** Debug

- **Build type:** Debug
- **Generator:** Ninja
- **CMake Options:**

```
-DCMAKE_PREFIX_PATH=$CMakeProjectDir$/.venv;$CMAKE_PREFIX_PATH
```

Python Debug Configuration

Create a **CMake Application** run configuration:

- **Name:** Python Debug
- **Target:** `All targets`
- **Executable:** <PROJECT_DIR>/.venv/bin/python3{:.language-sql .highlight}
- **Program arguments:** `$FilePath$`
- **Working directory:** `$ProjectFileDir$`

This allows setting C++ breakpoints and debugging Python scripts that use the DuckDB extension.

Debugging

Command Line Debugging

Set breakpoints and debug with lldb:

```
# Example Python script (test.py)
# import duckdb
# print(duckdb.sql("SELECT * FROM range(1000)").df())

lldb -- .venv/bin/python3 test.py
```

In lldb:

```
# Set breakpoint (library loads when imported)
(lldb) br s -n duckdb::DuckDBPyResult::FetchDF
(lldb) r
```

Cross-Platform Testing

You can run the Packaging workflow manually on your fork for any branch via the GitHub Actions web interface. Its inputs choose a minimal set of wheels or all of them, the test suite to run (none, fast or all), the commits of the package and of the DuckDB submodule to build and a forced package or DuckDB version.

Troubleshooting

Build Issues

Missing submodule: If the build fails with `RuntimeError: Duckdb submodule not initialized`, initialize the submodule:

```
git submodule update --init --recursive
```

R

This page contains instructions for building the R client library.

Parallelizing the Build

Problem: By default, R compiles packages using a single thread, which causes the build to be slow.

Solution: To parallelize the compilation, create or edit the `~/.R/Makevars` file, and add a line like the following:

```
MAKEFLAGS = -j8
```

The above will parallelize the compilation using 8 threads. On Linux/macOS, you can add the following to use all of the machine's threads:

```
MAKEFLAGS = -j$(nproc)
```

However, note that, the more threads that are used, the higher the RAM consumption. If the system runs out of RAM while compiling, then the R session will crash.

Troubleshooting

This page contains solutions to common problems reported by users. If you have platform-specific issues, make sure you also consult the troubleshooting guide for your platform such as the one for Linux builds.

The Build Runs Out of Memory

Problem: Ninja parallelizes the build, which can cause out-of-memory issues on systems with limited resources. These issues have also been reported to occur on Alpine Linux, especially on machines with limited resources.

Solution: Avoid using Ninja by setting the Makefile generator to empty via `GEN=`:

```
GEN= make
```

Unofficial and Unsupported Platforms

> **Warning.** The platforms listed on this page are not officially supported. The build instructions are provided on a best-effort basis. Community contributions are very welcome.

DuckDB is built and distributed for several platforms with different levels of support. DuckDB *can be built* for other platforms with varying levels of success. This page provides an overview of these with the intent to clarify which platforms can be expected to work.

32-bit Architectures

32-bit architectures are officially not supported but it is possible to build DuckDB manually for some of these platforms. For example, see the build instructions for 32-bit Raspberry Pi boards.

Note that 32-bit platforms are limited to using 4 GiB RAM due to the amount of addressable memory.

Big-Endian Architectures

Big-endian architectures (such as PowerPC) are not supported by DuckDB. While DuckDB can likely be built on such architectures, the resulting binary may exhibit correctness errors on certain operations. Therefore, its use is not recommended.

RISC-V Architectures

Native Build (Recommended)

DuckDB builds natively on RISC-V 64-bit boards without any special flags. Tested on a BananaPi F3 (SpacemiT K1, rv64gc, 8 cores @ 1.6 GHz, 16 GB RAM) running Debian Trixie:

```
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build
git clone --depth=1 https://github.com/duckdb/duckdb
cd duckdb
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make -j$(nproc)
```

Build time is approximately 2 hours on an 8-core SpacemiT K1. The resulting binary works out of the box:

```
./duckdb -c "SELECT 'Hello from RISC-V' AS message;"
```

```
┌──────────────────┐
│     message      │
│     varchar      │
├──────────────────┤
│ Hello from RISC-V │
└──────────────────┘
```

Aggregation queries work as expected:

```
./duckdb -c "CREATE TABLE test AS SELECT range AS id, range * 3.14 AS value FROM range(1000);
SELECT count(*) AS cnt, round(avg(value), 2) AS avg_val FROM test;"
```

```
┌───────┬─────────┐
│  cnt  │ avg_val │
│ int64 │ double  │
├───────┼─────────┤
│  1000 │ 1568.43 │
└───────┴─────────┘
```

The DuckDB Python package also builds successfully from source on riscv64:

```
pip install duckdb --no-binary duckdb
```

Build with RVV (RISC-V Vector Extension)

On boards with RVV 1.0 support (e.g., SpacemiT K3 with vlen 256), you can enable vector instructions for better performance. The user "LivingLinux" on Bluesky built DuckDB with RVV and published a video about it:

```
GEN=ninja \
    CC='gcc-14 -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    CXX='g++-14 -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    BUILD_EXTENSIONS='fts' \
    make
```

Cross-Compilation

For those who do not have RISC-V hardware, you can cross-compile DuckDB using the riscv-gnu-toolchain:

```
GEN=ninja \
    CC='riscv64-linux-gnu-gcc -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    CXX='riscv64-linux-gnu-g++ -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    make
```

For more reference information on DuckDB RISC-V cross-compiling, see the mocusez/duckdb-riscv-ci and DuckDB Pull Request #16549.

Known Issues

Platform Reported Incorrectly

As of version 1.5.2, DuckDB incorrectly reports the platform as `amd64` for unsupported platforms such as RISC-V. This causes is to download an incorrect extension binary. Currently, extensions are only available for officially supported platforms, so to use an extension, you need to build it e.g. using the `BUILD_EXTENSIONS` flag.

Benchmark Suite

DuckDB has an extensive benchmark suite. When making changes that have potential performance implications, it is important to run these benchmarks to detect potential performance regressions.

Getting Started

To build the benchmark suite, run the following command in the DuckDB repository:

```
BUILD_BENCHMARK=1 BUILD_EXTENSIONS='tpch' make
```

Listing Benchmarks

To list all available benchmarks, run:

```
build/release/benchmark/benchmark_runner --list
```

Running Benchmarks

Running a Single Benchmark

To run a single benchmark, issue the following command:

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark
```

The output will be printed to `stdout` in CSV format, in the following format:

```
name    run timing
benchmark/micro/nulls/no_nulls_addition.benchmark   1   0.121234
benchmark/micro/nulls/no_nulls_addition.benchmark   2   0.121702
benchmark/micro/nulls/no_nulls_addition.benchmark   3   0.122948
benchmark/micro/nulls/no_nulls_addition.benchmark   4   0.122534
benchmark/micro/nulls/no_nulls_addition.benchmark   5   0.124102
```

You can also specify an output file using the `--out` flag. This will write only the timings (delimited by newlines) to that file.

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark
--out=timings.out
```

The output will contain the following:

```
0.182472
0.185027
0.184163
0.185281
0.182948
```

Running Multiple Benchmarks Using a Regular Expression

You can also use a regular expression to specify which benchmarks to run. Be careful of shell expansion of certain regex characters (e.g., `*` will likely be expanded by your shell, hence this requires proper quoting or escaping).

```
build/release/benchmark/benchmark_runner "benchmark/micro/nulls/.*"
```

Running All Benchmarks Not specifying any argument will run all benchmarks.

```
build/release/benchmark/benchmark_runner
```

Other Options The `--info` flag gives you some other information about the benchmark.

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark --info
```

```
display_name:NULL Addition (no nulls)
group:micro
subgroup:nulls
```

The `--query` flag will print the query that is run by the benchmark.

```
SELECT min(i + 1) FROM integers;
```

The `--profile` flag will output a query tree.

Creating Benchmarks

Some development work is around performance, and including a benchmark along with the other tests not only validates any improvements, but also prevents future performance regressions in the feature.

Benchmark Example

To illustrate how to create a benchmark file, we can look at the benchmark for the `FILL` window function. (The `FILL` function linearly interpolates missing values in an ordered partition.)

Benchmarks are similar to unit test files, and have the same type of header.

```
# name: benchmark/micro/window/window_fill.benchmark
# description: Measure the performance of FILL
# group: [window]
```

The `make format-head` command can ensure that the header has the expected structure and prevent tidy check errors.

Below this header, there are a set of keywords summarizing the benchmark.

```
name FillPerformance
group micro
subgroup window
```

While some benchmarks run a single query, it can often be useful to *parameterize* a benchmark using the `argument` keyword. This allows the benchmark to be run with different settings, such as data volume. For the `FILL` benchmark, there are three arguments:

```
argument sf 10
argument errors 0.1
argument keys 4
```

For `FILL` these are

- The scale factor (millions of rows per partition)
- The error rate (fraction of the values that are missing)
- The number of partitions.

Benchmarks generally require some data preparation before running the query. Data preparation is done in the `load` section of the benchmark file. For the `FILL` benchmark, we create a table using the parameters and a random number generator.

```
load
select setseed(0.8675309);
create or replace table data as (
    select
        k::TINYINT as k,
        (case when random() > ${errors} then m - 1704067200000 else null end) as v,
        m,
    from range(1704067200000, 1704067200000 + ${sf} * 1_000_000 * 10, 10) times(m)
    cross join range(${keys}) keys(k)
);
```

The `argument` parameters are expanded in the query, similar to the way that `foreach` values are expanded in unit tests. Note that we can issue multiple SQL statements in the `load` section.

Once the data is prepared, we are finally ready to specify the query we will benchmark! This is done in the `run` section, and the restrictions are the same as for a unit test (e.g., no blank lines, etc.) For the `FILL` benchmark, we want to find all places where the interpolation fails:

```
run
SELECT
    m,
    k,
    fill(v) OVER (PARTITION BY k ORDER BY m) as v
FROM
    data
qualify v <> m - 1704067200000;
```

If the interpolation is correct, then we will have no output, no matter the scale. We can check this with the final `result` clause, which has the same syntax as a unit test:

```
result III
```

By providing no output rows, we can check the correctness of the query as well as its performance.

There are many other examples in the top level `benchmark/` directory, and you may want to have a look to discover some other techniques.

Testing

Overview

How is DuckDB Tested?

Testing is vital to make sure that DuckDB works properly and keeps working properly. For that reason, we put a large emphasis on thorough and frequent testing:

- We run a batch of small tests on every commit using GitHub Actions, and run a more exhaustive batch of tests on pull requests and commits in the `main` branch.
- We use a fuzzer, which automatically reports issues found through fuzzing DuckDB.
- We use SQLsmith for generating random queries.

sqllogictest Introduction

For testing plain SQL, we use an extended version of the SQL logic test suite, adopted from SQLite. Every test is a single self-contained file located in the `test/sql` directory. To run tests located outside of the default `test` directory, specify `--test-dir <root_directory>` and make sure provided test file paths are relative to that root directory.

The test describes a series of SQL statements, together with either the expected result, a `statement ok` indicator, or a `statement error` indicator. An example of a test file is shown below:

```
# name: test/sql/projection/test_simple_projection.test
# group [projection]

# enable query verification
statement ok
PRAGMA enable_verification

# create table
statement ok
CREATE TABLE a (i INTEGER, j INTEGER);

# insertion: 1 affected row
statement ok
INSERT INTO a VALUES (42, 84);

query II
SELECT * FROM a;
----
42  84
```

In this example, three statements are executed. The first statements are expected to succeed (prefixed by `statement ok`). The third statement is expected to return a single row with two columns (indicated by `query II`). The values of the row are expected to be 42 and 84 (separated by a tab character). For more information on query result verification, see the result verification section.

The top of every file should contain a comment describing the name and group of the test. The name of the test is always the relative file path of the file. The group is the folder that the file is in. The name and group of the test are relevant because they can be used to execute *only* that test in the unittest group. For example, if we wanted to execute *only* the above test, we would run the command `unittest test/sql/projection/test_simple_projection.test`. If we wanted to run all tests in a specific directory, we would run the command `unittest "[projection]"`.

Any tests that are placed in the `test` directory are automatically added to the test suite. Note that the extension of the test is significant. The sqllogictests should either use the `.test` extension, or the `.test_slow` extension. The `.test_slow` extension indicates that the test takes a while to run, and will only be run when all tests are explicitly run using `unittest *`. Tests with the extension `.test` will be included in the fast set of tests.

Query Verification

Many simple tests start by enabling query verification. This can be done through the following PRAGMA statement:

```
statement ok
PRAGMA enable_verification
```

Query verification performs extra validation to ensure that the underlying code runs correctly. The most important part of that is that it verifies that optimizers do not cause bugs in the query. It does this by running both an unoptimized and optimized version of the query, and verifying that the results of these queries are identical.

Query verification is very useful because it not only discovers bugs in optimizers, but also finds bugs in e.g., join implementations. This is because the unoptimized version will typically run using cross products instead. Because of this, query verification can be very slow to do when working with larger datasets. It is therefore recommended to turn on query verification for all unit tests, except those involving larger datasets (more than ~10-100 rows).

Editors & Syntax Highlighting

The sqllogictests are not exactly an industry standard, but several other systems have adopted them as well. Parsing sqllogictests is intentionally simple. All statements have to be separated by empty lines. For that reason, writing a syntax highlighter is not extremely difficult.

A syntax highlighter exists for Visual Studio Code. We have also made a fork that supports the DuckDB dialect of the sqllogictests. You can use the fork by installing the original, then copying the `syntaxes/sqllogictest.tmLanguage.json` into the installed extension (on macOS this is located in `~/.vscode/extensions/benesch.sqllogictest-0.1.1`).

A syntax highlighter is also available for CLion. It can be installed directly on the IDE by searching SQLTest on the marketplace. A GitHub repository is also available, with extensions and bug reports being welcome.

Temporary Files

For some tests (e.g., CSV/Parquet file format tests) it is necessary to create temporary files. Any temporary files should be created in the temporary testing directory. This directory can be used by placing the string `__TEST_DIR__` in a query. This string will be replaced by the path of the temporary testing directory.

```
statement ok
COPY csv_data TO '__TEST_DIR__/output_file.csv.gz' (COMPRESSION gzip);
```

Require & Extensions

To avoid bloating the core system, certain functionality of DuckDB is available only as an extension. Tests can be built for those extensions by adding a `require` field in the test. If the extension is not loaded, any statements that occur after the require field will be skipped. Examples of this are `require parquet` or `require icu`.

Another usage is to limit a test to a specific vector size. For example, adding `require vector_size 512` to a test will prevent the test from being run unless the vector size is greater than or equal to 512. This is useful because certain functionality is not supported for low vector sizes, but we run tests using a vector size of 2 in our CI.

Writing Tests

Development and Testing

It is crucial that any new features that get added have correct tests that not only test the "happy path", but also test edge cases and incorrect usage of the feature. In this section, we describe how DuckDB tests are structured and how to make new tests for DuckDB.

The tests can be run by running the `unittest` program located in the `test` folder. For the default compilations this is located in either `build/release/test/unittest` (release) or `build/debug/test/unittest` (debug).

Philosophy

When testing DuckDB, we aim to route all the tests through SQL. We try to avoid testing components individually because that makes those components more difficult to change later on. As such, almost all of our tests can (and should) be expressed in pure SQL. There are certain exceptions to this, which we will discuss in Catch Tests. However, in most cases you should write your tests in plain SQL.

Frameworks

SQL tests should be written using the sqllogictest framework.

C++ tests can be written using the Catch framework.

Client Connector Tests

DuckDB also has tests for various client connectors. These are generally written in the relevant client language, and can be found in `tools/*/tests`. They also double as documentation of what should be doable from a given client.

Functions for Generating Test Data

DuckDB has built-in functions for generating test data.

`test_all_types` Function

The `test_all_types` table function generates a table whose columns correspond to types (`BOOL`, `TINYINT`, etc.). The table has three rows encoding the minimum value, the maximum value, and the `NULL` value for each type.

```
FROM test_all_types();
```

```
┌─────────┬─────────┬──────────┬─────────────┬─────────────────────┬──────────────────────┬───┬─────────────────────┐
│  bool   │ tinyint │ smallint │     int     │       bigint        │       hugeint        │ … │       struct        │
│ boolean │  int8   │  int16   │    int32    │        int64        │        int128        │   │ struct(a integer, … │
├─────────┼─────────┼──────────┼─────────────┼─────────────────────┼──────────────────────┼───┼─────────────────────┤
│ false   │    -128 │   -32768 │ -2147483648 │ -9223372036854775808 │ -170141183460469231… │ … │ {'a': NULL, 'b': N… │
│ true    │     127 │    32767 │  2147483647 │  9223372036854775807 │ 170141183460469231… │ … │ {'a': 42, 'b': 🦆… │
│ NULL    │    NULL │     NULL │        NULL │                NULL │                 NULL │ … │ NULL                │
├─────────┴─────────┴──────────┴─────────────┴─────────────────────┴──────────────────────┴───┴─────────────────────┤
│ 3 rows                                                                                  44 columns (11 shown) │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

`test_vector_types` Function

The `test_vector_types` table function takes *n* arguments `col1`, ..., `coln` and an optional `BOOLEAN` argument `all_flat`. The function generates a table with *n* columns `test_vector`, `test_vector2`, ..., `test_vectorn`. In each row, each field contains values conforming to the type of their respective column.

```
FROM test_vector_types(NULL::BIGINT);
```

```
┌─────────────────────┐
│     test_vector     │
│        int64        │
├─────────────────────┤
│ -9223372036854775808 │
│  9223372036854775807 │
│                 NULL │
│         ·           │
└─────────────────────┘
```

```
FROM test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP);
```

```
┌──────────────────────────────────────────────────────────────────────────────┬──────────────────────────────┐
│                                 test_vector                                  │         test_vector2         │
│                 struct(i integer, j varchar, k double)                       │          timestamp           │
├──────────────────────────────────────────────────────────────────────────────┼──────────────────────────────┤
│ {'i': -2147483648, 'j': 🦆🦆🦆🦆🦆, 'k': -1.7976931348623157e+308} │ 290309-12-22 (BC) 00:00:00   │
│ {'i': 2147483647, 'j': goo\0se, 'k': 1.7976931348623157e+308}                │ 294247-01-10 04:00:54.775806 │
│ {'i': NULL, 'j': NULL, 'k': NULL}                                            │ NULL                         │
│                                          ·                                   │                              │
└──────────────────────────────────────────────────────────────────────────────┴──────────────────────────────┘
```

`test_vector_types` has an optional argument called `all_flat` of type `BOOL`. This only affects the internal representation of the vector.

```
FROM test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP, all_flat = true);
-- the output is the same as above but with a different internal representation
```

Unit Tester Configuration

There are three ways of setting options for the unit tester:

- Command Line Arguments
- Configuration File
- Environment Variables
- Configuration Options List

See the Configuration Options List for the full list of available options.

Command Line Arguments

See the Configuration Options List for all available options.

```
# booleans
unittest --checkpoint-on-shutdown
unittest --no-checkpoint-on-shutdown

# other types
unittest --checkpoint-wal-size 100000
```

Command line options `--settings`, `--on-init` and `--init-script` can be used to specify DuckDB configuration.

Configuration File

A JSON file that can be passed in using `--test-config`. See the Configuration Options List for all available options, replacing - with _ .

Example:

```
{
   "checkpoint_wal_size": 100000,
   "checkpoint_on_shutdown": "false"
}
```

The following additional options can also be used in the configuration file:

Option	Description
`extends`	List of config files to extend from
`inherit_skip_tests`	Path of config to inherit `skip_tests` from
`skip_tests`	Tests to be skipped

Example:

```
{
    "extends": ["base_config.json"],
    "inherit_skip_tests": "test/configs/force_storage.json",
    "skip_tests": [
        {
            "reason": "Contains explicit use of the memory catalog.",
            "paths": [
                "test/sql/attach/attach_use_rollback.test",
                "test/sql/function/list/lambdas/arrow/warn_deprecated_arrow.test",
                "test/sql/cte/warn_deprecated_union_in_using_key.test"
            ]
        }
    ]
}
```

Environment Variables

Most Configuration Options can also be set as environment variables, prefixed with DUCKDB_TEST_, upper-cased with underscores:

Example:

```
DUCKDB_TEST_CHECKPOINT_ON_SHUTDOWN=1 unittest
```

Configuration Options List

The following configuration options are available:

Option	Description
`--autoloading <none|available|all>`	Loading strategy for extensions not bundled in
`--base-config <path>`	Config file to load and base initial settings on
`--block-size <num>`	Block Allocation Size; must be a power of 2

Option	Description
`--checkpoint-on-shutdown`, `--no-checkpoint-on-shutdown`	Whether or not to checkpoint on database shutdown
`--checkpoint-wal-size <num>`	Size in bytes after which to trigger automatic checkpointing
`--comment <string>`	Extra free form comment line
`--data-dir <path>`	Root dir for test data; default WORKING_DIR/data. Relative values resolve per test cwd; absolute/remote (e.g. `az://...`) values are used verbatim.
`--database-destroy <on|off|on-success>`	Default on-success: keep on failure
`--description <string>`	Config description
`--emit-on-skip`	Print a `SKIP_TEST` marker per skipped test
`--emit-test-events`	Writes JSON with test events to stderr
`--force-reload`	Force reload the test framework
`--force-restart`, `--no-force-restart`	Force restart the database between runs
`--force-storage`	Test with persistent storage file
`--init-script <path>`	SQL script to execute on init
`--initial-db <path>`	Initial database path
`--keep-home`	Use `$HOME/USERPROFILE` instead of the sandboxed temp-dir home
`--max-test-threads <num>`	Max threads to be used by the test runner itself (for e.g. concurrent loop)
`--max-threads <num>`	Max threads to use during tests
`--on-cleanup <string>`	SQL statements to execute on test end
`--on-init <string>`	SQL statements to execute on init
`--on-load <string>`	SQL statements to execute on explicit load
`--on-new-connection <string>`	SQL statements to execute on connection
`--one-initialize`	Initialize buffers with all 1; also see `--zero-initialize`
`--require <extension_name>`	If the extension is missing: fail rather than skip tests that require it
`--run-id <'auto'|<string>>`	Override per-run identity; the `RUN_ID` path level; shared across a run's tests; auto = generate
`--select-tag <string>`	Select tests which match named tag (as singleton set; multiple sets are OR'd)
`--select-tag-set <comma-separated strings>`	Select tests which match *all* named tags (multiple sets are OR'd)
`--settings <list[struct]>`	Set DuckDB configuration, struct should have keys name and value
`--skip-compiled`, `--no-skip-compiled`	Skip compiled tests

Option	Description
`--skip-error-messages <comma-separated strings>`	Skip rather than fail tests that fail with a message that contains any of these substrings
`--skip-tag <string>`	Skip tests which match named tag (as singleton set; multiple sets are OR'd)
`--skip-tag-set <comma-separated strings>`	Skip tests which match *all* named tags (multiple sets are OR'd)
`--sort-style <none|rowsort|valuesort>`	Default sort style if none is configured in the test
`--statically-loaded-extensions <comma-separated strings>`	Extensions to be loaded (from the statically available ones)
`--stdin`	Read a single sqllogictest script from stdin
`--storage-fuzzer`, `--no-storage-fuzzer`	Run storage fuzzer tests
`--storage-version <string>`	Database storage version to use by default
`--summarize-failures`, `--no-summarize-failures`	Print a summary of all test failures after running
`--temp-dir-base <path>`	Override root of the temp-dir tree; may be local or remote (e.g. `s3://`)
`--temp-dir-create <never|on-absent|always>`	Default on-absent: mkdir -p, no clobber
`--temp-dir-destroy <never|on-success|always>`	Default on-success: keep on failure
`--temp-dir-run-id <on|off>`	Is RUN_ID a path level? (default on)
`--temp-dir-test-id <on|off>`	Is TEST_ID (the test name) a path level? (default on)
`--test-config <path>`	Path to a configuration file
`--test-dir <path>`	Override default working directory of test runner, available in tests as {WORKING_DIRECTORY}
`--test-env <list[struct]>`	The test variables, struct should have keys env_name and env_value
`--test-memory-leaks`, `--no-test-memory-leaks`	Run memory leak tests
`--verify-vector <string>`	Run vector verification for a specific vector type (options: dictionary_expression, dictionary_operator, constant_operator, sequence_operator, nested_shuffle, variant_vector)
`--zero-initialize`	Initialize buffers with all 0; also see `--one-initialize`

Debugging

The purpose of the tests is to figure out when things break. Inevitably changes made to the system will cause one of the tests to fail, and when that happens the test needs to be debugged.

First, it is always recommended to run in debug mode. This can be done by compiling the system using the command `make debug`. Second, it is recommended to only run the test that breaks. This can be done by passing the filename of the breaking test to the test suite as a command line parameter (e.g., `build/debug/test/unittest test/sql/projection/test_simple_projection.test`). For more options on running a subset of the tests see the Triggering which tests to run section.

After that, a debugger can be attached to the program and the test can be debugged. In the sqllogictests it is normally difficult to break on a specific query, however, we have expanded the test suite so that a function called `query_break` is called with the line number `line` as parameter for every query that is run. This allows you to put a conditional breakpoint on a specific query. For example, if we want to break on line number 43 of the test file we can create the following break point:

```
gdb: break query_break if line==43
lldb: break s -n query_break -c line==43
```

You can also skip certain queries from executing by placing `mode skip` in the file, followed by an optional `mode unskip`. Any queries between the two statements will not be executed.

Triggering Which Tests to Run

When running the unittest program, by default all the fast tests are run. A specific test can be run by adding the name of the test as an argument. For the sqllogictests, this is the relative path to the test file. To run only a single test:

```
build/debug/test/unittest test/sql/projection/test_simple_projection.test
```

All tests in a given directory can be executed by providing the directory as a parameter with square brackets. To run all tests in the "projection" directory:

```
build/debug/test/unittest "[projection]"
```

All tests, including the slow tests, can be run by running the tests with an asterisk. To run all tests, including the slow tests:

```
build/debug/test/unittest "*"
```

We can run a subset of the tests using the `--start-offset` and `--end-offset` parameters. To run the tests 200...250:

```
build/debug/test/unittest --start-offset=200 --end-offset=250
```

These are also available in percentages. To run tests 10% - 20%:

```
build/debug/test/unittest --start-offset-percentage=10 --end-offset-percentage=20
```

The set of tests to run can also be loaded from a file containing one test name per line, and loaded using the `-f` command.

```
cat test.list
```

```
test/sql/join/full_outer/test_full_outer_join_issue_4252.test
test/sql/join/full_outer/full_outer_join_cache.test
test/sql/join/full_outer/test_full_outer_join.test
```

To run only the tests labeled in the file:

```
build/debug/test/unittest -f test.list
```

Result Verification

The standard way of verifying results of queries is using the `query` statement, followed by the letter `I` times the number of columns that are expected in the result. After the query, four dashes (`----`) are expected followed by the result values separated by tabs. For example,

```
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20
```

For legacy reasons the letters R and T are also accepted to denote columns.

> **Deprecated.** DuckDB deprecated the usage of types in the sqllogictest. The DuckDB test runner does not use or need them internally – therefore, only `I` should be used to denote columns.

NULL Values and Empty Strings

Empty lines have special significance for the SQLLogic test runner: they signify an end of the current statement or query. For that reason, empty strings and NULL values have special syntax that must be used in result verification. NULL values should use the string `NULL`, and empty strings should use the string `(empty)`, e.g.:

```
query II
SELECT NULL, ''
----
NULL
(empty)
```

Error Verification

In order to signify that an error is expected, the `statement error` indicator can be used. The `statement error` also takes an optional expected result – which is interpreted as the *expected error message*. Similar to `query`, the expected error should be placed after the four dashes (`----`) following the query. The test passes if the error message *contains* the text under `statement error` – the entire error message does not need to be provided. It is recommended that you only use a subset of the error message, so that the test does not break unnecessarily if the formatting of error messages is changed.

```
statement error
SELECT * FROM non_existent_table;
----
Table with name non_existent_table does not exist!
```

Regex

In certain cases result values might be very large or complex, and we might only be interested in whether or not the result *contains* a snippet of text. In that case, we can use the `<REGEX>:` modifier followed by a certain regex. If the result value matches the regex the test is passed. This is primarily used for query plan analysis.

```
query II
EXPLAIN SELECT tbl.a FROM 'data/parquet-testing/arrow/alltypes_plain.parquet' tbl(a) WHERE a = 1 OR a = 2
----
physical_plan   <REGEX>:.*PARQUET_SCAN.*Filters: a=1 OR a=2.*
```

If we instead want the result *not* to contain a snippet of text, we can use the `<!REGEX>:` modifier.

File

As results can grow quite large, and we might want to re-use results over multiple files, it is also possible to read expected results from files using the `<FILE>` command. The expected result is read from the given file. As convention the file path should be provided as relative to the root of the GitHub repository.

```
query I
PRAGMA tpch(1)
----
<FILE>:extension/tpch/dbgen/answers/sf1/q01.csv
```

Row-Wise vs. Value-Wise Result Ordering

The result values of a query can be either supplied in row-wise order, with the individual values separated by tabs, or in value-wise order. In value wise order the individual *values* of the query must appear in row, column order each on an individual line. Consider the following example in both row-wise and value-wise order:

```
# row-wise
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20

# value-wise
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42
84
10
20
```

Hashes and Outputting Values

Besides direct result verification, the sqllogic test suite also has the option of using MD5 hashes for value comparisons. A test using hashes for result verification looks like this:

```
query I
SELECT g, string_agg(x,',') FROM strings GROUP BY g
----
200 values hashing to b8126ea73f21372cdb3f2dc483106a12
```

This approach is useful for reducing the size of tests when results have many output rows. However, it should be used sparingly, as hash values make the tests more difficult to debug if they do break.

After it is ensured that the system outputs the correct result, hashes of the queries in a test file can be computed by adding `mode output_hash` to the test file. For example:

```
mode output_hash

query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20
```

The expected output hashes for every query in the test file will then be printed to the terminal, as follows:

```
================================================================================
SQL Query
```

```
SELECT 42, 84 UNION ALL SELECT 10, 20;
================================================================================
4 values hashing to 498c69da8f30c24da3bd5b322a2fd455
================================================================================
```

In a similar manner, `mode output_result` can be used to force the program to print the result to the terminal for every query run in the test file.

Result Sorting

Queries can have an optional field that indicates that the result should be sorted in a specific manner. This field goes in the same location as the connection label. Because of that, connection labels and result sorting cannot be mixed.

The possible values of this field are `nosort`, `rowsort` and `valuesort`. An example of how this might be used is given below:

```
query I rowsort
SELECT 'world' UNION ALL SELECT 'hello'
----
hello
world
```

In general, we prefer not to use this field and rely on `ORDER BY` in the query to generate deterministic query answers. However, existing sqllogictests use this field extensively, hence it is important to know of its existence.

Query Labels

Another feature that can be used for result verification are `query labels`. These can be used to verify that different queries provide the same result. This is useful for comparing queries that are logically equivalent, but formulated differently. Query labels are provided after the connection label or sorting specifier.

Queries that have a query label do not need to have a result provided. Instead, the results of each of the queries with the same label are compared to each other. For example, the following script verifies that the queries `SELECT 42+1` and `SELECT 44-1` provide the same result:

```
query I nosort r43
SELECT 42+1;
----

query I nosort r43
SELECT 44-1;
----
```

Persistent Testing

By default, all tests are run in in-memory mode (unless `--force-storage` is enabled). In certain cases, we want to force the usage of a persistent database. We can initiate a persistent database using the `load` command, and trigger a reload of the database using the `restart` command.

```
# load the DB from disk
load __TEST_DIR__/storage_scan.db

statement ok
CREATE TABLE test (a INTEGER);

statement ok
INSERT INTO test VALUES (11), (12), (13), (14), (15), (NULL)

# ...
```

```
restart

query I
SELECT * FROM test ORDER BY a
----
NULL
11
12
13
14
15
```

Note that by default the tests run with `SET wal_autocheckpoint = '0KB'{:.language-sql .highlight}` – meaning a checkpoint is triggered after every statement. WAL tests typically run with the following settings to disable this behavior:

```
statement ok
PRAGMA disable_checkpoint_on_shutdown

statement ok
SET wal_autocheckpoint = '1TB'
```

Loops

Loops can be used in sqllogictests when it is required to execute the same query many times but with slight modifications in constant values. For example, suppose we want to fire off 100 queries that check for the presence of the values `0...100` in a table:

```
# create the table 'integers' with values 0...100
statement ok
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);

# verify individually that all 100 values are there
loop i 0 100

# execute the query, replacing the value
query I
SELECT count(*) FROM integers WHERE i = ${i};
----
1

# end the loop (note that multiple statements can be part of a loop)
endloop
```

Similarly, `foreach` can be used to iterate over a set of values.

```
foreach partcode millennium century decade year quarter month day hour minute second millisecond
microsecond epoch

query III
SELECT i, date_part('${partcode}', i) AS p, date_part(['${partcode}'], i) AS st
FROM intervals
WHERE p <> st['${partcode}'];
----

endloop
```

`foreach` also has a number of preset combinations that should be used when required. In this manner, when new combinations are added to the preset, old tests will automatically pick up these new combinations.

Preset	Expansion
<compression>	none uncompressed rle bitpacking dictionary fsst chimp patas

Preset	Expansion
<signed>	tinyint smallint integer bigint hugeint
<unsigned>	utinyint usmallint uinteger ubigint uhugeint
<integral>	<signed> <unsigned>
<numeric>	<integral> float double
<alltypes>	<numeric> bool interval varchar json

> Use large loops sparingly. Executing hundreds of thousands of SQL statements will slow down tests unnecessarily. Do not use loops for inserting data.

Data Generation without Loops

Loops should be used sparingly. While it might be tempting to use loops for inserting data using insert statements, this will considerably slow down the test cases. Instead, it is better to generate data using the built-in `range` and `repeat` functions.

To create the table `integers` with the values `[0, 1, ..., 98, 99]`, run:

```
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);
```

To create the table `strings` with 100 times the value `hello`, run:

```
CREATE TABLE strings AS SELECT * FROM repeat('hello', 100) t1(s);
```

Using these two functions, together with clever use of cross products and other expressions, many different types of datasets can be efficiently generated. The `random()` function can also be used to generate random data.

An alternative option is to read data from an existing CSV or Parquet file. There are several large CSV files that can be loaded from the directory `test/sql/copy/csv/data/real` using a `COPY INTO` statement or the `read_csv_auto` function.

The TPC-H and TPC-DS extensions can also be used to generate synthetic data, using e.g. `CALL dbgen(sf = 1)` or `CALL dsdgen(sf = 1)`.

Multiple Connections

For tests whose purpose is to verify that the transactional management or versioning of data works correctly, it is generally necessary to use multiple connections. For example, if we want to verify that the creation of tables is correctly transactional, we might want to start a transaction and create a table in `con1`, then fire a query in `con2` that checks that the table is not accessible yet until committed.

We can use multiple connections in the sqllogictests using `connection labels`. The connection label can be optionally appended to any `statement` or `query`. All queries with the same connection label will be executed in the same connection. A test that would verify the above property would look as follows:

```
statement ok con1
BEGIN TRANSACTION

statement ok con1
CREATE TABLE integers (i INTEGER);

statement error con2
SELECT * FROM integers;
```

Concurrent Connections

Using connection modifiers on the statement and queries will result in testing of multiple connections, but all the queries will still be run *sequentially* on a single thread. If we want to run code from multiple connections *concurrently* over multiple threads, we can use the `concurrentloop` construct. The queries in `concurrentloop` will be run concurrently on separate threads at the same time.

```
concurrentloop i 0 10

statement ok
CREATE TEMP TABLE t2 AS (SELECT 1);

statement ok
INSERT INTO t2 VALUES (42);

statement ok
DELETE FROM t2

endloop
```

One caveat with `concurrentloop` is that results are often unpredictable – as multiple clients can hammer the database at the same time we might end up with (expected) transaction conflicts. `statement maybe` can be used to deal with these situations. `statement maybe` essentially accepts both a success, and a failure with a specific error message.

```
concurrentloop i 1 10

statement maybe
CREATE OR REPLACE TABLE t2 AS (SELECT -541240333865773480040026564265315351l4 FROM t2 LIMIT 70%);
----
write-write conflict

endloop
```

Catch C/C++ Tests

While we prefer the sqllogic tests for testing most functionality, for certain tests only SQL is not sufficient. This typically happens when you want to test the C++ API. When using pure SQL is really not an option it might be necessary to make a C++ test using Catch.

Catch tests reside in the test directory as well. Here is an example of a catch test that tests the storage of the system:

```
#include "catch.hpp"
#include "test_helpers.hpp"

TEST_CASE("Test simple storage", "[storage]") {
    auto config = GetTestConfig();
    unique_ptr<QueryResult> result;
    auto storage_database = TestCreatePath("storage_test");

    // make sure the database does not exist
    DeleteDatabase(storage_database);
    {
        // create a database and insert values
        DuckDB db(storage_database, config.get());
        Connection con(db);
        REQUIRE_NO_FAIL(con.Query("CREATE TABLE test (a INTEGER, b INTEGER);"));
        REQUIRE_NO_FAIL(con.Query("INSERT INTO test VALUES (11, 22), (13, 22), (12, 21), (NULL, NULL)"));
        REQUIRE_NO_FAIL(con.Query("CREATE TABLE test2 (a INTEGER);"));
        REQUIRE_NO_FAIL(con.Query("INSERT INTO test2 VALUES (13), (12), (11)"));
    }
    // reload the database from disk a few times
    for (idx_t i = 0; i < 2; i++) {
        DuckDB db(storage_database, config.get());
        Connection con(db);
```

```
        result = con.Query("SELECT * FROM test ORDER BY a");
        REQUIRE(CHECK_COLUMN(result, 0, {Value(), 11, 12, 13}));
        REQUIRE(CHECK_COLUMN(result, 1, {Value(), 22, 21, 22}));
        result = con.Query("SELECT * FROM test2 ORDER BY a");
        REQUIRE(CHECK_COLUMN(result, 0, {11, 12, 13}));
    }
    DeleteDatabase(storage_database);
}
```

The test uses the `TEST_CASE` wrapper to create each test. The database is created and queried using the C++ API. Results are checked using either `REQUIRE_FAIL` / `REQUIRE_NO_FAIL` (corresponding to statement ok and statement error) or `REQUIRE(CHECK_COLUMN(...))` (corresponding to query with a result check). Every test that is created in this way needs to be added to the corresponding `CMakeLists.txt`.

Internal Errors

Internal errors signal an assertion failure within DuckDB. They usually occur due to unexpected conditions or errors in the program's logic.

For example, running issue 17002 on DuckDB 1.2.1 results in an internal error.

```
INTERNAL Error:
Attempted to access index 3 within vector of size 3
```

> The issue is fixed in DuckDB 1.2.2 and newer versions.

After encountering an internal error, DuckDB enters "restricted mode", where any further operations will result in the following error message:

```
FATAL Error:
Failed: database has been invalidated because of a previous fatal error.
The database must be restarted prior to being used again.
```

To continue working with the same database, start a new DuckDB session on the same database.

The reason DuckDB enters "restricted mode" is because the database is in an undefined state. Internal errors are by definition bugs that should never happen so the code path is never tested.

If you encounter an internal error, please consider creating a minimal reproducible example and submitting an issue to the DuckDB issue tracker.

Internals

Overview of DuckDB Internals

> **Tip.** For a detailed explanation of DuckDB internals, see the Design and Implementation of DuckDB Internals ("DiDi") library entry.

On this page is a brief description of the internals of the DuckDB engine.

Parser

The parser converts a query string into the following tokens:

- `SQLStatement`
- `QueryNode`
- `TableRef`
- `ParsedExpression`

The parser is not aware of the catalog or any other aspect of the database. It will not throw errors if tables do not exist, and will not resolve **any** types of columns yet. It only transforms a query string into a set of tokens as specified.

ParsedExpression

The ParsedExpression represents an expression within a SQL statement. This can be e.g., a reference to a column, an addition operator or a constant value. The type of the ParsedExpression indicates what it represents, e.g., a comparison is represented as a `ComparisonExpression`.

ParsedExpressions do **not** have types, except for nodes with explicit types such as CAST statements. The types for expressions are resolved in the Binder, not in the Parser.

TableRef

The TableRef represents any table source. This can be a reference to a base table, but it can also be a join, a table-producing function or a subquery.

QueryNode

The QueryNode represents either (1) a `SELECT` statement, or (2) a set operation (i.e., `UNION`, `INTERSECT` or `DIFFERENCE`).

SQL Statement

The SQLStatement represents a complete SQL statement. The type of the SQL Statement represents what kind of statement it is (e.g., `StatementType::SELECT` represents a `SELECT` statement). A single SQL string can be transformed into multiple SQL statements in case the original query string contains multiple queries.

Binder

The binder converts all nodes into their **bound** equivalents. In the binder phase:

- The tables and columns are resolved using the catalog
- Types are resolved
- Aggregate/window functions are extracted

The following conversions happen:

- SQLStatement → `BoundStatement`
- QueryNode → `BoundQueryNode`
- TableRef → `BoundStatement`
- ParsedExpression → `Expression`

Logical Planner

The logical planner creates `LogicalOperator` nodes from the bound statements. In this phase, the actual logical query tree is created.

Optimizer

After the logical planner has created the logical query tree, the optimizers are run over that query tree to create an optimized query plan. The following query optimizers are run:

- **Expression Rewriter**: Simplifies expressions, performs constant folding
- **Filter Pushdown**: Pushes filters down into the query plan and duplicates filters over equivalency sets. Also prunes subtrees that are guaranteed to be empty (because of filters that statically evaluate to false).
- **Join Order Optimizer**: Reorders joins using dynamic programming. Specifically, the DPhyp algorithm from the paper Dynamic Programming Strikes Back is used.
- **Common Sub Expressions**: Extracts common subexpressions from projection and filter nodes to prevent unnecessary duplicate execution.
- **In Clause Rewriter**: Rewrites large static IN clauses to a MARK join or INNER join.

Column Binding Resolver

The column binding resolver converts logical `BoundColumnRefExpression` nodes that refer to a column of a specific table into `BoundReferenceExpression` nodes that refer to a specific index into the DataChunks that are passed around in the execution engine.

Physical Plan Generator

The physical plan generator converts the resulting logical operator tree into a `PhysicalOperator` tree.

Execution

In the execution phase, the physical operators are executed to produce the query result. DuckDB uses a push-based vectorized model, where `DataChunks` are pushed through the operator tree. For more information, see the talk Push-Based Execution in DuckDB.

Storage Versions and Format

Compatibility

Backward Compatibility

Backward compatibility refers to the ability of a newer DuckDB version to read storage files created by an older DuckDB version. Version 0.10 is the first release of DuckDB that supports backward compatibility in the storage format. DuckDB 0.10 can read and operate on files created by the previous DuckDB version – DuckDB 0.9.

For future DuckDB versions, our goal is to ensure that any DuckDB version released **after** can read files created by previous versions, starting from this release. We want to ensure that the file format is fully backward compatible. This allows you to keep data stored in DuckDB files around and guarantees that you will be able to read the files without having to worry about which version the file was written with or having to convert files between versions.

Forward Compatibility

Forward compatibility refers to the ability of an older DuckDB version to read storage files produced by a newer DuckDB version. DuckDB 0.9 is **partially** forward compatible with DuckDB 0.10. Certain files created by DuckDB 0.10 can be read by DuckDB 0.9.

Forward compatibility is provided on a **best effort** basis. While stability of the storage format is important – there are still many improvements and innovations that we want to make to the storage format in the future. As such, forward compatibility may be (partially) broken on occasion.

> **Warning.** DuckDB versions before 2.0 cannot read database files that use the v2.0.0 storage format, which DuckDB 2.0 uses by default for new database files. See Default Storage Version and the migration guide.

How to Move between Storage Formats

When you update DuckDB and open an old database file, you might encounter an error message about incompatible storage formats, pointing to this page. To move your database(s) to newer format you only need the older and the newer DuckDB executable.

Open your database file with the older DuckDB and run the SQL statement `EXPORT DATABASE 'tmp'`. This allows you to save the whole state of the current database in use inside folder `tmp`. The content of the `tmp` folder will be overridden, so choose an empty/non yet existing location. Then, start the newer DuckDB and execute `IMPORT DATABASE 'tmp'` (pointing to the previously populated folder) to load the database, which can be then saved to the file you pointed DuckDB to.

A Bash script to achieve this (to be adapted with the file names and executable locations) is the following

```
/older/duckdb mydata.old.db -c "EXPORT DATABASE 'tmp'"
/newer/duckdb mydata.new.db -c "IMPORT DATABASE 'tmp'"
```

After this, `mydata.old.db` will remain in the old format, `mydata.new.db` will contain the same data but in a format accessible by the more recent DuckDB version, and the folder `tmp` will hold the same data in a universal format as different files.

Check EXPORT documentation for more details on the syntax.

Default Storage Version

> New Starting with DuckDB 2.0, new database files use the v2.0.0 storage format by default.

DuckDB versions v1.0 to v1.5 create a DuckDB database file with version 64, corresponding to v1.0.0. Starting with DuckDB 2.0, new database files are created with version 69, corresponding to v2.0.0. On older versions you can opt in to a newer format by setting an explicit storage version.

> **Warning.** A file created with the v2.0.0 storage format cannot be opened by DuckDB versions prior to v2.0.0. To write a file that older versions can still read, create it with an explicit storage version.

What's New in the v2.0.0 Format

> New The v2.0.0 storage format, used by default from DuckDB 2.0, introduces several improvements over earlier versions.

Compared to earlier storage versions, the v2.0.0 format:

- loads column metadata lazily, so databases with wide tables and large indexes open faster and use less memory;
- enables the `DICT_FSST` string compression method by default (see Compression);
- stores row deletions more compactly; and
- performs stronger corruption validation when reading data.

Explicit Storage Versions

DuckDB 1.2.0 introduced the `STORAGE_VERSION` option, which allows explicitly specifying the storage version. Since DuckDB 2.0, new database files use the latest storage version by default, so the main use of this option is to create database files that older DuckDB versions can still read:

```
ATTACH 'file.db' (STORAGE_VERSION 'v1.2.0');
```

To initialize a database with the latest storage version, use:

```
ATTACH 'file.db' (STORAGE_VERSION 'latest');
```

With the command line client, you can use the `-storage-version` argument:

```
duckdb -storage-version v1.2.0 my_database.duckdb
```

To create a database with the latest storage version, use:

```
duckdb -storage-version latest my_database.duckdb
```

The storage version setting specifies the minimum DuckDB version that should be able to read the database file. When database files are written with this option, the resulting files cannot be opened by older DuckDB released versions than the specified version. They can be read by the specified version and all newer versions of DuckDB.

If you attach to DuckDB databases, you can query the storage versions using the following command:

```
SELECT database_name, tags
FROM duckdb_databases();
```

This shows the storage versions:

database_name varchar	tags map(varchar, varchar)
file1	{storage_version=v1.2.0+}
file2	{storage_version=v1.0.0+}
...	...

This means that `file2` can be opened by past DuckDB versions while `file1` is compatible only with `v1.2.0` (or future versions).

Setting the Storage Compatibility

The `storage_compatibility_version` configuration option can also be used to specify the storage version to use. It can be specified in various ways.

In the Python client, you have to specify it when connecting to a new database:

```python
import duckdb

duckdb.connect("file.db", config={'storage_compatibility_version': 'latest'})
# or
duckdb.connect("file.db", config={'storage_compatibility_version': 'v1.4.0'})
```

In the CLI and some other clients, set the configuration option as follows:

```sql
SET storage_compatibility_version = 'latest';
-- or
SET storage_compatibility_version = 'v1.4.0';
```

In the CLI client, the storage version can also be specified using the `-storage-version` command line argument for the entire CLI session.

Converting between Storage Versions

To convert from the new format to the old format for compatibility, use the following sequence in DuckDB 1.2.0+:

```sql
ATTACH 'file1.db';
ATTACH 'converted_file.db' (STORAGE_VERSION 'v1.0.0');
COPY FROM DATABASE file1 TO converted_file;
```

Storage Header

DuckDB files start with a `uint64_t` which contains a checksum for the main header, followed by four magic bytes (DUCK), followed by a version number in a `uint64_t`. Files created in the v2.0.0 format hold 999 there. In other files it is a storage version, but it can be older than the format the file was upgraded to, which only the database headers record.

```
hexdump -n 20 -C mydata.db
```

```
00000000  ac a3 60 b1 80 cb c3 8e  44 55 43 4b e7 03 00 00  |..`.....DUCK....|
00000010  00 00 00 00                                       |....|
00000014
```

Because the number after the magic bytes does not always name the storage version, let DuckDB read the headers: attach the file read-only and query `duckdb_databases()`, as in the following Python example.

```python
import duckdb

con = duckdb.connect()
con.execute("ATTACH 'file.db' AS file_db (READ_ONLY)")
print(con.execute(
    "SELECT tags['storage_version'] FROM duckdb_databases() WHERE database_name = 'file_db'"
).fetchone()[0])
```

```
v2.0.0+
```

Storage Version Table

For changes in each given release, check out the change log on GitHub. To see the commits that changed each storage version, see the commit log.

Storage version	DuckDB version(s)
69	v2.0.x
68	v1.5.x
67	v1.4.x
66	v1.3.x
65	v1.2.x
64	v0.9.x, v0.10.x, v1.0.0, v1.1.x
51	v0.8.x
43	v0.7.x
39	v0.6.x
38	v0.5.x
33	v0.3.3, v0.3.4, v0.4.0
31	v0.3.2
27	v0.3.1
25	v0.3.0
21	v0.2.9
18	v0.2.8
17	v0.2.7
15	v0.2.6
13	v0.2.5
11	v0.2.4
6	v0.2.3
4	v0.2.2
1	v0.2.1 and prior

Compression

DuckDB uses lightweight compression. By default, compression is only applied to persistent databases and is **not applied to in-memory instances**. To turn on compression for in-memory databases, use ATTACH with the COMPRESS option.

Note that available compression algorithms depend on the storage version used, so you might need to set an explicit storage version to get access to all compression algorithms.

Compression Algorithms

The compression algorithms supported by DuckDB include the following:

- Constant Encoding
- Run-Length Encoding (RLE)
- Bit Packing
- Frame of Reference (FOR)
- Dictionary Encoding
- Fast Static Symbol Table (FSST) – VLDB 2020 paper
- Adaptive Lossless Floating-Point Compression (ALP) – SIGMOD 2024 paper

- Chimp – VLDB 2022 paper
- Patas
- Zstd

Disk Usage

The disk usage of DuckDB's format depends on a number of factors, including the data type and the data distribution, the compression methods used, etc. As a rough approximation, loading 100 GB of uncompressed CSV files into a DuckDB database file will require 25 GB of disk space, while loading 100 GB of Parquet files will require 120 GB of disk space.

Row Groups

DuckDB's storage format stores the data in *row groups,* i.e., horizontal partitions of the data. This concept is equivalent to Parquet's row groups. Several features in DuckDB, including parallelism and compression are based on row groups.

The row group size can be specified as an option of the ATTACH statement:

```
ATTACH '/tmp/somefile.db' AS db (ROW_GROUP_SIZE 16384);
```

Troubleshooting

Error Message When Opening a DuckDB 2.0 Database File with an Older Version

DuckDB versions before 2.0 cannot read database files in the v2.0.0 storage format. Opening such a file with DuckDB 1.5.6 results in the following error:

```
IO Error: Trying to read a database file with version number 69, but we can only read versions between 64 and 68.
The database file was created with a newer version of DuckDB.
```

DuckDB 1.5.5 and earlier versions report version number 999 instead of 69.

To read the data with an older DuckDB version, either upgrade that process to DuckDB 2.0, or use DuckDB 2.0 to copy the database into a file with an older storage version, e.g., `v1.5.0`.

Error Message When Opening an Incompatible Database File

When opening a database file that has been written by a different DuckDB version from the one you are using, the following error message may occur:

```
Error: unable to open database "...": Serialization Error: Failed to deserialize: ...
```

The message implies that the database file was created with a newer DuckDB version and uses features that the DuckDB version used to read the file does not support.

There are two potential workarounds:

1. Update your DuckDB version to the latest stable version.
2. Open the database with the latest version of DuckDB, export it to a standard format (e.g., Parquet), then import it to any version of DuckDB. See the `EXPORT/IMPORT DATABASE` statements for details.

Execution Format

`Vector` is the container format used to store in-memory data during execution. `DataChunk` is a collection of Vectors, used for instance to represent a column list in a `PhysicalProjection` operator.

Data Flow

DuckDB uses a vectorized query execution model. All operators in DuckDB are optimized to work on Vectors of a fixed size.

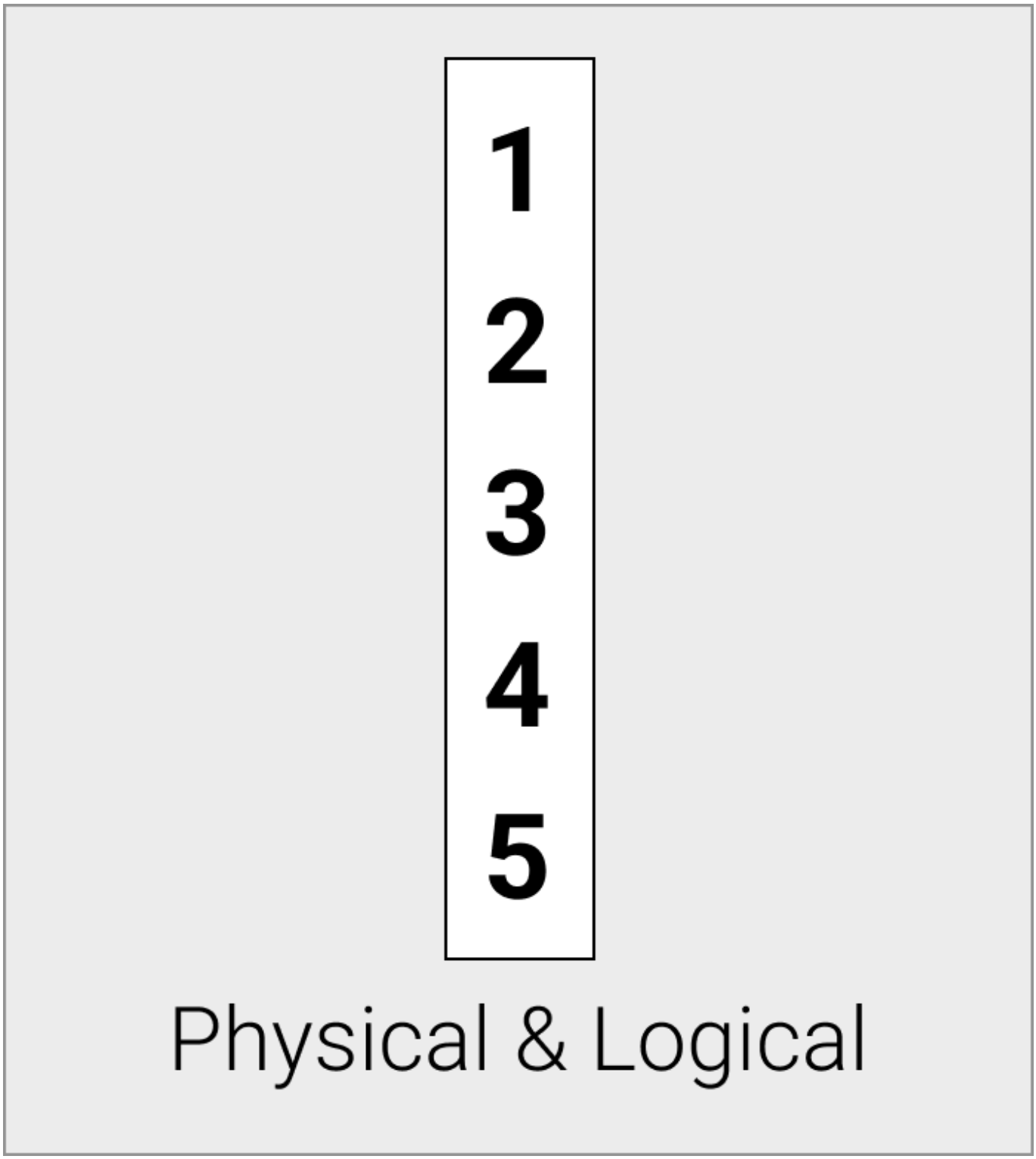
This fixed size is commonly referred to in the code as `STANDARD_VECTOR_SIZE`. The default `STANDARD_VECTOR_SIZE` is 2048 tuples.

Vector Format

Vectors logically represent arrays that contain data of a single type. DuckDB supports different *vector formats*, which allow the system to store the same logical data with a different *physical representation*. This allows for a more compressed representation, and potentially allows for compressed execution throughout the system. Below the list of supported vector formats is shown.

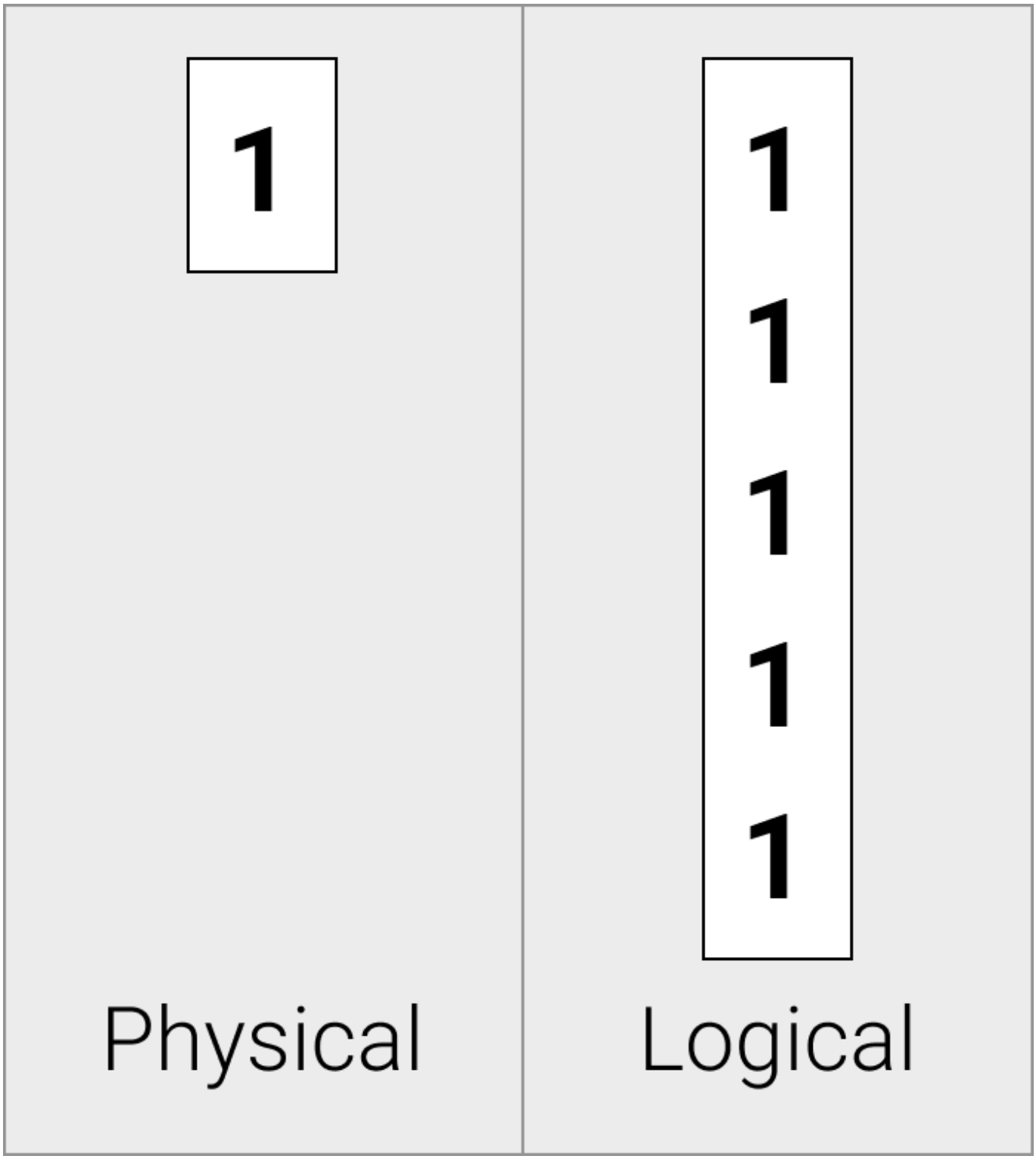
Flat Vectors

Flat vectors are physically stored as a contiguous array, this is the standard uncompressed vector format. For flat vectors the logical and physical representations are identical.



Constant Vectors

Constant vectors are physically stored as a single constant value.



Constant vectors are useful when data elements are repeated – for example, when representing the result of a constant expression in a function call, the constant vector allows us to only store the value once.

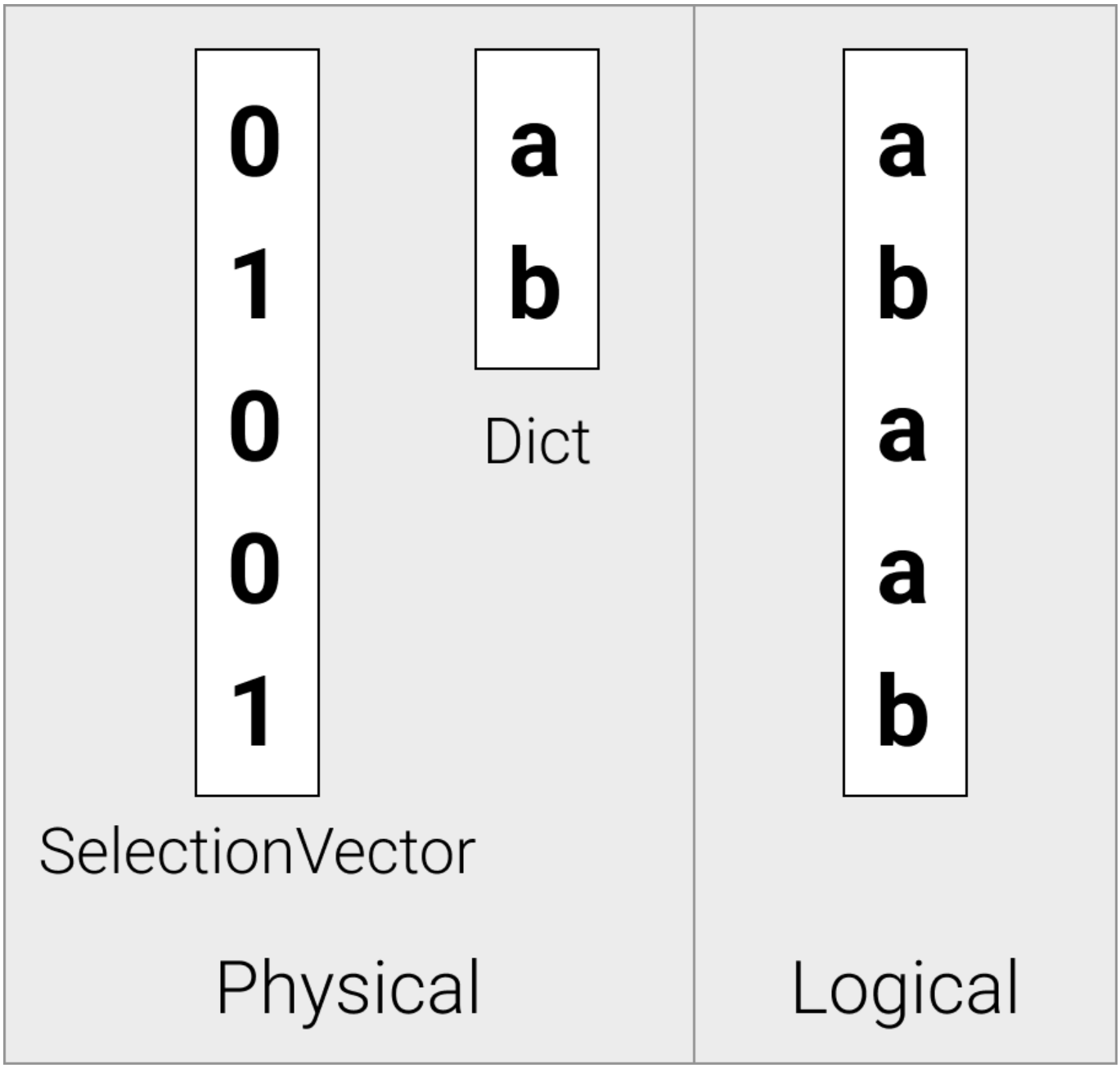
```
SELECT lst || 'duckdb'
FROM range(1000) tbl(lst);
```

Since `duckdb` is a string literal, the value of the literal is the same for every row. In a flat vector, we would have to duplicate the literal `duckdb` once for every row. The constant vector allows us to only store the literal once.

Constant vectors are also emitted by the storage when decompressing from constant compression.

Dictionary Vectors

Dictionary vectors are physically stored as a child vector, and a selection vector that contains indexes into the child vector.

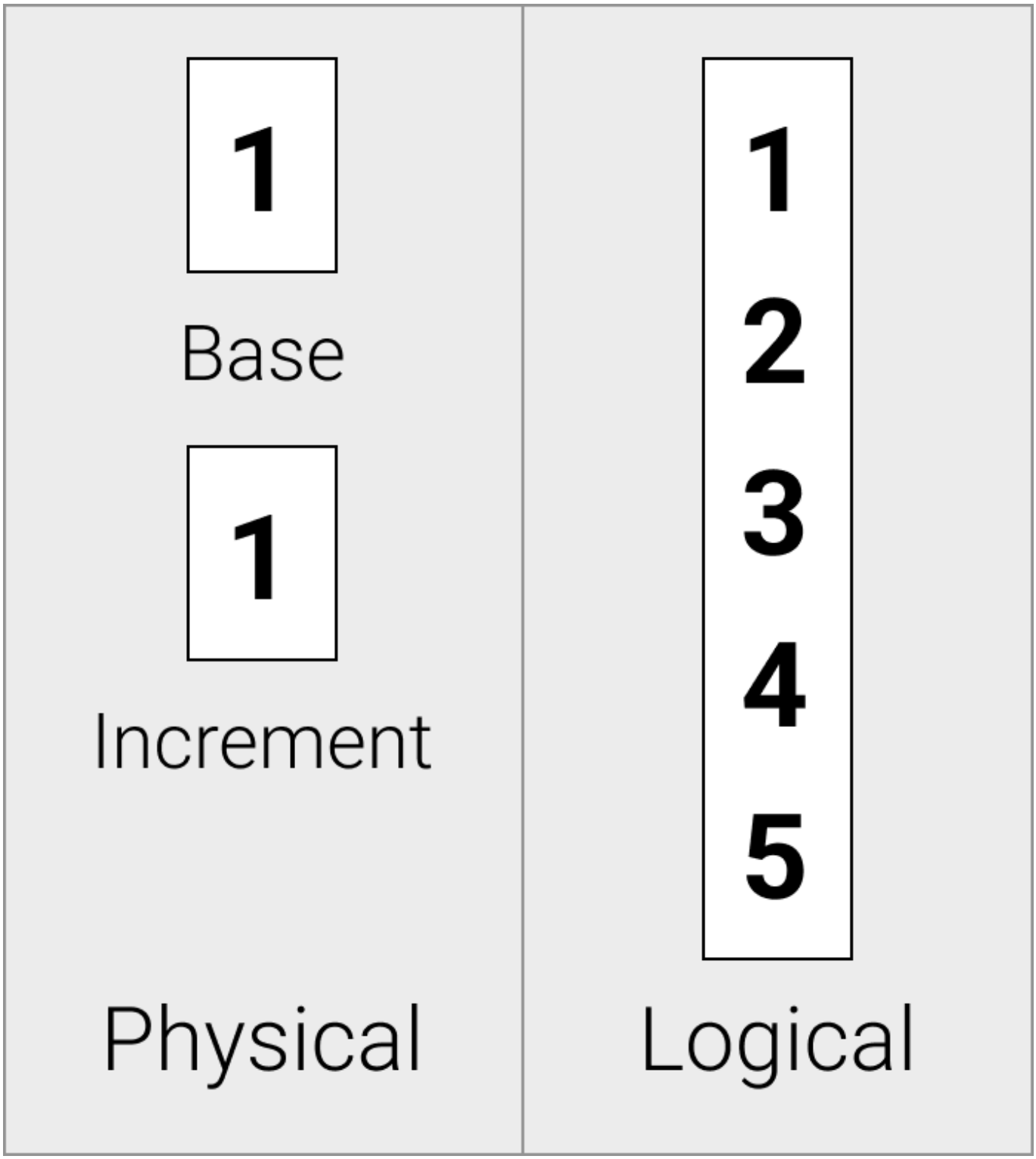


Dictionary vectors are emitted by the storage when decompressing from dictionary compression.

Just like constant vectors, dictionary vectors are also emitted by the storage. When deserializing a dictionary compressed column segment, we store this in a dictionary vector so we can keep the data compressed during query execution.

Sequence Vectors

Sequence vectors are physically stored as an offset and an increment value.



Sequence vectors are useful for efficiently storing incremental sequences. They are generally emitted for row identifiers.

Unified Vector Format

These properties of the different vector formats are great for optimization purposes, for example you can imagine the scenario where all the parameters to a function are constant, we can just compute the result once and emit a constant vector. But writing specialized code for every combination of vector types for every function is unfeasible due to the combinatorial explosion of possibilities.

Instead of doing this, whenever you want to generically use a vector regardless of the type, the UnifiedVectorFormat can be used. This format essentially acts as a generic view over the contents of the Vector. Every type of Vector can convert to this format.

Complex Types

String Vectors

To efficiently store strings, we make use of our `string_t` class.

```
struct string_t {
    union {
        struct {
            uint32_t length;
            char prefix[4];
            char *ptr;
        } pointer;
        struct {
            uint32_t length;
            char inlined[12];
        } inlined;
    } value;
};
```

Short strings (`<= 12 bytes`) are inlined into the structure, while larger strings are stored with a pointer to the data in the auxiliary string buffer. The length is used throughout the functions to avoid having to call `strlen` and having to continuously check for null-pointers. The prefix is used for comparisons as an early out (when the prefix does not match, we know the strings are not equal and don't need to chase any pointers).

List Vectors

List vectors are stored as a series of *list entries* together with a child Vector. The child vector contains the *values* that are present in the list, and the list entries specify how each individual list is constructed.

```
struct list_entry_t {
    idx_t offset;
    idx_t length;
};
```

The offset refers to the start row in the child Vector, the length keeps track of the size of the list of this row.

List vectors can be stored recursively. For nested list vectors, the child of a list vector is again a list vector.

For example, consider this mock representation of a Vector of type `BIGINT[][]`:

```
{
   "type": "list",
   "data": "list_entry_t",
   "child": {
      "type": "list",
      "data": "list_entry_t",
      "child": {
         "type": "bigint",
         "data": "int64_t"
      }
   }
}
```

Array Vectors

Array vectors store fixed-size lists. Unlike list vectors, they do not use per-row `list_entry_t` entries, because every row has the same, fixed number of elements. Instead, an array vector stores a single child Vector that holds the values of all rows laid out contiguously. The

elements of row `i` occupy the child positions in the range `[i * array_size, (i + 1) * array_size)`, where `array_size` is the fixed length of the array type.

Like list vectors, array vectors can be nested: the child of an array vector can itself be an array vector or a list vector.

For example, consider this mock representation of a Vector of type `INTEGER[3]`:

```
{
   "type": "array",
   "array_size": 3,
   "child": {
      "type": "integer",
      "data": "int32_t"
   }
}
```

Struct Vectors

Struct vectors store a list of child vectors. The number and types of the child vectors is defined by the schema of the struct.

Map Vectors

Internally map vectors are stored as a `LIST[STRUCT(key KEY_TYPE, value VALUE_TYPE)]`.

Union Vectors

Internally `UNION` utilizes the same structure as a `STRUCT`. The first "child" is always occupied by the Tag Vector of the `UNION`, which records for each row which of the `UNION`'s types apply to that row.

jemalloc

On supported platforms, DuckDB replaces the system's memory allocator with jemalloc. `jemalloc` is statically linked into DuckDB and cannot be installed or loaded during runtime.

> For technical reasons, we used to distribute `jemalloc` as an extension. Since DuckDB 1.5.3, it is distributed as a third-party library.

Operating System Support

The availability of `jemalloc` depends on the operating system.

Linux

64-bit Linux distributions of DuckDB ship with `jemalloc` by default (with the exception of builds for musl-based distributions). To disable `jemalloc`, build DuckDB from source and set the `BUILD_JEMALLOC` flag to `0` as follows:

```
GEN=ninja BUILD_JEMALLOC=0 make
```

This flag is passed to CMake as `-DENABLE_JEMALLOC=0`.

macOS and Windows

On macOS and Windows, `jemalloc` is not available. DuckDB uses the system's memory allocator on these platforms.

Configuration

Environment Variables

The jemalloc allocator in DuckDB can be configured via the `DUCKDB_JE_MALLOC_CONF` environment variable. Setting this is equivalent to setting the `MALLOC_CONF` environment variable for jemalloc but DuckDB uses a different environment variable name to avoid potential name clashes with other applications.

Background Threads

By default, jemalloc's background threads are disabled. To enable them, use the following configuration option:

```
SET allocator_background_threads = true;
```

Background threads asynchronously purge outstanding allocations so that this doesn't have to be done synchronously by the foreground threads. This improves allocation performance, and should be noticeable in allocation-heavy workloads, especially on many-core CPUs.

Pivot Internals

PIVOT

Pivoting is implemented as a combination of SQL query re-writing and a dedicated `PhysicalPivot` operator for higher performance. Each `PIVOT` is implemented as set of aggregations into lists and then the dedicated `PhysicalPivot` operator converts those lists into column names and values. Additional pre-processing steps are required if the columns to be created when pivoting are detected dynamically (which occurs when the `IN` clause is not in use).

DuckDB, like most SQL engines, requires that all column names and types be known at the start of a query. To automatically detect the columns that should be created as a result of a `PIVOT` statement, it must be translated into multiple queries. ENUM types are used to find the distinct values that should become columns. Each `ENUM` is then injected into one of the `PIVOT` statement's `IN` clauses.

After the `IN` clauses have been populated with `ENUM`s, the query is re-written again into a set of aggregations into lists.

For example:

```
PIVOT cities
ON year
USING sum(population);
```

is initially translated into:

```
CREATE TEMPORARY TYPE __pivot_enum_0_0 AS ENUM (
    SELECT DISTINCT
        year::VARCHAR
    FROM cities
    ORDER BY
        year
    );
PIVOT cities
ON year IN __pivot_enum_0_0
USING sum(population);
```

and finally translated into:

```
SELECT country, name, list(year), list(population_sum)
FROM (
    SELECT country, name, year, sum(population) AS population_sum
    FROM cities
    GROUP BY ALL
)
GROUP BY ALL;
```

This produces the result:

country	name	list(”year”)	list(population_sum)
NL	Amsterdam	[2000, 2010, 2020]	[1005, 1065, 1158]
US	Seattle	[2000, 2010, 2020]	[564, 608, 738]
US	New York City	[2000, 2010, 2020]	[8015, 8175, 8772]

The `PhysicalPivot` operator converts those lists into column names and values to return this result:

country	name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

> The rewrite into list-based aggregations shown above is used only when the number of distinct pivot values exceeds the `pivot_filter_threshold` configuration option (default `20`). Below that threshold, DuckDB instead uses filtered aggregates. The list-based approach materializes all values in memory, so you can lower or raise `pivot_filter_threshold` to trade off time against memory usage.

`UNPIVOT`

Internals

Unpivoting is implemented entirely as rewrites into SQL queries. Each `UNPIVOT` is implemented as set of `unnest` functions, operating on a list of the column names and a list of the column values. If dynamically unpivoting, the `COLUMNS` expression is evaluated first to calculate the column list.

For example:

```
UNPIVOT monthly_sales
ON jan, feb, mar, apr, may, jun
INTO
    NAME month
    VALUE sales;
```

is translated into:

```
SELECT
    empid,
    dept,
    unnest(['jan', 'feb', 'mar', 'apr', 'may', 'jun']) AS month,
    unnest(["jan", "feb", "mar", "apr", "may", "jun"]) AS sales
FROM monthly_sales;
```

Note the single quotes to build a list of text strings to populate `month`, and the double quotes to pull the column values for use in `sales`. This produces the same result as the initial example:

empid	dept	month	sales
1	electronics	jan	1
1	electronics	feb	2
1	electronics	mar	3
1	electronics	apr	4
1	electronics	may	5
1	electronics	jun	6
2	clothes	jan	10
2	clothes	feb	20
2	clothes	mar	30
2	clothes	apr	40

empid	dept	month	sales
2	clothes	may	50
2	clothes	jun	60
3	cars	jan	100
3	cars	feb	200
3	cars	mar	300
3	cars	apr	400
3	cars	may	500
3	cars	jun	600

Acknowledgments

This document is built with Pandoc using the Eisvogel template. The scripts to build the document are available in the DuckDB-Web repository.

The emojis used in this document are provided by Twemoji under the CC-BY 4.0 license.

The syntax highlighter uses the Bluloco Light theme by Umut Topuzoğlu.