DuckLake Documentation

DuckLake version 1.0
Generated on 2026-04-21 at 07:18 UTC

Contents

Contents i

Summary 1

Specification 3

Introduction 4

Data Types 5

Queries 12

Tables 21

Tables . . . 21
ducklake_column . . . 24
ducklake_column_mapping . . . 25
ducklake_column_tag . . . 25
ducklake_data_file . . . 26
ducklake_delete_file . . . 27
ducklake_file_column_stats . . . 27
ducklake_file_partition_value . . . 28
ducklake_file_variant_stats . . . 28
ducklake_files_scheduled_for_deletion . . . 29
ducklake_inlined_data_tables . . . 30
ducklake_macro . . . 30
ducklake_macro_impl . . . 31
ducklake_macro_parameters . . . 31
ducklake_metadata . . . 31
ducklake_name_mapping . . . 33
ducklake_partition_column . . . 34
ducklake_partition_info . . . 34
ducklake_schema . . . 35
ducklake_schema_versions . . . 35
ducklake_snapshot . . . 36
ducklake_snapshot_changes . . . 36
ducklake_sort_expression . . . 37
ducklake_sort_info . . . 37
ducklake_table . . . 38
ducklake_table_column_stats . . . 38
ducklake_table_stats . . . 39
ducklake_tag . . . 39
ducklake_view . . . 40

DuckDB Extension 41

Introduction 42

Usage 45
Connecting . . . 45
Choosing a Catalog Database . . . 47
Choosing Storage . . . 48
Snapshots . . . 49
Schema Evolution . . . 50
Time Travel . . . 52
Upserting . . . 52
Configuration . . . 55
Paths . . . 57

Maintenance 58
Recommended Maintenance . . . 58
Merge Adjacent Files . . . 58
Expire Snapshots . . . 60
Cleanup of Files . . . 61
Rewrite Heavily Deleted Files . . . 62
Checkpoint . . . 63

Advanced Features 65
Constraints . . . 65
Conflict Resolution . . . 65
Data Change Feed . . . 66
Data Inlining . . . 68
Encryption . . . 72
Partitioning . . . 73
Transactions . . . 73
Row Lineage . . . 74
Macros . . . 74
Views . . . 75
Comments . . . 75
Sorted Tables . . . 76
Logging . . . 77

Metadata 80
List Files . . . 80
Adding Files . . . 81

Migrations 83
DuckDB to DuckLake . . . 83

Guides 85
Guides . . . 85
Access Control . . . 85
Backup and Recovery . . . 89
Public DuckLake on Object Storage . . . 91
Using a Remote Data Path . . . 93
Troubleshooting . . . 95

Unsupported Features 96

Acknowledgments **98**

Summary

This document contains DuckLake's documentation 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:

- DuckLake documentation source code: github.com/duckdb/ducklake-web
- DuckDB `ducklake` extension: github.com/duckdb/ducklake

Specification

Introduction

This page contains the specification for the DuckLake format, version 1.0.

Building Blocks

DuckLake requires two main components:

- **Catalog database:** DuckLake requires a database that supports transactions and primary key constraints as defined by the SQL-92 standard.
- **Data storage:** The DuckLake specification requires a storage component for storing the data in Parquet format.

Catalog Database

DuckLake uses SQL tables and queries to define the catalog information (metadata, statistics, etc.). This specification explains the schema and semantics of these:

- Data Types
- Queries
- Tables

If you are reading this specification for the first time, we recommend starting with the "Queries" page, which introduces the queries used by DuckLake.

Data Storage

DuckLake uses Parquet files to represent its tables. These files can be stored in object storage (blob storage), block storage or file storage.

Data Types

DuckLake specifies multiple different data types for field values, and also supports nested types. The types of columns are defined in the `column_type` field of the `ducklake_column` table.

Primitive Types

Type	Description
`boolean`	True or false
`int8`	8-bit signed integer
`int16`	16-bit signed integer
`int32`	32-bit signed integer
`int64`	64-bit signed integer
`uint8`	8-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer
`int128`	128-bit signed integer
`uint128`	128-bit unsigned integer
`float32`	32-bit IEEE 754 floating-point value
`float64`	64-bit IEEE 754 floating-point value
`decimal(P, S)`	Fixed-point decimal with precision P and scale S
`time`	Time of day, microsecond precision
`timetz`	Time of day, microsecond precision, with time zone
`date`	Calendar date
`timestamp`	Timestamp, microsecond precision
`timestamptz`	Timestamp, microsecond precision, with time zone
`timestamp_s`	Timestamp, second precision
`timestamp_ms`	Timestamp, millisecond precision
`timestamp_ns`	Timestamp, nanosecond precision
`interval`	Time interval in three different granularities: months, days, and milliseconds
`varchar`	Text
`blob`	Binary data
`json`	JSON
`uuid`	Universally unique identifier

Nested Types

DuckLake supports nested types and primitive types. Nested types are defined recursively using the `parent_column` field in the `ducklake_column` table. For example, a column of type `INT[]` is stored as two rows: a parent column of type `list`, and a child column of type `int32` whose `parent_column` points to the parent.

The following nested types are supported:

Type	Description
`list`	Collection of values with a single child type
`struct`	A tuple of typed values
`map`	A collection of key-value pairs

Semi-Structured Types

Type	Description
`variant`	A dynamically typed value that can hold any primitive or nested type, stored in a binary encoding.

The `variant` type is similar to JSON but is more strongly typed internally, supports a wider range of types (e.g., `date`, `timestamp`, `decimal`), and is stored in a binary-encoded format rather than as a string. Variants are stored in Parquet files according to the Parquet variant encoding specification.

Variants can be **shredded** into their constituent primitive types when all rows share a consistent schema for a given sub-field. Shredded fields are stored and queried with the same efficiency as native primitive columns. Per-file statistics for shredded sub-fields are recorded in the `ducklake_file_variant_stats` table.

> **Note.** The `variant` type is natively supported in DuckDB. For catalog databases that do not have a native variant type (e.g., PostgreSQL, SQLite), variants cannot yet be stored as inline values in those catalogs.

Geometry Types

DuckLake supports geometry types using the `geometry` type of the Parquet format. The `geometry` type can store different types of spatial representations called geometry primitives, of which DuckLake supports the following:

Geometry primitive	Description
`point`	A single location in coordinate space.
`linestring`	A sequence of points connected by straight line segments.
`polygon`	A planar surface defined by one exterior boundary and zero or more interior boundaries (holes).
`multipoint`	A collection of `point` geometries.
`multilinestring`	A collection of `linestring` geometries.
`multipolygon`	A collection of `polygon` geometries.
`linestring_z`	A `linestring` geometry with an additional Z (elevation) coordinate for each point.
`geometrycollection`	A heterogeneous collection of geometry primitives (e.g., points, lines, polygons, etc.).

Type Encoding for Statistics

Statistics values are string-encoded, as they must be stored as `VARCHAR`, since that allows storing types that are not native to the catalog database system (e.g., PostgreSQL does not support `VARIANT` natively). Statistics are stored in both `ducklake_file_column_stats` and `ducklake_table_column_stats`. Most types follow a straightforward encoding, however some do not. The following table describes the encoding of each type:

Type	Description	Example
`boolean`	`0` or `1`	`1`
`int8`	Integer string	`42`
`int16`	Integer string	`1000`
`int32`	Integer string	`100000`
`int64`	Integer string	`100000000`
`uint8`	Integer string	`255`
`uint16`	Integer string	`65535`
`uint32`	Integer string	`4294967295`
`uint64`	Integer string	`18446744073709551615`
`float32`	Numeric string with special values: `inf`, `-inf`. NaN values are excluded from min/max, the `contains_nan` flag indicates their presence	`3.14`
`float64`	Numeric string with special values: `inf`, `-inf`. NaN values are excluded from min/max, the `contains_nan` flag indicates their presence	`3.14159265358979`
`decimal`	Numeric string (independent of precision and scale)	`12345.67`
`date`	ISO 8601 date with special values: `infinity`, `-infinity`	`2024-01-15`
`time`	ISO 8601 time with microseconds if non-zero	`12:30:00.123456`
`timestamp`	ISO 8601 timestamp with microseconds if non-zero, with special values: `infinity`, `-infinity`	`2024-01-15 12:30:00.123456`
`timestamptz`	ISO 8601 timestamp with UTC offset, with special values: `infinity`, `-infinity`	`2024-01-15 12:30:00.123456+00`
`timestamp_s`	ISO 8601 timestamp (second precision) with special values: `infinity`, `-infinity`	`2024-01-15 12:30:00`
`timestamp_ms`	ISO 8601 timestamp (millisecond precision) with special values: `infinity`, `-infinity`	`2024-01-15 12:30:00.123`
`timestamp_ns`	ISO 8601 timestamp (nanosecond precision) with special values: `infinity`, `-infinity`	`2024-01-15 12:30:00.123456789`
`varchar`	As-is	`hello`
`json`	As-is	`{"key": "value"}`
`blob`	Hex-encoded string of the raw bytes	`68656C6C6F20776F726C64`
`uuid`	Standard UUID string	`550e8400-e29b-41d4-a716-446655440000`

The following types do not currently have min/max statistics, as they are not supported by the underlying Parquet format:

1. `int128`
2. `uint128`
3. `timetz`

4. `interval`

Nested Types

Nested types (`list`, `struct`, `map`) do not have min/max statistics themselves, statistics are collected recursively for their child columns instead. For example:

Given the following table:

```
CREATE TABLE nested_types (
    col_list INT[],
    col_struct STRUCT(a INT, b VARCHAR),
    col_map MAP(VARCHAR, INT)
);
INSERT INTO nested_types VALUES
    ([1, 2, 3], {'a': 10, 'b': 'hello'}, MAP {'x': 1}),
    ([4, 5, 6], {'a': 20, 'b': 'world'}, MAP {'y': 2});
```

The parent columns (`col_list`, `col_struct`, `col_map`) have no min/max statistics. Instead, the child columns store:

Child column	Type	Min	Max
`element`	`int32`	`1`	`6`
`a`	`int32`	`10`	`20`
`b`	`varchar`	`hello`	`world`
`key`	`varchar`	`x`	`y`
`value`	`int32`	`1`	`2`

Extra Statistics

The `geometry` and `variant` types do not use string-encoded min/max values. Instead, their statistics are stored as JSON in the `extra_stats` column of the statistics tables.

Geometry Geometry statistics contain a bounding box (`bbox`) and a list of geometry primitive types present in the column (`types`). For example:

```
{
    "bbox": {
        "xmin": 0.000000,
        "xmax": 5.000000,
        "ymin": 0.000000,
        "ymax": 5.000000,
        "zmin": null,
        "zmax": null,
        "mmin": null,
        "mmax": null
    },
    "types": ["point"]
}
```

Variant Variant statistics contain a JSON array with one entry per shredded sub-field. Each entry records the field path, its shredded type, and min/max statistics for that sub-field. If the variant cannot be shredded (i.e., has inconsistent types across rows), `extra_stats` is `NULL`. An example of a Variant `extra_stats`:

```
[
    {
        "field_name": "root",
        "shredded_type": "int32",
        "null_count": 0,
        "min": "42",
        "max": "99",
        "num_values": 3,
        "column_size_bytes": 37,
        "any_valid": true
    }
]
```

Type Encoding for Data Inlining

When storing columns for data inlining, a table is created in the catalog to store small inserts and updates in the database. Since DuckDB supports all DuckDB types, inlining types in a DuckDB catalog is straightforward. However, other catalogs might require encodings for non-native types.

PostgreSQL

PostgreSQL does not natively support all DuckDB types. The following table describes how each DuckDB type is mapped to a PostgreSQL type for inlined data storage. Types not listed below (e.g., `BOOLEAN`, `SMALLINT`, `INTEGER`, `BIGINT`, `FLOAT`, `DOUBLE`, `DECIMAL`, `TIME`, `TIMETZ`, `INTERVAL`, `UUID`) are stored using their native PostgreSQL equivalents.

DuckDB Type	PostgreSQL Type	Encoding	Example
`TINYINT`	`SMALLINT`	Direct cast	`42` -> `42`
`UTINYINT`	`INTEGER`	Direct cast	`200` -> `200`
`USMALLINT`	`INTEGER`	Direct cast	`50000` -> `50000`
`UINTEGER`	`BIGINT`	Direct cast	`3000000000` -> `3000000000`
`UBIGINT`	`VARCHAR`	Stored as text	`18446744073709551615` -> `'18446744073709551615'`
`HUGEINT`	`VARCHAR`	Stored as text	`-170141183460469231731687303715884` -> `'-170141183460469231731687303715884`
`UHUGEINT`	`VARCHAR`	Stored as text	`340282366920938463463374607431768` -> `'340282366920938463463374607431768`
`VARCHAR`	`BYTEA`	Text reinterpreted as bytes. PostgreSQL cannot store null bytes in `VARCHAR`/`TEXT`	`'hello world'` -> `\x68656c6c6f20776f726c64`
`BLOB`	`BYTEA`	Binary data stored directly as `BYTEA`	`'\x00\xFF'::BLOB` -> `\x00ff`
`JSON`	`BYTEA`	Text reinterpreted as bytes. DuckDB's `JSON` type is internally `VARCHAR`, so it follows the same as `VARCHAR` to `BYTEA`	`'{"key": "value"}'` -> `\x7b226b6579223a202276616c756522`
`DATE`	`VARCHAR`	Stored as text	`2024-01-15` -> `'2024-01-15'`

DuckDB Type	PostgreSQL Type	Encoding	Example
`TIMESTAMP`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00.123456` -> `'2024-01-15 12:30:00.123456'`
`TIMESTAMPTZ`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00.123456+00` -> `'2024-01-15 12:30:00.123456+00'`
`TIMESTAMP_S`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00` -> `'2024-01-15 12:30:00'`
`TIMESTAMP_MS`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00.123` -> `'2024-01-15 12:30:00.123'`
`TIMESTAMP_NS`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00.123456789` -> `'2024-01-15 12:30:00.123456789'`
Nested types	`VARCHAR`	Stored as text, similar to stats	`[1, 2, 3]` -> `'[1, 2, 3]'`

SQLite

SQLite has a flexible type system with only five storage classes: `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL`. DuckDB's SQLite scanner maps all integer types to `BIGINT` (i.e., SQLite's `INTEGER` type) and most other types to `VARCHAR` (SQLite's `TEXT` type). Only `BLOB` is stored as `BLOB`. The following table describes the non-trivial type mappings:

DuckDB Type	SQLite Type	Encoding	Example
`BOOLEAN`	`BIGINT`	Stored as integer (`0`/`1`)	`true` -> `1`
`TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`	`BIGINT`	All integer types use SQLite's `INTEGER` affinity	`42` -> `42`
`UTINYINT`, `USMALLINT`, `UINTEGER`	`BIGINT`	Unsigned integers fit within `BIGINT` range	`200` -> `200`
`UBIGINT`	`VARCHAR`	Stored as text	`18446744073709551615` -> `'18446744073709551615'`
`HUGEINT`	`VARCHAR`	Stored as text	`-170141183460469231731687303715884` -> `'-170141183460469231731687303715884`
`UHUGEINT`	`VARCHAR`	Stored as text	`340282366920938463463374607431768` -> `'340282366920938463463374607431768`
`FLOAT`	`VARCHAR`	Stored as text	`3.14` -> `'3.14'`
`DOUBLE`	`VARCHAR`	Stored as text	`3.14159265358979` -> `'3.14159265358979'`
`DECIMAL`	`VARCHAR`	Stored as text	`12345.67` -> `'12345.67'`

DuckDB Type	SQLite Type	Encoding	Example
`UUID`	`VARCHAR`	Stored as text	`550e8400-...` -> `'550e8400-...'`
`DATE`, `TIME`, `TIMETZ`	`VARCHAR`	Stored as text	`2024-01-15` -> `'2024-01-15'`
`TIMESTAMP`, `TIMESTAMPTZ`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00.123456` -> `'2024-01-15 12:30:00.123456'`
`TIMESTAMP_S`, `TIMESTAMP_MS`, `TIMESTAMP_NS`	`VARCHAR`	Stored as text	`2024-01-15 12:30:00.123` -> `'2024-01-15 12:30:00.123'`
`INTERVAL`	`VARCHAR`	Stored as text	`1 year 2 months 3 days` -> `'1 year 2 months 3 days'`
Nested types	`VARCHAR`	Stored as text, similar to stats	`[1, 2, 3]` -> `'[1, 2, 3]'`

Queries

This page explains the queries issued to the DuckLake catalog database for reading and writing data.

Reading Data

DuckLake specifies *tables* and *update transactions* to modify them. DuckLake is not a black box: all metadata is stored as SQL tables under the user's control. Of course, they can be queried in whichever way is best for a client. Below we describe a small working example to retrieve table data.

> The information below is to provide transparency to users and to aid developers making their own implementation of DuckLake. If you are using the `ducklake` DuckDB extension, you do not need to worry about these: the extension is running these operations in the background for you.

Get Current Snapshot

Before anything else we need to find a snapshot id to be queried. There can be many snapshots in the `ducklake_snapshot` table. A snapshot id is a continuously increasing number that identifies a snapshot. In most cases, you would query the most recent one like so:

```
SELECT snapshot_id
FROM ducklake_snapshot
WHERE snapshot_id =
    (SELECT max(snapshot_id) FROM ducklake_snapshot);
```

List Schemas

A DuckLake catalog can contain many SQL-style schemas, which each can contain many tables. These are listed in the `ducklake_schema` table. Here's how we get the list of valid schemas for a given snapshot:

```
SELECT schema_id, schema_name
FROM ducklake_schema
WHERE
    <snapshot_id> >= begin_snapshot AND
    (<snapshot_id> < end_snapshot OR end_snapshot IS NULL);
```

where

- `<snapshot_id>` is a `BIGINT` referring to the `snapshot_id` column in the `ducklake_snapshot` table.

List Tables

We can list the tables available in a schema for a specific snapshot using the `ducklake_table` table:

```
SELECT table_id, table_name
FROM ducklake_table
WHERE
    schema_id = <schema_id> AND
    <snapshot_id> >= begin_snapshot AND
    (<snapshot_id> < end_snapshot OR end_snapshot IS NULL);
```

where

- <schema_id> is a BIGINT referring to the schema_id column in the ducklake_schema table.
- <snapshot_id> is a BIGINT referring to the snapshot_id column in the ducklake_snapshot table.

Show the Structure of a Table

For each given table, we can list the available top-level columns using the ducklake_column table:

```
SELECT column_id, column_name, column_type
FROM ducklake_column
WHERE
    table_id = <table_id> AND
    parent_column IS NULL AND
    <snapshot_id> >= begin_snapshot AND
    (<snapshot_id> < end_snapshot OR end_snapshot IS NULL)
ORDER BY column_order;
```

where

- <table_id> is a BIGINT referring to the table_id column in the ducklake_table table.
- <snapshot_id> is a BIGINT referring to the snapshot_id column in the ducklake_snapshot table.

> DuckLake supports nested columns – the filter for `parent_column IS NULL` only shows the top-level columns.

For the list of supported data types, please refer to the "Data Types" page.

SELECT

Now that we know the table structure we can query actual data from the Parquet files that store table data. We need to join the list of data files with the list of delete files (if any). There can be at most one delete file per file in a single snapshot.

```
SELECT data.path AS data_file_path, del.path AS delete_file_path
FROM ducklake_data_file AS data
LEFT JOIN (
    SELECT *
    FROM ducklake_delete_file
    WHERE
        <snapshot_id> >= begin_snapshot AND
        (<snapshot_id> < end_snapshot OR end_snapshot IS NULL)
    ) AS del
USING (data_file_id)
WHERE
    data.table_id = <table_id> AND
    <snapshot_id> >= data.begin_snapshot AND
    (<snapshot_id> < data.end_snapshot OR data.end_snapshot IS NULL)
ORDER BY file_order;
```

where

- <table_id> is a BIGINT referring to the table_id column in the ducklake_table table.
- <snapshot_id> is a BIGINT referring to the snapshot_id column in the ducklake_snapshot table.

Now we have a list of files. In order to reconstruct actual table rows, we need to read all rows from the `data_file_path` files and remove the rows labeled as deleted in the `delete_file_path`.

Not all files have to contain all the columns currently defined in the table, some files may also have columns that existed previously but have been removed.

> DuckLake also supports changing the schema, see schema evolution.

Note on Paths In DuckLake, paths can be relative to the initially specified data path. Whether a path is relative or not to the `data_path` prefix from `ducklake_metadata`, is stored in the `ducklake_data_file` and `ducklake_delete_file` entries (`path_is_relative`).

SELECT with File Pruning

One of the main strengths of lakehouse formats is the ability to *prune* files that cannot contain data relevant to the query. The `ducklake_file_column_stats` table contains the file-level statistics. We can use the information there to prune the list of files to be read if a filter predicate is given.

We can get a list of all files that are part of a given table like described above. We can then reduce that list to only relevant files by querying the per-file column statistics. For example, for scalar equality we can find the relevant files using the query below:

```
SELECT data_file_id
FROM ducklake_file_column_stats
WHERE
    table_id = <table_id> AND
    column_id = <column_id> AND
    (<scalar> >= min_value OR min_value IS NULL) AND
    (<scalar> <= max_value OR max_value IS NULL);
```

where

- `<table_id>` is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- `<column_id>` is a `BIGINT` referring to the `column_id` column in the `ducklake_column` table.
- `<scalar>` is the scalar comparison value for the pruning.

Of course, other filter predicates like “greater than” will require slightly different filtering here.

> The minimum and maximum values for each column are stored as strings and need to be cast for correct range filters on numeric columns.

Writing Data

Snapshot Creation

Any changes to data stored in DuckLake require the creation of a new snapshot. We need to:

- create a new snapshot in `ducklake_snapshot` and
- log the changes a snapshot made in `ducklake_snapshot_changes`

```
INSERT INTO ducklake_snapshot (
    snapshot_id,
    snapshot_timestamp,
    schema_version,
    next_catalog_id,
    next_file_id
)
VALUES (
    <snapshot_id>,
    now(),
    <schema_version>,
    <next_catalog_id>,
    <next_file_id>
);
```

```
INSERT INTO ducklake_snapshot_changes (
    snapshot_id,
    snapshot_changes,
    author,
    commit_message,
    commit_extra_info
)
VALUES (
    <snapshot_id>,
    <changes>,
    <author>,
    <commit_message>,
    <commit_extra_info>
);
```

where

- `<snapshot_id>` is the new snapshot identifier. This should be `max(snapshot_id) + 1`.
- `<schema_version>` is the schema version for the new snapshot. If any schema changes are made, this needs to be incremented. Otherwise the previous snapshot's `schema_version` can be re-used.
- `<next_catalog_id>` gives the next unused identifier for tables, schemas, or views. This only has to be incremented if new catalog entries are created.
- `<next_file_id>` is the same but for data or delete files.
- `<changes>` contains a list of changes performed by the snapshot. See the list of possible values in the `ducklake_snapshot_changes` table's documentation.
- `<author>` contains information about the author of the commit (optional).
- `<commit_message>` attaches a commit message to the transaction (optional).
- `<commit_extra_info>` attaches extra information to the transaction (optional).

CREATE SCHEMA

A schema is a collection of tables. In order to create a new schema, we can just insert into the `ducklake_schema` table:

```
INSERT INTO ducklake_schema (
    schema_id,
    schema_uuid,
    begin_snapshot,
    end_snapshot,
    schema_name
)
VALUES (
    <schema_id>,
    uuid(),
    <snapshot_id>,
    NULL,
    <schema_name>
);
```

where

- `<schema_id>` is the new schema identifier. This should be created by incrementing `next_catalog_id` from the previous snapshot.
- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<schema_name>` is the name of the new schema.

CREATE TABLE

Creating a table in a schema is very similar to creating a schema. We insert into the `ducklake_table` table:

```
INSERT INTO ducklake_table (
    table_id,
    table_uuid,
    begin_snapshot,
    end_snapshot,
    schema_id,
    table_name
)
VALUES (
    <table_id>,
    uuid(),
    <snapshot_id>,
    NULL,
    <schema_id>,
    <table_name>
);
```

where

- `<table_id>` is the new table identifier. This should be created by further incrementing `next_catalog_id` from the previous snapshot.
- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<schema_id>` is a `BIGINT` referring to the `schema_id` column in the `ducklake_schema` table table.
- `<table_name>` is the name of the new table.

A table needs some columns, we can add columns to the new table by inserting into the `ducklake_column` table table. For each column to be added, we run the following query:

```
INSERT INTO ducklake_column (
    column_id,
    begin_snapshot,
    end_snapshot,
    table_id,
    column_order,
    column_name,
    column_type,
    nulls_allowed
)
VALUES (
    <column_id>,
    <snapshot_id>,
    NULL,
    <table_id>,
    <column_order>,
    <column_name>,
    <column_type>,
    <nulls_allowed>
);
```

where

- `<column_id>` is the new column identifier. This ID must be unique *within the table* over its entire life time.
- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<table_id>` is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- `<column_order>` is a number that defines where the column is placed in an ordered list of columns.
- `<column_name>` is the name of the column.

- `<column_type>` is the data type of the column. See the "Data Types" page for details.
- `<nulls_allowed>` is a boolean that defines if `NULL` values can be stored in the column. Typically set to `true`.

> We skipped some complexity in this example around default values and nested types and just left those fields as `NULL`. See the table schema definition for additional details.

DROP TABLE

Dropping a table in DuckLake requires an update to the `end_snapshot` field in all metadata entries corresponding to the dropped table ID.

```
UPDATE ducklake_table
SET end_snapshot = <snapshot_id> WHERE table_id = <table_id> AND end_snapshot IS NULL;

UPDATE ducklake_partition_info
SET end_snapshot = <snapshot_id> WHERE table_id = <table_id> AND end_snapshot IS NULL;

UPDATE ducklake_column
SET end_snapshot = <snapshot_id> WHERE table_id = <table_id> AND end_snapshot IS NULL;

UPDATE ducklake_column_tag
SET end_snapshot = <snapshot_id> WHERE table_id = <table_id> AND end_snapshot IS NULL;

UPDATE ducklake_data_file
SET end_snapshot = <snapshot_id> WHERE table_id = <table_id> AND end_snapshot IS NULL;

UPDATE ducklake_delete_file
SET end_snapshot = <snapshot_id> WHERE table_id = <table_id> AND end_snapshot IS NULL;

UPDATE ducklake_tag
SET end_snapshot = <snapshot_id> WHERE object_id  = <table_id> AND end_snapshot IS NULL;
```

where

- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<table_id>` is the identifier of the table that will be dropped.

DROP SCHEMA

Dropping a schema in DuckLake requires updating the `end_snapshot` in the `ducklake_schema` table.

```
UPDATE ducklake_schema
SET
    end_snapshot = <snapshot_id>
WHERE
    schema_id = <schema_id> AND
    end_snapshot IS NULL;
```

where

- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<schema_id>` is the identifier of the schema that will be dropped.

> `DROP SCHEMA` is only allowed on empty schemas. Ensure that all tables within the schema are dropped beforehand.

INSERT

Inserting data into a DuckLake table consists of two main steps: first, we need to write a Parquet file containing the actual row data to storage, and second, we need to register that file in the metadata tables and update global statistics. Let's assume the file has already been written.

```
INSERT INTO ducklake_data_file (
    data_file_id,
    table_id,
    begin_snapshot,
    end_snapshot,
    path,
    path_is_relative,
    file_format,
    record_count,
    file_size_bytes,
    footer_size,
    row_id_start
)
VALUES (
    <data_file_id>,
    <table_id>,
    <snapshot_id>,
    NULL,
    <path>,
    true,
    'parquet',
    <record_count>,
    <file_size_bytes>,
    <footer_size>,
    <row_id_start>
);
```

where

- `<data_file_id>` is the new data file identifier. This ID must be unique *within the table* over its entire life time.
- `<table_id>` is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<path>` is the file name relative to the DuckLake data path from the top-level metadata.
- `<record_count>` is the number of rows in the file.
- `<file_size_bytes>` is the file size.
- `<footer_size>` is the position of the Parquet footer. This helps with efficiently reading the file.
- `<row_id_start>` is the first logical row ID from the file. This number can be read from the `ducklake_table_stats` table via column `next_row_id`.

We have omitted some complexity around relative paths, encrypted files, partitioning and partial files in this example. Refer to the `ducklake_data_file` table documentation for details.

> DuckLake also supports changing the schema, see schema evolution.

We also have to update some statistics in the `ducklake_table_stats` table and `ducklake_table_column_stats` table tables.

```
UPDATE ducklake_table_stats
SET
    record_count = record_count + <record_count>,
    next_row_id = next_row_id + <record_count>,
    file_size_bytes = file_size_bytes + <file_size_bytes>
WHERE table_id = <table_id>;

UPDATE ducklake_table_column_stats
```

```
SET
    contains_null = contains_null OR <null_count> > 0,
    contains_nan = contains_nan,
    min_value = min(min_value, <min_value>),
    max_value = max(max_value, <max_value>)
WHERE
    table_id = <table_id> AND
    column_id = <column_id>;

INSERT INTO ducklake_file_column_stats (
    data_file_id,
    table_id,
    column_id,
    value_count,
    null_count,
    min_value,
    max_value,
    contains_nan
)
VALUES (
    <data_file_id>,
    <table_id>,
    <column_id>,
    <record_count>,
    <null_count>,
    <min_value>,
    <max_value>,
    <contains_nan>;
);
```

where

- `<table_id>` is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- `<column_id>` is a `BIGINT` referring to the `column_id` column in the `ducklake_column` table.
- `<data_file_id>` is a `BIGINT` referring to the `data_file_id` column in the `ducklake_data_file` table.
- `<record_count>` is the number of values (including `NULL` and `NaN` values) in the file column.
- `<null_count>` is the number of `NULL` values in the file column.
- `<min_value>` is the *minimum* value in the file column as a string.
- `<max_value>` is the *maximum* value in the file column as a string.
- `<file_size_bytes>` is the size of the new Parquet file.
- `<contains_nan>` is a flag whether the column contains any `NaN` values. This is only relevant for floating-point types.

> This example assumes there are already rows in the table. If there are none, we need to use `INSERT` instead of `UPDATE` here. We also skipped the `column_size_bytes` column here, it can safely be set to `NULL`.

DELETE

Deleting data from a DuckLake table consists of two main steps: first, we need to write a *Parquet delete file* containing the row index to be deleted to storage, and second, we need to register that delete file in the metadata tables. Let's assume the file has already been written.

```
INSERT INTO ducklake_delete_file (
    delete_file_id,
    table_id,
    begin_snapshot,
    end_snapshot,
    data_file_id,
    path,
    path_is_relative,
```

```
    format,
    delete_count,
    file_size_bytes,
    footer_size
)
VALUES (
    <delete_file_id>,
    <table_id>,
    <snapshot_id>,
    NULL,
    <data_file_id>,
    <path>,
    true,
    'parquet',
    <delete_count>,
    <file_size_bytes>,
    <footer_size>
);
```

where

- `<delete_file_id>` is the identifier for the new delete file.
- `<table_id>` is a `BIGINT` referring to the `table_id` column in the `ducklake_table` table.
- `<snapshot_id>` is the snapshot identifier of the new snapshot as described above.
- `<data_file_id>` is the identifier of the data file from which the rows are to be deleted.
- `<path>` is the file name relative to the DuckLake data path from the top-level metadata.
- `<delete_count>` is the number of deletion records in the file.
- `<file_size_bytes>` is the file size.
- `<footer_size>` is the position of the Parquet footer. This helps with efficiently reading the file.

Notes:

- We have omitted some complexity around relative paths and encrypted files in this example. Refer to the `ducklake_delete_file` table documentation for details.
- In DuckLake, the strategy used for `DELETE` operations is **merge-on-read**. Delete files are referenced in the `ducklake_delete_file` table.
- Please note that `DELETE` operations also do not require updates to table statistics, as the statistics are maintained as upper bounds, and deletions do not violate these bounds.

UPDATE

In DuckLake, `UPDATE` operations are expressed as a combination of a `DELETE` followed by an `INSERT`. Specifically, the outdated row is marked for deletion, and the updated version of that row is inserted. As a result, the changes to the metadata tables are equivalent to performing a `DELETE` and an `INSERT` operation sequentially within the same transaction.

Tables

Tables

DuckLake v1.0 uses 28 tables to store metadata and to stage data fragments for data inlining. Below we describe all those tables and their semantics.

The following figure shows the most important 11 tables defined by the DuckLake schema:

Snapshots

- `ducklake_snapshot`
- `ducklake_snapshot_changes`

DuckLake Schema

- `ducklake_schema`
- `ducklake_table`
- `ducklake_view`

- `ducklake_column`

Macros

- `ducklake_macro`
- `ducklake_macro_impl`
- `ducklake_macro_parameters`

Data Files and Tables

- `ducklake_data_file`
- `ducklake_delete_file`
- `ducklake_files_scheduled_for_deletion`
- `ducklake_inlined_data_tables`

Data File Mapping

- `ducklake_column_mapping`
- `ducklake_name_mapping`

Statistics

DuckLake supports statistics on the table, column and file level.

- `ducklake_table_stats`
- `ducklake_table_column_stats`
- `ducklake_file_column_stats`
- `ducklake_file_variant_stats`

Partitioning Information

DuckLake supports defining explicit partitioning.

- `ducklake_partition_info`
- `ducklake_partition_column`
- `ducklake_file_partition_value`

Sort Information

DuckLake supports defining a sort order for tables to improve query performance.

- `ducklake_sort_info`
- `ducklake_sort_expression`

Auxiliary Tables

- `ducklake_metadata`
- `ducklake_tag`
- `ducklake_column_tag`
- `ducklake_schema_versions`

Full Schema Creation Script

For reference, you can see the full SQL script to create a DuckLake metadata database:

ducklake_column

This table describes the columns that are part of a table, including their types, default values, etc.

Column name	Column type
`column_id`	`BIGINT`
`begin_snapshot`	`BIGINT`
`end_snapshot`	`BIGINT`
`table_id`	`BIGINT`
`column_order`	`BIGINT`
`column_name`	`VARCHAR`
`column_type`	`VARCHAR`
`initial_default`	`VARCHAR`
`default_value`	`VARCHAR`
`nulls_allowed`	`BOOLEAN`
`parent_column`	`BIGINT`
`default_value_type`	`VARCHAR`
`default_value_dialect`	`VARCHAR`

- `column_id` is the numeric identifier of the column. If the Parquet file includes a field identifier, it corresponds to the file's `field_id`. This identifier should remain consistent throughout all versions of the column, until it's dropped. The `column_id` must be unique per table.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. This version of the column exists *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. This version of the column exists *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, this version of the column is currently valid.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_order` is a number that defines the position of the column in the list of columns. It needs to be unique within a snapshot but does not have to be contiguous (gaps are ok).
- `column_name` is the name of this version of the column, e.g., `my_column`.
- `column_type` is the type of this version of the column as defined in the list of data types.
- `initial_default` is the *initial* default value as the column is being created, e.g., in `ALTER TABLE`, encoded as a string. Can be `NULL`.
- `default_value` is the *operational* default value as data is being inserted and updated, e.g., in `INSERT`, encoded as a string. Can be `NULL`.
- `nulls_allowed` defines whether `NULL` values are allowed in this version of the column. Note that default values have to be set if this is set to `false`.
- `parent_column` is the `column_id` of the parent column. This is `NULL` for top-level and non-nested columns. For example, for `STRUCT` types, this would refer to the "parent" `STRUCT` column.
- `default_value_type` defines the default value type. It can either be `literal` (e.g., 42) or `expression` (e.g., `random()`)
- `default_value_dialect` defines the dialect used to interpret default values, especially useful for expressions. The dialect is the name of the system that created that value (e.g., duckdb).

> Every ALTER of the column creates a new version of the column, which will use the same `column_id`.

ducklake_column_mapping

Mappings contain the information used to map Parquet fields to column ids in the absence of `field-ids` in the Parquet file.

Column name	Column type
`mapping_id`	`BIGINT`
`table_id`	`BIGINT`
`type`	`VARCHAR`

- `mapping_id` is the numeric identifier of the mapping. `mapping_id` is incremented from `next_catalog_id` in the `ducklake_snapshot` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `type` defines what method is used to perform the mapping.

The valid `type` values are the following:

`type`	Description
`map_by_name`	Map the columns based on the names in the Parquet file

ducklake_column_tag

Columns can also have tags, those are defined in this table.

Column name	Column type
`table_id`	`BIGINT`
`column_id`	`BIGINT`
`begin_snapshot`	`BIGINT`
`end_snapshot`	`BIGINT`
`key`	`VARCHAR`
`value`	`VARCHAR`

- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The tag is valid *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The tag is valid *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the tag is currently valid.
- `key` is an arbitrary key string. The key can't be `NULL`.
- `value` is the arbitrary value string.

ducklake_data_file

Data files contain the actual row data.

Column name	Column type	
data_file_id	BIGINT	Primary key
table_id	BIGINT	
begin_snapshot	BIGINT	
end_snapshot	BIGINT	
file_order	BIGINT	
path	VARCHAR	
path_is_relative	BOOLEAN	
file_format	VARCHAR	
record_count	BIGINT	
file_size_bytes	BIGINT	
footer_size	BIGINT	
row_id_start	BIGINT	
partition_id	BIGINT	
encryption_key	VARCHAR	
mapping_id	BIGINT	
partial_max	BIGINT	

- `data_file_id` is the numeric identifier of the file. It is a primary key. `data_file_id` is incremented from `next_file_id` in the `ducklake_snapshot` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The file is part of the table *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The file is part of the table *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the file is currently part of the table.
- `file_order` is a number that defines the vertical position of the file in the table. It needs to be unique within a snapshot but does not have to be contiguous (gaps are ok).
- `path` is the file path of the data file, e.g., `my_file.parquet` for a relative path.
- `path_is_relative` whether the `path` is relative to the `path` of the table (true) or an absolute path (false).
- `file_format` is the storage format of the file. Currently, only `parquet` is allowed.
- `record_count` is the number of records (row) in the file.
- `file_size_bytes` is the size of the file in bytes.
- `footer_size` is the size of the file metadata footer, in the case of Parquet the Thrift data. This is an optimization that allows for faster reading of the file.
- `row_id_start` is the first logical row id in the file. (Every row has a unique row id that is maintained.)
- `partition_id` refers to a `partition_id` from the `ducklake_partition_info` table.
- `encryption_key` contains the encryption for the file if encryption is enabled.
- `mapping_id` refers to a `mapping_id` from the `ducklake_column_mapping` table.
- `partial_max` is the maximum snapshot id stored in a partial data file. When multiple snapshots are merged into a single file, per-row snapshot ownership is tracked via the `_ducklake_internal_snapshot_id` column embedded in the Parquet file. `partial_max` records the highest snapshot id present in that merged file, so reads and time travel can determine whether snapshot filtering is necessary. It is `NULL` for files that are not shared across snapshots.

ducklake_delete_file

Delete files contain the row ids of rows that are deleted. Each data file will have its own delete file if any deletes are present for this data file.

Column name	Column type	
`delete_file_id`	`BIGINT`	Primary key
`table_id`	`BIGINT`	
`begin_snapshot`	`BIGINT`	
`end_snapshot`	`BIGINT`	
`data_file_id`	`BIGINT`	
`path`	`VARCHAR`	
`path_is_relative`	`BOOLEAN`	
`format`	`VARCHAR`	
`delete_count`	`BIGINT`	
`file_size_bytes`	`BIGINT`	
`footer_size`	`BIGINT`	
`encryption_key`	`VARCHAR`	
`partial_max`	`BIGINT`	

- `delete_file_id` is the numeric identifier of the delete file. It is a primary key. `delete_file_id` is incremented from `next_file_id` in the `ducklake_snapshot` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The delete file is part of the table *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The delete file is part of the table *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the delete file is currently part of the table.
- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `path` is the file name of the delete file, e.g., `my_file-deletes.parquet` for a relative path.
- `path_is_relative` whether the `path` is relative to the `path` of the table (true) or an absolute path (false).
- `format` is the storage format of the delete file. Supported values are `parquet` (positional delete file) and `puffin` (Iceberg V3 deletion vector, experimental).
- `delete_count` is the number of deletion records in the file.
- `file_size_bytes` is the size of the file in bytes.
- `footer_size` is the size of the file metadata footer, in the case of Parquet the Thrift data. This is an optimization that allows for faster reading of the file.
- `encryption_key` contains the encryption for the file if encryption is enabled.
- `partial_max` is the maximum snapshot id stored in a partial deletion file. When multiple deletes target the same data file across different snapshots, the deletion file is rewritten as a partial deletion file that tracks which rows were deleted in which snapshot. This column stores the highest snapshot id present in such a file. It is `NULL` for standard (non-partial) delete files.

ducklake_file_column_stats

This table contains column-level statistics for a single data file.

Column name	Column type
`data_file_id`	`BIGINT`
`table_id`	`BIGINT`
`column_id`	`BIGINT`
`column_size_bytes`	`BIGINT`
`value_count`	`BIGINT`
`null_count`	`BIGINT`
`min_value`	`VARCHAR`
`max_value`	`VARCHAR`
`contains_nan`	`BOOLEAN`
`extra_stats`	`VARCHAR`

- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `column_size_bytes` is the byte size of the column.
- `value_count` is the number of values in the column. This does not have to correspond to the number of records in the file for nested types.
- `null_count` is the number of values in the column that are `NULL`.
- `min_value` contains the minimum value for the column, encoded as a string. This does not have to be exact but has to be a lower bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- `max_value` contains the maximum value for the column, encoded as a string. This does not have to be exact but has to be an upper bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- `contains_nan` is a flag whether the column contains any `NaN` values. This is only relevant for floating-point types.
- `extra_stats` contains additional type-specific statistics, such as bounding box information for geometry types or global shredded-field statistics for `variant` columns encoded as JSON.

ducklake_file_partition_value

This table defines which data file belongs to which partition.

Column name	Column type
`data_file_id`	`BIGINT`
`table_id`	`BIGINT`
`partition_key_index`	`BIGINT`
`partition_value`	`VARCHAR`

- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `partition_key_index` refers to a `partition_key_index` from the `ducklake_partition_column` table.
- `partition_value` is the value that all the rows in the data file have, encoded as a string.

ducklake_file_variant_stats

This table contains per-file statistics for the shredded sub-fields of `variant` columns.

Column name	Column type
data_file_id	BIGINT
table_id	BIGINT
column_id	BIGINT
variant_path	VARCHAR
shredded_type	VARCHAR
column_size_bytes	BIGINT
value_count	BIGINT
null_count	BIGINT
min_value	VARCHAR
max_value	VARCHAR
contains_nan	BOOLEAN
extra_stats	VARCHAR

- `data_file_id` refers to a `data_file_id` from the `ducklake_data_file` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table and identifies the `variant` column that contains the shredded field.
- `variant_path` is the path to the shredded sub-field within the variant. Named fields are always quoted (e.g., `"l_orderkey"`), and any quote characters within a field name are escaped by doubling them. Two special path values exist:
  - `root` — the variant value itself is a primitive (i.e., not nested).
  - `element` — the statistics apply to elements of an array-typed variant.
  - Paths can be composed, e.g., `element."a"` refers to the a field of each element of an array.
- `shredded_type` is the DuckLake type name of the shredded field (e.g., `int64`, `date`, `decimal(15,2)`, `varchar`).
- `column_size_bytes` is the byte size of the shredded field in this file.
- `value_count` is the number of non-null values in the shredded field.
- `null_count` is the number of rows where the shredded field is `NULL` or absent.
- `min_value` contains the minimum value for the shredded field, encoded as a string.
- `max_value` contains the maximum value for the shredded field, encoded as a string.
- `contains_nan` is a flag whether the shredded field contains any NaN values. Only relevant for floating-point types.
- `extra_stats` is reserved for additional type-specific statistics.

A row is written to this table for every **fully shredded** sub-field of a `variant` column in a data file. A sub-field is considered fully shredded when, for every row in the file, the field is either present as a primitive of a single consistent type, absent, or `NULL`. Fields that mix types across rows are not recorded.

Global (table-wide) statistics for shredded variant fields that are consistently shredded across every file are stored in the `extra_stats` column of the `ducklake_table_column_stats` table, encoded as JSON.

ducklake_files_scheduled_for_deletion

Files that are no longer part of any snapshot are scheduled for deletion.

Column name	Column type
data_file_id	BIGINT
path	VARCHAR

Column name	Column type
path_is_relative	BOOLEAN
schedule_start	TIMESTAMPTZ

- data_file_id refers to a data_file_id from the ducklake_data_file table.
- path is the file name of the file, e.g., my_file.parquet. The file name is either relative to the data_path value in ducklake_metadata or absolute. If relative, the path_is_relative field is set to true.
- path_is_relative defines whether the path is absolute or relative, see above.
- schedule_start is a timestamp of when this file was scheduled for deletion.

ducklake_inlined_data_tables

This table links DuckLake snapshots with inlined data tables.

Column name	Column type
table_id	BIGINT
table_name	VARCHAR
schema_version	BIGINT

- table_id refers to a table_id from the ducklake_table table.
- table_name is a string that names the data table for inlined data.
- schema_version refers to a schema version in the ducklake_snapshot table.

ducklake_macro

This table stores macro definitions. Each macro is associated with a schema and tracks its lifecycle through snapshots.

Column name	Column type
schema_id	BIGINT
macro_id	BIGINT
macro_name	VARCHAR
begin_snapshot	BIGINT
end_snapshot	BIGINT

- schema_id refers to a schema_id from the ducklake_schema table.
- macro_id is the numeric identifier of the macro. macro_id is incremented from next_catalog_id in the ducklake_snapshot table.
- macro_name is the name of the macro, e.g., my_macro.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The macro exists *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The macro exists *up to but not including* this snapshot id. If end_snapshot is NULL, the macro is currently valid.

ducklake_macro_impl

This table stores macro implementations. A single macro can have multiple implementations.

Column name	Column type
macro_id	BIGINT
impl_id	BIGINT
dialect	VARCHAR
sql	VARCHAR
type	VARCHAR

- macro_id refers to a macro_id from the ducklake_macro table.
- impl_id is the numeric identifier of the implementation within the macro.
- dialect is the SQL dialect of the macro implementation, e.g., duckdb.
- sql is the SQL expression or query that defines the macro body.
- type is the type of macro: scalar for scalar macros or table for table macros.

ducklake_macro_parameters

This table stores the parameters for each macro implementation.

Column name	Column type
macro_id	BIGINT
impl_id	BIGINT
column_id	BIGINT
parameter_name	VARCHAR
parameter_type	VARCHAR
default_value	VARCHAR
default_value_type	VARCHAR

- macro_id refers to a macro_id from the ducklake_macro table.
- impl_id refers to an impl_id from the ducklake_macro_impl table.
- column_id is the positional index of the parameter within the implementation.
- parameter_name is the name of the parameter.
- parameter_type is the DuckLake type of the parameter. Set to unknown if the parameter type is not specified.
- default_value is the default value of the parameter. Set to NULL if no default is specified.
- default_value_type is the DuckLake type of the default value. Set to unknown if no default is specified.

ducklake_metadata

The ducklake_metadata table contains key/value pairs with information about the specific setup of the DuckLake catalog.

Column name	Column type	
key	VARCHAR	Not NULL
value	VARCHAR	Not NULL
scope	VARCHAR	
scope_id	BIGINT	

- key is an arbitrary key string. See below for a list of pre-defined keys. The key cannot be NULL.
- value is the arbitrary value string. The value cannot be NULL.
- scope defines the scope of the setting.
- scope_id is the id of the item that the setting is scoped to (see the table below) or NULL for the Global scope.

Scope	scope	Description
Global	NULL	The scope of the setting is global for the entire catalog.
Schema	schema	The setting is scoped to the schema_id referenced by scope_id.
Table	table	The setting is scoped to the table_id referenced by scope_id.

Currently, the following values for key are specified:

Name	Description	Notes	Scope(s)
version	DuckLake format version.		Global
created_by	Tool used to write the DuckLake.		Global
table	A string that identifies which program wrote the schema, e.g., DuckDB v1.5.2.		Global
data_path	Path to data files, e.g., s3://mybucket/myprefix/.	Has to end in /	Global
encrypted	Whether or not to encrypt Parquet files written to the data path.	'true' or 'false'	Global
data_inlining_row_limit	Maximum amount of rows to inline in a single insert.		Global, Schema, Table
target_file_size	The target data file size for insertion and compaction operations.		Global, Schema, Table
parquet_row_group_size_bytes	Number of bytes per row group in Parquet files.		Global, Schema, Table
parquet_row_group_size	Number of rows per row group in Parquet files.		Global, Schema, Table
parquet_compression	Compression algorithm for Parquet files, e.g., zstd.	uncompressed, snappy, gzip, zstd, brotli, lz4, lz4_raw	Global, Schema, Table
parquet_compression_level	Compression level for Parquet files.		Global, Schema, Table
parquet_version	Parquet format version.	1 or 2	Global, Schema, Table

Name	Description	Notes	Scope(s)
`hive_file_pattern`	If partitioned data should be written in a Hive-style folder structure.	`'true'` or `'false'`	Global, Schema, Table
`require_commit_message`	If an explicit commit message is required for a snapshot commit.	`'true'` or `'false'`	Global
`rewrite_delete_threshold`	Minimum amount of data (0-1) that must be removed from a file before a rewrite is warranted.	Value between `0` and `1`	Global, Schema, Table
`delete_older_than`	How old unused files must be to be removed by the `ducklake_delete_orphaned_files` and `ducklake_cleanup_old_files` functions.	Duration string (e.g., `7d`, `24h`)	Global
`expire_older_than`	How old snapshots must be, by default, to be expired by `ducklake_expire_snapshots`.	Duration string (e.g., `30d`)	Global
`auto_compact`	Whether compaction functions run on a table. Defaults to `true`.	Used by `ducklake_flush_inlined_data`, `ducklake_merge_adjacent_files`, `ducklake_rewrite_data_files`, etc.	Global, Schema, Table
`per_thread_output`	Whether to create separate output files per thread during parallel insertion.	`'true'` or `'false'`	Global

ducklake_name_mapping

This table contains the information used to map a name to a `column_id` for a given `mapping_id` with the `map_by_name` type.

Column name	Column type
`mapping_id`	`BIGINT`
`column_id`	`BIGINT`
`source_name`	`VARCHAR`
`target_field_id`	`BIGINT`
`parent_column`	`BIGINT`
`is_partition`	`BOOLEAN`

- `mapping_id` refers to a `mapping_id` from the `ducklake_column_mapping` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `source_name` refers to the name of the field this mapping applies to.
- `target_field_id` refers to the `field-id` that a field with the `source_name` is mapped to.
- `parent_column` is the `column_id` of the parent column. This is `NULL` for top-level and non-nested columns. For example, for `STRUCT` types, this would refer to the "parent" `STRUCT` column.
- `is_partition` determines whether a column is used for hive-partitioning.

ducklake_partition_column

Partitions can refer to one or more columns, possibly with transformations such as hashing or bucketing.

Column name	Column type
`partition_id`	`BIGINT`
`table_id`	`BIGINT`
`partition_key_index`	`BIGINT`
`column_id`	`BIGINT`
`transform`	`VARCHAR`

- `partition_id` refers to a `partition_id` from the `ducklake_partition_info` table.
- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `partition_key_index` defines where in the partition key the column is using 0-based indexing. For example, in a partitioning by (a, b, c) the `partition_key_index` of b would be `1`.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `transform` defines the type of a transform that is applied to the column value, e.g., `year`.

The table of supported transforms is as follows.

Transform	Source type(s)	Description	Result type
`identity`	Any	Source value, unmodified	Source type
`bucket(N)`	Any	Hash the value with Murmur3 and assign to one of N buckets: `(murmur3_32(v) & INT_MAX) % N`	`int32`
`year`	`date`, `timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a date or timestamp year, as years from 1970	`int64`
`month`	`date`, `timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a date or timestamp month, as months from 1970-01-01	`int64`
`day`	`date`, `timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a date or timestamp day, as days from 1970-01-01	`int64`
`hour`	`timestamp`, `timestamptz`, `timestamp_s`, `timestamp_ms`, `timestamp_ns`	Extract a timestamp hour, as hours from 1970-01-01 00:00:00	`int64`

ducklake_partition_info

Column name	Column type
partition_id	BIGINT
table_id	BIGINT
begin_snapshot	BIGINT
end_snapshot	BIGINT

- partition_id is a numeric identifier for a partition. partition_id is incremented from next_catalog_id in the ducklake_snapshot table.
- table_id refers to a table_id from the ducklake_table table.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The partition is valid *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The partition is valid *up to but not including* this snapshot id. If end_snapshot is NULL, the partition is currently valid.

ducklake_schema

This table defines valid schemas.

Column name	Column type	
schema_id	BIGINT	Primary key
schema_uuid	UUID	
begin_snapshot	BIGINT	
end_snapshot	BIGINT	
schema_name	VARCHAR	
path	VARCHAR	
path_is_relative	BOOLEAN	

- schema_id is the numeric identifier of the schema. schema_id is incremented from next_catalog_id in the ducklake_snapshot table.
- schema_uuid is a UUID that gives a persistent identifier for this schema. The UUID is stored here for compatibility with existing lakehouse formats.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The schema exists *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The schema exists *up to but not including* this snapshot id. If end_snapshot is NULL, the schema is currently valid.
- schema_name is the name of the schema, e.g., my_schema.
- path is the data_path of the schema.
- path_is_relative whether the path is relative to the data_path of the catalog (true) or an absolute path (false).

ducklake_schema_versions

This table contains the schema versions for a range of snapshots. It is necessary to compact files with different schemas.

Column name	Column type
`begin_snapshot`	`BIGINT`
`schema_version`	`BIGINT`
`table_id`	`BIGINT`

- `begin_snapshot` refers to a `snapshot_id` in the `ducklake_snapshot` table.
- `schema_version` refers to the `schema_version` of a `ducklake_snapshot`.
- `table_id` refers to the id of the table, allowing to track the schema changes per table.

ducklake_snapshot

This table contains the valid snapshots in a DuckLake.

Column name	Column type	
`snapshot_id`	`BIGINT`	Primary key
`snapshot_time`	`TIMESTAMPTZ`	
`schema_version`	`BIGINT`	
`next_catalog_id`	`BIGINT`	
`next_file_id`	`BIGINT`	

- `snapshot_id` is the continuously increasing numeric identifier of the snapshot. It is a primary key and is referred to by various other tables.
- `snapshot_time` is the timestamp at which the snapshot was created.
- `schema_version` is a continuously increasing number that is incremented whenever the schema is changed, e.g., by creating a table. This allows for caching of schema information if only data is changed.
- `next_catalog_id` is a continuously increasing number that describes the next identifier for schemas, tables, views, partitions, and column name mappings. This is only changed if one of those entries is created, i.e., the schema is changing.
- `next_file_id` is a continuously increasing number that contains the next id for a data or deletion file to be added. It is only changed if data is being added or deleted, i.e., not for schema changes.

ducklake_snapshot_changes

This table lists changes that happened in a snapshot for easier conflict detection.

Column name	Column type	
`snapshot_id`	`BIGINT`	Primary key
`changes_made`	`VARCHAR`	
`author`	`VARCHAR`	
`commit_message`	`VARCHAR`	
`commit_extra_info`	`VARCHAR`	

The `ducklake_snapshot_changes` table contains a summary of changes made by a snapshot. This table is used during Conflict Resolution to quickly find out if two snapshots have conflicting changesets.

- snapshot_id refers to a snapshot_id from the ducklake_snapshot table.
- changes_made is a comma-separated list of high-level changes made by the snapshot. The values that are contained in this list have the following format:
  - created_schema:<schema_name> – the snapshot created a schema with the given name.
  - created_table:<table_name> – the snapshot created a table with the given name.
  - created_view:<view_name> – the snapshot created a view with the given name.
  - inserted_into_table:<table_id> – the snapshot inserted data into the given table.
  - deleted_from_table:<table_id> – the snapshot deleted data from the given table.
  - compacted_table:<table_id> – the snapshot ran a compaction operation on the given table.
  - dropped_schema:<schema_id> – the snapshot dropped the given schema.
  - dropped_table:<table_id> – the snapshot dropped the given table.
  - dropped_view:<view_id> – the snapshot dropped the given view.
  - altered_table:<table_id> – the snapshot altered the given table.
  - altered_view:<view_id> – the snapshot altered the given view.
- author is the author of the snapshot.
- commit_message is the commit message associated with the snapshot.
- commit_extra_info contains extra information regarding the commit.

> Names are written in quoted-format using SQL-style escapes, i.e., the name this "table" contains quotes is written as "this ""table"" contains quotes".

ducklake_sort_expression

The ducklake_sort_expression table stores the individual sort key expressions for each sort configuration. Each row corresponds to one expression in a SET SORTED BY clause.

Column name	Column type
sort_id	BIGINT
table_id	BIGINT
sort_key_index	BIGINT
expression	VARCHAR
dialect	VARCHAR
sort_direction	VARCHAR
null_order	VARCHAR

- sort_id refers to a sort_id from the ducklake_sort_info table.
- table_id refers to a table_id from the ducklake_table table.
- sort_key_index defines the position of this expression within the sort key using 0-based indexing. For example, in SET SORTED BY (a ASC, b DESC, c ASC) the sort_key_index of b is 1.
- expression is the sort expression as a string. Currently only column names are supported.
- dialect identifies the SQL dialect used to interpret expression. Currently always duckdb.
- sort_direction is either ASC or DESC.
- null_order is either NULLS_FIRST or NULLS_LAST.

ducklake_sort_info

The ducklake_sort_info table records the version history of sort settings for tables. Each row represents one sort configuration applied to a table, with snapshot-based validity tracking.

Column name	Column type
sort_id	BIGINT
table_id	BIGINT
begin_snapshot	BIGINT
end_snapshot	BIGINT

- sort_id is a numeric identifier for a sort configuration. sort_id is allocated from next_catalog_id in the ducklake_snapshot table.
- table_id refers to a table_id from the ducklake_table table.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The sort configuration is valid *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The sort configuration is valid *up to but not including* this snapshot id. If end_snapshot is NULL, the sort configuration is currently active.

The expressions associated with a sort configuration are stored in ducklake_sort_expression.

ducklake_table

This table describes tables. Inception!

Column name	Column type
table_id	BIGINT
table_uuid	UUID
begin_snapshot	BIGINT
end_snapshot	BIGINT
schema_id	BIGINT
table_name	VARCHAR
path	VARCHAR
path_is_relative	BOOLEAN

- table_id is the numeric identifier of the table. table_id is incremented from next_catalog_id in the ducklake_snapshot table.
- table_uuid is a UUID that gives a persistent identifier for this table. The UUID is stored here for compatibility with existing lakehouse formats.
- begin_snapshot refers to a snapshot_id from the ducklake_snapshot table. The table exists *starting with* this snapshot id.
- end_snapshot refers to a snapshot_id from the ducklake_snapshot table. The table exists *up to but not including* this snapshot id. If end_snapshot is NULL, the table is currently valid.
- schema_id refers to a schema_id from the ducklake_schema table.
- table_name is the name of the table, e.g., my_table.
- path is the data_path of the table.
- path_is_relative whether the path is relative to the path of the schema (true) or an absolute path (false).

ducklake_table_column_stats

This table contains column-level statistics for an entire table.

Column name	Column type
`table_id`	`BIGINT`
`column_id`	`BIGINT`
`contains_null`	`BOOLEAN`
`contains_nan`	`BOOLEAN`
`min_value`	`VARCHAR`
`max_value`	`VARCHAR`
`extra_stats`	`VARCHAR`

- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `column_id` refers to a `column_id` from the `ducklake_column` table.
- `contains_null` is a flag whether the column contains any `NULL` values.
- `contains_nan` is a flag whether the column contains any `NaN` values. This is only relevant for floating-point types.
- `min_value` contains the minimum value for the column, encoded as a string. This does not have to be exact but has to be a lower bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- `max_value` contains the maximum value for the column, encoded as a string. This does not have to be exact but has to be an upper bound. The value has to be cast to the actual type for accurate comparison, e.g., on integer types.
- `extra_stats` contains additional type-specific statistics, such as bounding box information for geometry types or global shredded-field statistics for `variant` columns encoded as JSON. Variant global stats are populated only for sub-fields that are consistently shredded across every data file; if any file contains inconsistent data for a field, global stats for that field are omitted.

ducklake_table_stats

This table contains table-level statistics.

Column name	Column type
`table_id`	`BIGINT`
`record_count`	`BIGINT`
`next_row_id`	`BIGINT`
`file_size_bytes`	`BIGINT`

- `table_id` refers to a `table_id` from the `ducklake_table` table.
- `record_count` is the total amount of rows in the table. This can be approximate.
- `next_row_id` is the row id for newly inserted rows. Used for row lineage tracking.
- `file_size_bytes` is the total file size of all data files in the table. This can be approximate.

ducklake_tag

Schemas, tables, and views, etc. can have tags, those are declared in this table.

Column name	Column type
`object_id`	`BIGINT`
`begin_snapshot`	`BIGINT`

Column name	Column type
`end_snapshot`	`BIGINT`
`key`	`VARCHAR`
`value`	`VARCHAR`

- `object_id` refers to a `schema_id`, `table_id`, etc. from various tables above.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The tag is valid *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The tag is valid *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the tag is currently valid.
- `key` is an arbitrary key string. The key can't be `NULL`.
- `value` is the arbitrary value string.

ducklake_view

This table describes SQL-style `VIEW` definitions.

Column name	Column type
`view_id`	`BIGINT`
`view_uuid`	`UUID`
`begin_snapshot`	`BIGINT`
`end_snapshot`	`BIGINT`
`schema_id`	`BIGINT`
`view_name`	`VARCHAR`
`dialect`	`VARCHAR`
`sql`	`VARCHAR`
`column_aliases`	`VARCHAR`

- `view_id` is the numeric identifier of the view. `view_id` is incremented from `next_catalog_id` in the `ducklake_snapshot` table.
- `view_uuid` is a UUID that gives a persistent identifier for this view. The UUID is stored here for compatibility with existing lakehouse formats.
- `begin_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The view exists *starting with* this snapshot id.
- `end_snapshot` refers to a `snapshot_id` from the `ducklake_snapshot` table. The view exists *up to but not including* this snapshot id. If `end_snapshot` is `NULL`, the view is currently valid.
- `schema_id` refers to a `schema_id` from the `ducklake_schema` table.
- `view_name` is the name of the view, e.g., `my_view`.
- `dialect` is the SQL dialect of the view definition, e.g., duckdb.
- `sql` is the SQL string that defines the view, e.g., `SELECT * FROM my_table`.
- `column_aliases` contains a possible rename of the view columns. Can be `NULL` if no rename is set.

DuckDB Extension

Introduction

In DuckDB, DuckLake is supported through the `ducklake` extension.

Installation

Install the latest DuckDB release.

```
INSTALL ducklake;
```

Configuration

To use DuckLake, you need to make two decisions: which metadata catalog database you want to use and where you want to store those files. In the simplest case, you use a local DuckDB file for the metadata catalog and a local folder on your computer for file storage.

Creating a New Database

DuckLake databases are created by simply starting to use them with the ATTACH statement. In the simplest case, you can create a local, DuckDB-backed DuckLake like so:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
```

This will create a file `my_ducklake.ducklake`, which is a DuckDB database with the DuckLake schema.

We also use USE so we don't have to prefix all table names with `my_ducklake`. Once data is inserted, this will also create a folder `my_ducklake.ducklake.files` in the same directory, where Parquet files are stored.

If you would like to use another directory, you can specify this in the DATA_PATH parameter for ATTACH:

```
ATTACH 'ducklake:my_other_ducklake.ducklake' AS my_other_ducklake (DATA_PATH 'some/other/path/');
USE my_other_ducklake;
```

The path is stored in the DuckLake metadata and does not have to be specified again to attach to an existing DuckLake catalog.

> Both DATA_PATH and the database file path should be relative paths (e.g., `./some/path/` or `some/path/`). Moreover, for database creation the path needs to exist already, i.e., `ATTACH 'ducklake:db/my_ducklake.ducklake' AS my_ducklake;` where db needs to be an existing directory.

Attaching an Existing Database

Attaching to an existing database also uses the ATTACH syntax. For example, to re-connect to the example from the previous section in a new DuckDB session, we can just type:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
```

If you use the DuckDB command line client, you can pass it the filename or URL of the DuckLake as an argument:

```
duckdb ducklake:my_ducklake.ducklake
```

Using DuckLake

DuckLake is used just like any other DuckDB database. You can create schemas and tables, insert data, update data, delete data, modify table schemas, etc.

Note that – similarly to other data lake and lakehouse formats – the DuckLake format does not support indexes, primary keys, foreign keys, and `UNIQUE` or `CHECK` constraints.

Don't forget to either specify the database name of the DuckLake explicitly or use `USE`. Otherwise you might inadvertently use the temporary, in-memory database.

Example

Let's observe what happens in DuckLake when we interact with a dataset. We will use the Netherlands train traffic dataset here.

We use the example DuckLake from above:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
USE my_ducklake;
```

Let's now import the dataset into a new table:

```
CREATE TABLE nl_train_stations AS
    FROM 'https://blobs.duckdb.org/nl_stations.csv';
```

Now let's peek behind the curtains. The data was just read into a Parquet file, which we can also just query.

```
FROM glob('my_ducklake.ducklake.files/**/*');
FROM 'my_ducklake.ducklake.files/**/*.parquet' LIMIT 10;
```

But now let's change some things around. We're really unhappy with the old name of the ”Amsterdam Bijlmer ArenA” station now that the stadium has been renamed to ”Johan Cruijff ArenA” and everyone here loves Johan. So let's change that.

```
UPDATE nl_train_stations
SET name_long = 'Johan Cruijff ArenA'
WHERE code = 'ASB';
```

Poof, it's changed. We can confirm:

```
SELECT name_long
FROM nl_train_stations
WHERE code = 'ASB';
```

In the background, more files have appeared:

```
FROM glob('my_ducklake.ducklake.files/**/*');
```

We now see three files. The original data file, the rows that were deleted, and the rows that were inserted. Like most systems, DuckLake models updates as deletes followed by inserts. The deletes are just a Parquet file, we can query it:

```
FROM 'my_ducklake.ducklake.files/**/ducklake-*-delete.parquet';
```

The file should contain a single row that marks row 29 as deleted. A new file has appeared that contains the new values for this row.

There are now three snapshots, the table creation, data insertion, and the update. We can query that using the `snapshots()` function:

```
FROM my_ducklake.snapshots();
```

And we can query this table at each point:

```
SELECT name_long
FROM nl_train_stations AT (VERSION => 1)
WHERE code = 'ASB';
```

```
SELECT name_long
FROM nl_train_stations AT (VERSION => 2)
WHERE code = 'ASB';
```

Time travel finally achieved!

Detaching from a DuckLake

To detach from a DuckLake, make sure that your DuckLake is not your default database, then use the DETACH statement:

```
USE memory;
DETACH my_ducklake;
```

Using DuckLake from a Client

DuckLake v1.0 is supported by with DuckDB v1.5.2+.

Usage

Connecting

To use DuckLake, you must first either connect to an existing DuckLake, or create a new DuckLake. The ATTACH command can be used to select the DuckLake instance to connect to. In the ATTACH command, you must specify the catalog database and the data storage location. When attaching, a new DuckLake is automatically created if none exists in the specified catalog database.

Note that the data storage location only has to be specified when creating a new DuckLake. When connecting to an existing DuckLake, the data storage location is loaded from the catalog database.

```
ATTACH 'ducklake:<metadata_storage_location>' (DATA_PATH '<data_storage_location>');
```

In addition, DuckLake connection parameters can also be stored in secrets.

```
ATTACH 'ducklake:<secret_name>';
```

Examples

Connect to DuckLake, reading the configuration from the default (unnamed) secret:

```
ATTACH 'ducklake:';
```

Connect to DuckLake, reading the configuration from the secret named `my_secret`:

```
ATTACH 'ducklake:<my_secret>';
```

Use a DuckDB database `duckdb_database.ducklake` as the catalog database with the data path defaulting to `duckdb_database.ducklake.files`:

```
ATTACH 'ducklake:<duckdb_database.ducklake>';
```

Use a DuckDB database `duckdb_database.ducklake` as the catalog database with the data path explicitly specified as the `my_files` directory:

```
ATTACH 'ducklake:<duckdb_database.ducklake>' (DATA_PATH '<my_files/>');
```

Use a PostgreSQL database as the catalog database and an S3 path as the data path:

```
ATTACH 'ducklake:postgres:dbname=postgres' (DATA_PATH 's3://<my-bucket/my-data/>');
```

Connect to DuckLake in read-only mode:

```
ATTACH 'ducklake:postgres:dbname=postgres' (READ_ONLY);
```

It is also possible to override the data path for a particular connection. This will not change the value of the `DATA_PATH` stored in the DuckLake metadata, but it will override it for the current connection allowing data to be stored in a different path.

```
ATTACH 'ducklake:<duckdb_database.ducklake>' (DATA_PATH '<other_data_path/>', OVERRIDE_DATA_PATH true);
```

> If `OVERRIDE_DATA_PATH` is used, data under the original `DATA_PATH` will not be able to be queried in the current connection. This behavior may be changed in the future to allow to query data in a catalog regardless of the current write `DATA_PATH`.

Parameters

The following parameters are supported for ATTACH:

Name	Description	Default
CREATE_IF_NOT_EXISTS	Creates a new DuckLake if the specified one does not already exist	true
DATA_INLINING_ROW_LIMIT	The number of rows for which data inlining is used	0
DATA_PATH	The storage location of the data files	<metadata_file>.files for DuckDB files, required otherwise
ENCRYPTED	Whether or not data is stored encrypted	false
META_<PARAMETER_NAME>	Pass <PARAMETER_NAME> to the catalog server	
METADATA_CATALOG	The name of the attached catalog database	__ducklake_metadata_<ducklake_name>
METADATA_PARAMETERS	Map of parameters to pass to the catalog server	{}
METADATA_PATH	The connection string for connecting to the metadata catalog	
METADATA_SCHEMA	The schema in the catalog server in which to store the DuckLake tables	main
AUTOMATIC_MIGRATION	Automatically migrates the DuckLake catalog schema if the version does not match	false
OVERRIDE_DATA_PATH	If the path provided in data_path differs from the stored path and this option is set to true, the path is overridden	true
SNAPSHOT_TIME	If provided, connect to DuckLake at a snapshot at a specified point in time	
SNAPSHOT_VERSION	If provided, connect to DuckLake at a specified snapshot id	

In addition, any parameters that are prefixed with META_ are passed to the catalog used to store the metadata. The supported parameters depend on the metadata catalog that is used. For example, postgres supports the SECRET parameter. By using the META_SECRET parameter we can pass this parameter to the PostgreSQL instance.

Secrets

Instead of configuring the connection using ATTACH, secrets can be created that contain all required information for setting up a connection. Secrets support the same list of parameters as ATTACH, in addition to the METADATA_PATH and METADATA_PARAMETERS parameters.

Name	Description	Default
METADATA_PATH	The connection string for connecting to the metadata	
METADATA_PARAMETERS	Map of parameters to pass to the catalog server	{}

```
-- Default (unnamed) secret
CREATE SECRET (
    TYPE ducklake,
```

```
    METADATA_PATH '<metadata.duckdb>',
    DATA_PATH '<metadata_files/>'
);

ATTACH 'ducklake:' AS my_ducklake;

-- Named secrets
CREATE SECRET <my_secret> (
    TYPE ducklake,
    METADATA_PATH '',
    DATA_PATH 's3://<my-s3-bucket>/',
    METADATA_PARAMETERS MAP {'TYPE': 'postgres', 'SECRET': 'postgres_secret'}
);
ATTACH 'ducklake:<my_secret>' AS my_ducklake;
```

To persist secrets, use the `CREATE PERSISTENT SECRET` statement.

Choosing a Catalog Database

You may choose different *catalog databases* for your DuckLake. The choice depends on several factors, including whether you need to use multiple clients, which database systems are available in your organization, etc.

On the technical side, consider the following:

- If you would like to perform **local data warehousing with a single client**, use DuckDB as the catalog database.
- If you would like to perform **local data warehousing using multiple local clients**, use SQLite as the catalog database.
- If you would like to operate a **multi-user lakehouse** with potentially remote clients, use PostgreSQL as the catalog database.

DuckDB

DuckDB can, of course, natively connect to DuckDB database files. So, to get started with using DuckDB *as your DuckLake catalog database,* you only need to install the `ducklake` extension and attach to your DuckLake:

```
INSTALL ducklake;

ATTACH 'ducklake:metadata.ducklake' AS my_ducklake;
USE my_ducklake;
```

Note that if you are using DuckDB as your catalog database, you're limited to a single client.

PostgreSQL

DuckDB can interact with a PostgreSQL database using the `postgres` extension. Install the `ducklake` and the `postgres` extension, and attach to your DuckLake as follows:

```
INSTALL ducklake;
INSTALL postgres;

-- Make sure that the database `ducklake_catalog` exists in PostgreSQL
ATTACH 'ducklake:postgres:dbname=ducklake_catalog host=localhost' AS my_ducklake
    (DATA_PATH 'data_files/');
USE my_ducklake;
```

For details on how to configure the connection, see the `postgres` extension's documentation.

The `ducklake` and `postgres` extensions require PostgreSQL 12 or newer.

SQLite

DuckDB can read and write a SQLite database file using the `sqlite` extension. Install the `ducklake` and the `sqlite` extension, and attach to your DuckLake as follows:

```
INSTALL ducklake;
INSTALL sqlite;

ATTACH 'ducklake:sqlite:metadata.sqlite' AS my_ducklake
    (DATA_PATH 'data_files/');
USE my_ducklake;
```

While SQLite doesn't allow concurrent reads and writes, its default mode is to ATTACH and DETACH for every query, together with providing a “retry time-out” for queries when a write-lock is encountered. This allows a reasonable amount of multi-processing support (effectively hiding the single-writer model).

MySQL

> **Warning.** There are a number of known issues with MySQL as a catalog for DuckLake. This is due to some limitations regarding the DuckDB MySQL connector. We therefore do not recommend to use MySQL as a catalog for DuckLake.

DuckDB can interact with a MySQL database using the `mysql` extension. Install the `ducklake` and the `mysql` extension, and attach to your DuckLake as follows:

```
INSTALL ducklake;
INSTALL mysql;

-- Make sure that the database `ducklake_catalog` exists in MySQL
ATTACH 'ducklake:mysql:db=ducklake_catalog host=localhost' AS my_ducklake
    (DATA_PATH 'data_files/');
USE my_ducklake;
```

For details on how to configure the connection, see the `mysql` extension's documentation.

Using the `ducklake` and `mysql` extensions requires MySQL 8 or newer.

Choosing Storage

DuckLake as a concept will *never* change existing files, neither by changing existing content nor by appending to existing files. This greatly reduces the consistency requirements of file systems and greatly simplifies caching.

The DuckDB `ducklake` extension can work with any file system backend that DuckDB supports. This currently includes:

- Local files and folders
- Cloud object store like
  - AWS S3 and compatible (e.g., Cloudflare R2, Hetzner Object Storage, etc.)
  - Google Cloud Storage
  - Azure Blob Store
- Virtual network attached file systems
  - NFS
  - SMB
  - FUSE
  - Python fsspec file systems
  - etc.

When choosing storage, it's important to consider the following factors

- *Access latency and data transfer throughput*, a cloud further away will be accessible to everyone but have a higher latency. local files are very fast, but not accessible to anyone else. A compromise might be a site-local storage server.
- *Scalability and cost*, an object store is quite scalable, but potentially charges for data transfer. A local server might not incur significant operating expenses, but might struggle serving thousands of clients.

It might also be interesting to use DuckLake encryption when choosing external cloud storage.

Snapshots

Snapshots represent commits made to DuckLake. Every snapshot performs a set of changes that alter the state of the database. Snapshots can create tables, insert or delete data, and alter schemas.

Changes can only be made to DuckLake using snapshots. Every set of changes must be accompanied by a snapshot.

Listing Snapshots

The set of snapshots can be queried using the `snapshots` function. This returns a list of all snapshots and their changesets.

```
ATTACH 'ducklake:my_ducklake.duckdb' AS my_ducklake;
CREATE TABLE my_ducklake.people (a INTEGER, b VARCHAR);
INSERT INTO my_ducklake.people VALUES (1, 'pedro');
SELECT * FROM my_ducklake.snapshots();
```

snapshot_id	snapshot_time	schema_version	changes	author	commit_message	commit_extra_info
0	2026-04-10 12:57:02.432386+02	0	{schemas_created=[main]}	NULL	NULL	NULL
1	2026-04-10 12:57:02.439404+02	1	{tables_created=[main.people]}	NULL	NULL	NULL
2	2026-04-10 12:57:02.449289+02	1	{inlined_insert=[1]}	NULL	NULL	NULL

It is also possible to retrieve the latest snapshot id directly with a function.

```
FROM my_ducklake.current_snapshot();
```

id
2

The DuckLake extension also provides a function to get the latest committed snapshot for an existing open connection. This may be useful when multiple connections are updating the same target.

```
FROM my_ducklake.last_committed_snapshot();
```

Which would return the following for the current connection:

id
2

But if a new connection is open, it will return:

id
NULL

Adding a Commit Message to a Snapshot

An author and commit message can also be added in the context of a transaction. Optionally, you can also add some extra information.

```
ATTACH 'ducklake:my_ducklake.duckdb' AS my_ducklake;
CREATE TABLE my_ducklake.people (a INTEGER, b VARCHAR);

-- Begin Transaction
BEGIN;
INSERT INTO my_ducklake.people VALUES (1, 'pedro');
CALL my_ducklake.set_commit_message('Pedro', 'Inserting myself', extra_info => '{''foo'': 7, ''bar'':
10}');
COMMIT;
-- End transaction
```

Query the snapshots:

```
SELECT * FROM my_ducklake.snapshots();
```

snapshot_id	snapshot_time	schema_version	changes	author	commit_message	commit_extra_info
0	2026-04-10 12:57:40.148846+02	0	{schemas_created=[main]}	NULL	NULL	NULL
1	2026-04-10 12:57:40.155454+02	1	{tables_created=[main.people]}	NULL	NULL	NULL
2	2026-04-10 12:57:40.168217+02	1	{inlined_insert=[1]}	Pedro	Inserting myself	{'foo': 7, 'bar': 10}

Schema Evolution

DuckLake supports the evolution of the schemas of tables without requiring any data files to be rewritten. The schema of a table can be changed using the `ALTER TABLE` statement. The following statements are supported:

Adding Columns / Fields

To add a new column of type `INTEGER`, with default value `NULL`, use:

```
ALTER TABLE tbl ADD COLUMN new_column INTEGER;
```

To add a new column with an explicit default value, use:

```
ALTER TABLE tbl ADD COLUMN new_column VARCHAR DEFAULT 'my_default';
```

Fields can be added to columns of type `struct`. The path to the `struct` column must be specified, followed by the name of the new field and the type of the new field.

```
-- Add a new field of type INTEGER, with default value NULL
ALTER TABLE tbl ADD COLUMN nested_column.new_field INTEGER;
```

Dropping Columns / Fields

To drop the top-level column `new_column` from the table, use:

```
ALTER TABLE tbl DROP COLUMN new_column;
```

Fields can be dropped by specifying the full path to the field. For example, to drop the field `new_field` from the struct column `nested_column`, use:

```
ALTER TABLE tbl DROP COLUMN nested_column.new_field;
```

Renaming Columns / Fields

To rename the top-level column `new_column` to `new_name`, use:

```
ALTER TABLE tbl RENAME new_column TO new_name;
```

Fields can be renamed by specifying the full path to the field. For example, to rename the field `new_field` within the struct column `nested_column` to `new_name`:

```
ALTER TABLE tbl RENAME nested_column.new_field TO new_name;
```

Renaming Tables

To rename the table `tbl` to `tbl_new_name`, use:

```
ALTER TABLE tbl RENAME TO tbl_new_name;
```

Type Promotion

The types of columns can be changed.

To change the type of `col1` to `BIGINT`, use:

```
ALTER TABLE tbl ALTER col1 SET TYPE BIGINT;
```

To change the type of field `new_field` within the struct column `nested_column` to `BIGINT`:

```
ALTER TABLE tbl ALTER nested_column.new_field SET TYPE BIGINT;
```

Note that not all type changes are valid. Only *type promotions* are supported. Type promotions must be lossless. As such, valid type promotions are promoting from a narrower type (`int32`) to a wider type (`int64`).

The full set of valid type promotions is as follows:

Source	Target
`int8`	`int16`, `int32`, `int64`
`int16`	`int32`, `int64`
`int32`	`int64`
`uint8`	`uint16`, `uint32`, `uint64`
`uint16`	`uint32`, `uint64`
`uint32`	`uint64`
`float32`	`float64`

Field Identifiers

Columns are tracked using **field identifiers**. These identifiers are stored in the `column_id` field of the `ducklake_column` table. The identifiers are also written to each of the data files. For Parquet files, these are written in the `field_id` field. These identifiers are used to reconstruct the data of a table for a given snapshot.

When reading the data for a table, the schema together with the correct field identifiers is read from the `ducklake_column` table. Data files can contain any number of columns that exist in that schema, and can also contain columns that do not exist in that schema.

- If we drop a column, previously written data files still contain the dropped column.
- If we add a column, previously written data files do not contain the new column.
- If we change the type of a column, previously written data files contain data for the column in the old type.

To reconstruct the correct table data for a given snapshot, we must perform *field id remapping*. This is done as follows:

- Data for a column is read from the column with the corresponding `field_id`. The data types might not match in case of type promotion. In this case, the values must be cast to the correct type of the column.
- Any column that has a `field_id` that exists in the data file but not in the table schema must be ignored
- Any column that has a `field_id` that does not exist in the data file must be replaced with the `initial_default` value in the `ducklake_column` table

Time Travel

In DuckLake, every snapshot represents a consistent state of the database. DuckLake keeps a record of all historic snapshots and their changesets, unless compaction is triggered and historic snapshots are explicitly deleted.

Using time travel, it is possible to query the state of the database as of any recorded snapshot. The snapshot to query can be specified either (1) using a timestamp, or (2) explicitly using a snapshot identifier. The `snapshots` function can be used to obtain a list of valid snapshots for a given DuckLake database.

Examples

Query the table at a specific snapshot version.

```
SELECT * FROM tbl AT (VERSION => 3);
```

Query the table as it was last week.

```
SELECT * FROM tbl AT (TIMESTAMP => now() - INTERVAL '1 week');
```

Attach a DuckLake database at a specific snapshot version.

```
ATTACH 'ducklake:metadata.duckdb' (SNAPSHOT_VERSION 3);
```

Attach a DuckLake database as it was at a specific time.

```
ATTACH 'ducklake:metadata.duckdb' (SNAPSHOT_TIME '2025-05-26 00:00:00');
```

Upserting

Upserting is the combination of updating and inserting. In database operations this usually means *do something to a record if it already exists* and *do something else if it doesn't*. Many databases support primary keys to assist with this behavior. This is also the case with DuckDB, which allows for the syntax `INSERT INTO ... ON CONFLICT`.

DuckLake, on the other hand, does not support primary keys. However, the `MERGE INTO` syntax provides the same upserting functionality.

Syntax

```
MERGE INTO target_table [target_alias]
    USING source_table [source_alias]
    ON (target_table.field = source_table.field) -- USING (field)
    WHEN MATCHED THEN UPDATE [SET] | DELETE
    WHEN NOT MATCHED THEN INSERT;
```

Usage

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 updating or inserting a whole row.

```
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;
```

id	name	salary
2	Anna	100000.0
1	John	105000.0
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.

```
MERGE INTO people
    USING (
        SELECT
            1 AS id,
            98_000.0 AS salary
    ) AS salary_updates
    ON (salary_updates.id = people.id)
    WHEN MATCHED THEN UPDATE SET salary = salary_updates.salary;

FROM people;
```

id	name	salary
2	Anna	100000.0
3	Sarah	95000.0
1	John	98000.0

id	name	salary

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
    ON (deletes.id = people.id)
    WHEN MATCHED THEN DELETE;

FROM people;
```

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` greater than or equal to a certain amount.

```
MERGE INTO people
    USING (
        SELECT
            unnest([3, 2]) AS id
    ) AS deletes
    ON (deletes.id = people.id)
    WHEN MATCHED AND people.salary >= 100_000.0 THEN DELETE;

FROM people;
```

id	name	salary
3	Sarah	95000.0

Unsupported Behavior

Multiple `UPDATE` or `DELETE` operators are not currently supported. The following query **would not work:**

```
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 AND people.salary < 100_000.0 THEN UPDATE
    -- Second update or delete condition
    WHEN MATCHED AND people.salary > 100_000.0 THEN DELETE
    WHEN NOT MATCHED THEN INSERT;
```

```
Not implemented Error:
MERGE INTO with DuckLake only supports a single UPDATE/DELETE action currently
```

Configuration

ducklake Extension Configuration

The `ducklake` extension also allows for some configuration regarding retry mechanism for transaction conflicts.

Name	Description	Default
`ducklake_default_data_inlining_row_limit`	Default row limit for data inlining across all connections (0 disables inlining)	`10`
`ducklake_max_retry_count`	The maximum amount of retry attempts for a DuckLake transaction	`10`
`ducklake_retry_backoff`	Backoff factor for exponentially increasing retry wait time	`1.5`
`ducklake_retry_wait_ms`	Time between retries in ms	`100`
`ducklake_write_deletion_vectors`	**Experimental.** Write Iceberg V3 deletion vectors (puffin) instead of positional delete files (parquet)	`false`

> **Warning.** Deletion vectors are an experimental feature.

To set these configuration options, use the SET statement:

```
SET ducklake_default_data_inlining_row_limit = 50;
SET ducklake_max_retry_count = 100;
SET ducklake_retry_backoff = 2;
SET ducklake_retry_wait_ms = 100;
```

DuckLake-Specific Configuration

DuckLake supports persistent and scoped configuration operations. These options can be set using the `set_option` function call. The options that have been set can be queried using the `options` function. Configuration is persisted in the `ducklake_metadata` table.

DuckLake-Specific Configuration Options

Name	Description	Default
`auto_compact`	Whether a table is included when compaction functions are called without a specific table argument	`true`
`created_by`	Tool used to write the DuckLake	
`data_inlining_row_limit`	Maximum amount of rows to inline in a single insert	`10`
`data_path`	Path to data files	
`delete_older_than`	How old unused files must be to be removed by cleanup functions	
`encrypted`	Whether or not to encrypt Parquet files written to the data path	`false`
`expire_older_than`	How old snapshots must be to be expired by default	
`hive_file_pattern`	If partitioned data should be written in a Hive-style folder structure	`true`
`parquet_compression_level`	Compression level for Parquet files	`3`

Name	Description	Default
`parquet_compression`	Compression algorithm for Parquet files (uncompressed, snappy, gzip, zstd, brotli, lz4, lz4_raw)	`snappy`
`parquet_row_group_size_bytes`	Number of bytes per row group in Parquet files	
`parquet_row_group_size`	Number of rows per row group in Parquet files	`122880`
`parquet_version`	Parquet format version (1 or 2)	`1`
`per_thread_output`	Whether to create separate output files per thread during parallel insertion	`false`
`require_commit_message`	If an explicit commit message is required for a snapshot commit	`false`
`rewrite_delete_threshold`	Minimum fraction of data removed from a file before a rewrite is warranted (0...1)	`0.95`
`sort_on_insert`	Whether to sort data on `INSERT` according to `SET SORTED BY`	`true`
`target_file_size`	The target data file size for insertion and compaction operations	`512MB`
`version`	DuckLake format version	
`write_deletion_vectors`	**Experimental.** Write Iceberg V3 deletion vectors (puffin) instead of positional delete files (parquet)	`false`

Setting DuckLake-Specific Configuration Values

Set the global Parquet compression algorithm used when writing Parquet files:

```
CALL my_ducklake.set_option('parquet_compression', 'zstd');
```

Set the Parquet compression algorithm used for tables in a specific schema:

```
CALL my_ducklake.set_option('parquet_compression', 'zstd', schema => 'my_schema');
```

Set the Parquet compression algorithm used for a specific table:

```
CALL my_ducklake.set_option('parquet_compression', 'zstd', table_name => 'my_table');
```

See all options for the given attached DuckLake

```
FROM my_ducklake.options();
```

Scoping

Options can be set either globally, per-schema or per-table. The most specific scope that is set is always used for any given setting, i.e., settings are used in the following priority:

Priority	Setting Scope
1	Table
2	Schema
3	Global
4	Default

DuckLake Instance Settings

The `ducklake_settings` function returns metadata about a DuckLake instance: the catalog type, the extension version and the data path.

Column	Description
`catalog_type`	Metadata catalog backend (`duckdb`, `postgres`, `sqlite`)
`extension_version`	Version of the `ducklake` extension
`data_path`	Path where data files are stored

```
FROM ducklake_settings('my_ducklake');
```

Using the convenience macro on an attached DuckLake:

```
FROM my_ducklake.settings();
```

Paths

DuckLake manages files stored in a separate storage location. The paths to the files are stored in the catalog server. Paths can be either absolute or relative to their parent path. Whether or not a path is relative is stored in the `path_is_relative` column, alongside the `path`. By default, all paths written by DuckLake are relative paths.

Path type	Path location	Parent path
File path	`ducklake_data_file` / `ducklake_delete_file`	Table path
Table path	`ducklake_table`	Schema path
Schema path	`ducklake_schema`	Data path
Data path	`ducklake_metadata`	

Default Path Structure

The root `data_path` is specified through the `data_path` parameter when creating a new DuckLake. When loading an existing DuckLake, the `data_path` is loaded from the `ducklake_metadata` if not provided.

Schemas. When creating a schema, a schema path is set. By default, this path is the name of the schema for alphanumeric names `<schema_name>/`, or `<schema_uuid>/` otherwise. This path is set as relative to the root `data_path`.

Tables. When creating a table, a table path is set. By default, this path is the name of the table for alphanumeric names `<table_name>>/`, or `<table_uuid>/` otherwise. This path is set as relative to the path of the parent schema.

Files. When writing a new data or delete file to the table, a new file path is generated. For unpartitioned tables, this path is `ducklake-<uuid>.parquet` – relative to the table path.

Partitioned Files. When writing data to a partitioned table, the files are by default written to directories in the Hive partitioning style. Writing data in this manner is not required as the partition values are tracked in the catalog server itself.

This results in the following path structure:

```
main
├── unpartitioned_table
│   └── ducklake-<uuid>.parquet
└── partitioned_table
    └── year=2025
        └── ducklake-<uuid>.parquet
```

Maintenance

Recommended Maintenance

Metadata Maintenance

Most operations performed by DuckLake happen in the catalog database. As such, the maintenance of the metadata server are handled by the chosen catalog database. For example, when running PostgreSQL, it is likely sufficient to occasionally run `VACUUM` in order to ensure the system stays performant.

Data File Maintenance

The data files that DuckLake writes to the data directory may require maintenance depending on how the insertions take place. When snapshots write small batches of data at a time and data inlining is not used small Parquet files will be written to storage. It is recommended to merge these Parquet files using the `merge_adjacent_files` function.

DuckLake also never deletes old data files. As old data remains accessible through time travel. Even when a table is dropped, the data files associated with that table are not deleted. In order to trigger a delete of these files, the snapshots that refer to that table must be expired and files should be cleaned up.

If you have tables that are heavily deleted, it can be the case that you have a lot of delete files that will slow read performance. In this case, we recommend you run a function to rewrite the deleted files.

If you need to run all of this operations periodically, then we recommend you use the `CHECKPOINT` statement.

Merge Adjacent Files

Unless data inlining is used, each insert to DuckLake writes data to a new Parquet file. If small insertions are performed, the Parquet files that are written are small. These only hold a few rows. For performance reasons, it is generally recommended that Parquet files are at least a few megabytes each.

DuckLake supports merging of files **without needing to expire snapshots**. Effectively, we can merge multiple Parquet files into a single Parquet file that holds data inserted by multiple snapshots. The resulting file is a *partial data file*: per-row snapshot ownership is tracked via a `_ducklake_internal_snapshot_id` column embedded in the Parquet file, and the highest snapshot id present in the merged file is stored in the `partial_max` column of `ducklake_data_file`.

This preserves all of the original behavior – including time travel and data change feeds – for these snapshots. In effect, this manner of compaction is completely transparent from a user perspective.

This compaction technique can be triggered using the `merge_adjacent_files` function. For example:

```
CALL ducklake_merge_adjacent_files('my_ducklake');
```

Or if you want to target a specific table within a schema:

```
CALL ducklake_merge_adjacent_files('my_ducklake', 't', schema => 'some_schema');
```

Controlling Which Tables Are Compacted

When `ducklake_merge_adjacent_files` is called without a table argument, it runs on all tables where the `auto_compact` option is `true` (the default). This lets you opt specific tables or schemas out of bulk compaction calls.

> **Note** `auto_compact` does not trigger compaction automatically after writes. Compaction always runs explicitly when you call a maintenance function. The option only controls *which tables are included* when the function is called without a specific table argument.

Disable compaction for a specific table:

```
CALL my_ducklake.set_option('auto_compact', false, table_name => 'my_table');
```

Disable compaction for an entire schema, then re-enable it for one table within that schema:

```
CALL my_ducklake.set_option('auto_compact', false, schema => 'my_schema');
CALL my_ducklake.set_option('auto_compact', true, schema => 'my_schema', table_name => 'important_
table');
```

With the above settings, calling `ducklake_merge_adjacent_files('my_ducklake')` will compact only `my_schema.important_table` and skip all other tables in `my_schema`. Tables in other schemas are unaffected and will still be compacted by default.

Advanced Options

The `merge_adjacent_files` function supports optional parameters to filter which files are considered for compaction and control memory usage. This enables advanced compaction strategies and more granular control over the compaction process.

- **`max_compacted_files`:** Limits the maximum number of compaction operations produced in a single call, *per table*. Compacting data files can be a very memory intensive operation, so you may consider performing this operation in batches by specifying this parameter. Note that the number of actual compacted files is highly dependent on the `target_file_size` setting.
- **`min_file_size`:** Files smaller than this size (in bytes) are excluded from compaction. If not specified, all files are considered regardless of minimum size.
- **`max_file_size`:** Files at or larger than this size (in bytes) are excluded from compaction. If not specified, it defaults to `target_file_size`. Must be greater than 0.

Example with compacted files limit (applies per table when running across all tables):

```
CALL ducklake_merge_adjacent_files('my_ducklake', 'my_table', max_compacted_files => 10);
```

Example with size filtering:

```
-- Only merge files between 10KB and 100KB
CALL ducklake_merge_adjacent_files('my_ducklake', min_file_size => 10240, max_file_size => 102400);
```

Example: Tiered Compaction Strategy for Streaming Workloads

File size filtering enables tiered compaction strategies, which are particularly useful for real-time/streamed ingestion patterns. A tiered approach merges files in stages:

- **Tier 0 → Tier 1:** Done often, merge small files (< 1MB) into ~5MB files
- **Tier 1 → Tier 2:** Done occasionally, merge medium files (1MB-10MB) into ~32MB files
- **Tier 2 → Tier 3:** Done rarely, merge large files (10MB-64MB) into ~128MB files

This compaction strategy provides more predictable I/O amplification and better incremental compaction for streaming workloads.

Example tiered compaction workflow:

```
-- Tier 0 → Tier 1: merge small files
CALL ducklake_set_option('my_ducklake', 'target_file_size', '5MB');
CALL ducklake_merge_adjacent_files('my_ducklake', max_file_size => 1048576);

-- Tier 1 → Tier 2: merge medium files
CALL ducklake_set_option('my_ducklake', 'target_file_size', '32MB');
CALL ducklake_merge_adjacent_files('my_ducklake', min_file_size => 1048576, max_file_size => 10485760);

-- Tier 2 → Tier 3: merge large files
CALL ducklake_set_option('my_ducklake', 'target_file_size', '128MB');
CALL ducklake_merge_adjacent_files('my_ducklake', min_file_size => 10485760, max_file_size => 67108864);
```

Return Values

`ducklake_merge_adjacent_files` returns one row per output file created, with the following columns:

Column	Type	Description
`schema_name`	VARCHAR	Name of the schema containing the table
`table_name`	VARCHAR	Name of the table
`files_processed`	BIGINT	Number of input files merged into this output file
`files_created`	BIGINT	Always 1 — each row represents one output file created

Because each row corresponds to one output file, `files_created` is always 1. To see the total number of output files produced per table, use a GROUP BY:

```
SELECT schema_name, table_name, sum(files_created) AS total_output_files
FROM ducklake_merge_adjacent_files('my_ducklake')
GROUP BY schema_name, table_name;
```

Sorted Compaction

If a table has a sort order defined, `ducklake_merge_adjacent_files` sorts the merged output by those keys before writing the resulting Parquet file. The sort order applied is the one currently active on the table at the time compaction runs — not the order that was active when the original files were written.

> Calling this function does not immediately delete the old files. See the cleanup old files section on how to trigger a cleanup of these files.

Expire Snapshots

DuckLake in normal operation never removes any data, even when tables are dropped or data is deleted. Due to time travel, the removed data is still accessible.

Data can only be physically removed from DuckLake by expiring snapshots that refer to the old data. This can be done using the `ducklake_expire_snapshots` function.

Usage

The below command expires a snapshot with a specific snapshot id.

```
CALL ducklake_expire_snapshots('my_ducklake', versions => [2]);
```

The below command expires snapshots older than a week.

```
CALL ducklake_expire_snapshots('my_ducklake', older_than => now() - INTERVAL '1 week');
```

The below command performs a *dry run*, which only lists the snapshots that will be deleted, instead of actually deleting them.

```
CALL ducklake_expire_snapshots('my_ducklake', dry_run => true, older_than => now() - INTERVAL '1 week');
```

It is also possible to set a DuckLake option to expire snapshots that applies to the whole catalog.

```
CALL my_ducklake.set_option('expire_older_than', '1 month');
```

Cleaning Up Files

Note that expiring snapshots does not immediately delete files that are no longer referenced. See the cleanup old files section on how to trigger a cleanup of these files.

Cleanup of Files

In DuckLake you may want to delete data files that are either a part of an expired snapshot or are simply not tracked by DuckLake anymore (i.e., orphaned files).

Cleanup of Files from Expired Snapshots

When files are no longer required in DuckLake, due to e.g. snapshots being expired or files being merged, they are not immediately deleted. The reason for this is that there might still be active queries that are scanning these files.

The files are instead added to the `ducklake_files_scheduled_for_deletion` table. These files can then be deleted at a later point. It is generally safe to delete files that have been scheduled for deletion more than a few days ago, provided there are no read transactions that last that long. The files can be deleted using the `ducklake_cleanup_old_files` function.

Usage of the `ducklake_cleanup_old_files` Function

The below command deletes all files scheduled for deletion.

```
CALL ducklake_cleanup_old_files(
    'my_ducklake',
    cleanup_all => true
);
```

The below command deletes all files that were scheduled for deletion more than a week ago.

```
CALL ducklake_cleanup_old_files(
    'my_ducklake',
    older_than => now() - INTERVAL '1 week'
);
```

The below command performs a *dry run*, which only lists the files that will be deleted, instead of actually deleting them.

```
CALL ducklake_cleanup_old_files(
    'my_ducklake',
    dry_run => true,
    older_than => now() - INTERVAL '1 week'
);
```

Cleanup of Orphaned Files

Orphan files are files that are untracked by the DuckLake metadata catalog. They may appear due to, for example, a systems failure. DuckLake provides the `ducklake_delete_orphaned_files` function to delete these files.

Usage of the `ducklake_delete_orphaned_files` Function

The below command deletes all orphaned files.

```
CALL ducklake_delete_orphaned_files(
    'my_ducklake',
    cleanup_all => true
);
```

The below command deletes all files that are older than a specified time.

```
CALL ducklake_delete_orphaned_files(
    'my_ducklake',
    older_than => now() - INTERVAL '1 week'
);
```

The below command performs a *dry run*, which only lists the files that will be deleted, instead of actually deleting them.

```
CALL ducklake_delete_orphaned_files(
    'my_ducklake',
    dry_run => true,
    older_than => now() - INTERVAL '1 week'
);
```

There is also a catalog-level option available.

```
CALL my_ducklake.set_option('delete_older_than', '1 week');
```

Rewrite Heavily Deleted Files

DuckLake uses a merge-on-read strategy when data is deleted from a table. In short, this means that DuckLake uses a delete file which contains a pointer to the deleted records on the original file. This makes deletes very efficient. However, for heavily deleted tables, reading performance will be hindered by this approach. To solve this problem, DuckLake exposes a function called `ducklake_rewrite_data_files` that rewrites files that contain an amount of deletes bigger than a given threshold to a new file that contains non-deleted records. These files can then be further compacted with a `ducklake_merge_adjacent_files` operation. The default value for the delete threshold is 0.95.

Usage

Apply to all tables in a catalog:

```
CALL ducklake_rewrite_data_files('my_ducklake');
```

Apply only to a specific table:

```
CALL ducklake_rewrite_data_files('my_ducklake', 't');
```

Provide a specific value for the delete threshold:

```
CALL ducklake_rewrite_data_files('my_ducklake', 't', delete_threshold => 0.5);
```

Set a specific threshold for the whole catalog:

```
CALL my_ducklake.set_option('rewrite_delete_threshold', 0.5);
```

Set a specific threshold for a schema:

```
CALL my_ducklake.set_option('rewrite_delete_threshold', 0.5, schema => 'my_schema');
```

Set a specific threshold for a table:

```
CALL my_ducklake.set_option('rewrite_delete_threshold', 0.5, table_name => 'my_table');
```

Disable automatic compaction for a specific table:

```
CALL my_ducklake.set_option('auto_compact', false, table_name => 'my_table');
```

Return Values

`ducklake_rewrite_data_files` returns one row per output file created, with the following columns:

Column	Type	Description
`schema_name`	`VARCHAR`	Name of the schema containing the table
`table_name`	`VARCHAR`	Name of the table
`files_processed`	`BIGINT`	Number of input files rewritten into this output file
`files_created`	`BIGINT`	Always 1 — each row represents one output file created

Because each row corresponds to one output file, `files_created` is always `1`. To see the total number of output files produced per table, use a `GROUP BY`:

```
SELECT schema_name, table_name, sum(files_created) AS total_output_files
FROM ducklake_rewrite_data_files('my_ducklake')
GROUP BY schema_name, table_name;
```

Checkpoint

DuckLake provides the option to implement all the maintenance functions bundled in the `CHECKPOINT` statement. This statement will run in order the following DuckLake functions:

- `ducklake_flush_inlined_data`
- `ducklake_expire_snapshots`
- `ducklake_merge_adjacent_files`
- `ducklake_rewrite_data_files`
- `ducklake_cleanup_old_files`
- `ducklake_delete_orphaned_files`

Usage

The `CHECKPOINT` statement takes the following global DuckLake options:

- `rewrite_delete_threshold`: A threshold that determines the minimum amount of data that must be removed from a file before a rewrite is warranted (0...1). Used by `ducklake_rewrite_data_files`. Can be scoped globally, per schema, or per table.
- `delete_older_than`: How old unused files must be to be removed by the `ducklake_delete_orphaned_files` and `ducklake_cleanup_old_files` cleanup functions.
- `expire_older_than`: How old snapshots must be, by default, to be expired by `ducklake_expire_snapshots`.

- `auto_compact`: Whether the compaction functions `ducklake_flush_inlined_data`, `ducklake_merge_adjacent_files`, `ducklake_rewrite_data_files` and `ducklake_delete_orphaned_files` run on a given table. Defaults to `true`. Can be scoped globally, per schema, or per table.

If these options are not provided via the `ducklake.set_option` function, `CHECKPOINT` will use the default values when applicable and will run a `CHECKPOINT` of the whole DuckLake.

```
CHECKPOINT;
```

Advanced Features

Constraints

DuckLake has limited support for constraints. The only constraint type that is currently supported is `NOT NULL`. It does not support `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE` or `CHECK` constraints.

Examples

Define a column as not accepting `NULL` values using the `NOT NULL` constraint.

```
CREATE TABLE tbl (col INTEGER NOT NULL);
```

Add a `NOT NULL` constraint to an existing column of an existing table.

```
ALTER TABLE tbl ALTER col SET NOT NULL;
```

Drop a `NOT NULL` constraint from a table.

```
ALTER TABLE tbl ALTER col DROP NOT NULL;
```

Conflict Resolution

In DuckLake, snapshot identifiers are written in a sequential order. The first snapshot has `snapshot_id` `0`, and subsequent snapshot ids are monotonically increasing such that the second snapshot has id `1`, etc. The sequential nature of snapshot identifiers is used to **detect conflicts** between snapshots. The `ducklake_snapshot` table has a `PRIMARY KEY` constraint defined over the `snapshot_id` column.

When two connections try to write to a `ducklake` table, they will try to write a snapshot with the same identifier and one of the transactions will trigger a `PRIMARY KEY` constraint violation and fail to commit. When such a conflict occurs – we try to resolve the conflict. In many cases, such as when both transactions are inserting data into a table, we can retry the commit without having to rewrite any actual files. During conflict resolution, we query the `ducklake_snapshot_changes` table to figure out the high-level set of changes that other snapshots have made in the meantime.

- If there are no logical conflicts between the changes that the snapshots have made – we automatically retry the transaction in the metadata catalog without rewriting any data files.
- If there are logical conflicts, we abort the transaction and instruct the user that conflicting changes have been made.

Logical Conflicts

Below is a list of logical conflicts based on the snapshot's changeset. Snapshots conflict when any of the following conflicts occur:

Schemas

- Both snapshots create a schema with the same name
- Both snapshots drop the same schema
- A snapshot tries to drop a schema in which another transaction created an entry

Tables / Views

- Both snapshots create a table or view with the same name in the same schema
- A snapshot tries to create a table or view in a schema that was dropped by another snapshot
- Both snapshots drop the same table or view
- A snapshot tries to alter a table or view that was dropped or altered by another snapshot

Data

- A snapshot tries to insert data into a table that was dropped or altered by another snapshot
- A snapshot tries to delete data from a table that was dropped, altered, deleted from or compacted by another snapshot

Compaction

- A snapshot tries to compact a table that was deleted from by another snapshot
- A snapshot tries to compact a table that was dropped by another snapshot

Data Change Feed

In addition to allowing you to query the state of the database at any point in time, DuckLake allows you to query the *changes that were made between any two snapshots*. This can be done using the `table_changes` function.

Examples

Consider the following DuckLake instance:

```
ATTACH 'ducklake:changes.duckdb' AS db (DATA_PATH 'change_files/');
-- Snapshot 1
CREATE TABLE db.tbl (id INTEGER, val VARCHAR);
-- Snapshot 2
INSERT INTO db.tbl VALUES (1, 'Hello'), (2, 'DuckLake');
-- Snapshot 3
DELETE FROM db.tbl WHERE id = 1;
-- Snapshot 4
UPDATE db.tbl SET val = concat(val, val, val);
```

Changes Made by a Specific Snapshot

```
FROM db.table_changes('tbl', 2, 2);
```

snapshot_id	rowid	change_type	id	val
2	0	insert	1	Hello
2	1	insert	2	DuckLake

Changes Made between Multiple Snapshots

```
FROM db.table_changes('tbl', 3, 4)
ORDER BY snapshot_id;
```

snapshot_id	rowid	change_type	id	val
3	0	delete	1	Hello
4	1	update_preimage	2	DuckLake
4	1	update_postimage	2	DuckLakeDuckLakeDuckLake

Changes Made in the Last Week

```
FROM changes.table_changes('tbl', now() - INTERVAL '1 week', now());
```

`table_changes`

The `table_changes` function takes as input the table for which changes should be returned, and two bounds: the start snapshot and the end snapshot (inclusive). The bounds can be given either as a snapshot id, or as a timestamp.

The result of the function is the set of changes, read using the table schema as of the end snapshot provided, and three extra columns: `snapshot_id`, `rowid` and `change_type`.

Column	Description
`snapshot_id`	The snapshot which made the change
`rowid`	The row identifier of the row which was changed
`change_type`	`insert`, `update_preimage`, `update_postimage` or `delete`

Updates are split into two rows: the `update_preimage` and `update_postimage`. `update_preimage` is the row as it was prior to the update operation. `update_postimage` is the row as it is after the update operation.

When the schema of a table is altered, changes are read as of the schema of the table as of the end snapshot. As such, if a column is dropped in between the provided bounds, the dropped column is omitted from the entire result. If a column is added, any changes made to the table prior to the addition of the column will have the column substituted with its default value.

`table_deletions`

The `table_deletions` function returns only the rows that were *deleted* between two snapshots. It has the same signature as `table_changes`: it takes a table name and two bounds (start and end snapshot, inclusive), which can be given as snapshot ids or timestamps.

```
FROM db.table_deletions('tbl', 3, 3);
```

snapshot_id	rowid	id	val
3	0	1	Hello

The result contains `snapshot_id` and `rowid` columns followed by the data columns of the table. There is no `change_type` column because all rows returned are deletions.

`table_deletions` is an alias for the `ducklake_table_deletions` function.

`table_insertions`

The `table_insertions` function returns only the rows that were *inserted* between two snapshots. It has the same signature as `table_changes`.

```
FROM db.table_insertions('tbl', 2, 2);
```

snapshot_id	rowid	id	val
2	0	1	Hello
2	1	2	DuckLake

`table_insertions` is an alias for the `ducklake_table_insertions` function.

Compaction

Compaction operations that expire snapshots can limit the change feed that can be read. For example, if deleted rows are removed as part of compaction, these cannot be returned by the change feed anymore.

Data Inlining

When writing small changes to DuckLake, it can be wasteful to write each changeset to an individual Parquet file. DuckLake supports directly writing small changes to the metadata using *data inlining*. Instead of writing a Parquet file to the data storage and then writing a reference to that file in the metadata catalog, DuckLake writes the data directly to tables within the metadata catalog.

Data inlining applies to both inserts and deletes:

- **Insertion inlining** – small inserts are written to a per-table inlined data table instead of a new Parquet data file.
- **Deletion inlining** – small deletes from existing Parquet data files are written to a per-table inlined deletion table instead of a new Parquet delete file.

Inlined data behaves exactly the same as data written to Parquet files. The only difference is that it lives in the metadata catalog rather than in Parquet files in the data path.

Configuring Data Inlining

Data inlining is enabled by default with a row limit of 10. Any inserts or deletes that affect fewer rows than `data_inlining_row_limit` are automatically written to inlined tables instead of Parquet files.

Global Default

The global default row limit is controlled by the `ducklake_default_data_inlining_row_limit` DuckDB setting. This applies to all DuckLake connections that do not have an explicit `data_inlining_row_limit` configured:

```
-- Change the default row limit to 50 rows
SET ducklake_default_data_inlining_row_limit = 50;

-- Disable data inlining by default
SET ducklake_default_data_inlining_row_limit = 0;
```

Per-Connection Override

The `DATA_INLINING_ROW_LIMIT` parameter of the `ATTACH` statement overrides the default for a single connection. This setting is not persisted and must be specified on each attach:

```
ATTACH 'ducklake:inlining.duckdb' AS my_ducklake (DATA_INLINING_ROW_LIMIT 10);
```

Persistent Override

The `data_inlining_row_limit` option can be persisted in the DuckLake metadata at the table, schema or global level. A persisted value takes priority over both the global DuckDB default and the ATTACH parameter:

```
ATTACH 'ducklake:inlining.duckdb' AS my_ducklake;
USE my_ducklake;
CREATE TABLE t (a INT);
CALL my_ducklake.set_option('data_inlining_row_limit', 10, table_name => 't');
```

Insertion Inlining

When insertion inlining is enabled, small inserts are stored directly in the metadata catalog instead of creating a new Parquet data file. DuckLake automatically creates and manages a per-table inlined data table with the following structure:

```
-- created and managed internally by DuckLake; one table per schema version
ducklake_inlined_data_<table_id>_<schema_version> (
    row_id         BIGINT,
    begin_snapshot BIGINT,
    end_snapshot   BIGINT,
    -- one column per table column, matching the table schema
    ...
)
```

`begin_snapshot` is the snapshot in which the row was inserted. `end_snapshot` is the snapshot in which the row was deleted, or `NULL` if the row is still live. Deletions of inlined rows are recorded by setting `end_snapshot` rather than creating a separate inlined deletion entry. A new inlined data table is created each time the table schema changes (e.g., a column is added or dropped), so the column layout always matches the current schema.

For example, when inserting a low number of rows, data is automatically inlined:

```
ATTACH 'ducklake:inlining.duckdb' AS my_ducklake (DATA_INLINING_ROW_LIMIT 10);
USE my_ducklake;

CREATE TABLE tbl (col INTEGER);
-- Inserting 3 rows - below the threshold, data is inlined
INSERT INTO tbl VALUES (1), (2), (3);
-- No Parquet files exist
SELECT count(*) FROM glob('inlining.duckdb.files/**');
```

count_star()
int64
0

When inserting more rows than the `DATA_INLINING_ROW_LIMIT`, inserts are automatically written to Parquet:

```
INSERT INTO tbl FROM range(100);
SELECT count(*) FROM glob('inlining.duckdb.files/**');
```

count_star()
int64
1

Deletion Inlining

When deletion inlining is enabled, deletes from existing Parquet data files that affect fewer rows than `DATA_INLINING_ROW_LIMIT` are stored directly in the metadata catalog instead of creating a Parquet delete file. DuckLake automatically creates and manages a per-table inlined deletion table with the following structure:

```
-- created and managed internally by DuckLake; one table per DuckLake table
ducklake_inlined_delete_<table_id> (
    file_id        BIGINT,
    row_id         BIGINT,
    begin_snapshot BIGINT
)
```

`file_id` is the ID of the Parquet data file containing the deleted row, `row_id` is the position of the deleted row within that file, and `begin_snapshot` is the snapshot in which the deletion occurred.

> **Note** Deletion inlining only applies to rows in existing Parquet data files. Deletes that target inlined insert rows are handled by setting `end_snapshot` on the inlined insert row and do not create an inlined deletion entry.

For example, with deletion inlining enabled, a small delete produces no new Parquet delete file:

```
ATTACH 'ducklake:inlining.duckdb' AS my_ducklake (DATA_INLINING_ROW_LIMIT 10);
USE my_ducklake;

CREATE TABLE tbl (col INTEGER);
-- Insert enough rows to exceed the threshold, so they go to Parquet
INSERT INTO tbl FROM range(100);

-- Delete 2 rows - below the threshold, stored inline
DELETE FROM tbl WHERE col < 2;
-- No new delete file is created
SELECT count(*) FROM glob('inlining.duckdb.files/**/*.parquet') WHERE file LIKE '%-delete.parquet';
```

ALTER TABLE Support

The following `ALTER TABLE` operations are supported within a transaction that also contains inlined data:

- `ADD COLUMN`
- `DROP COLUMN`
- `RENAME TABLE`
- `RENAME COLUMN`
- `ALTER COLUMN TYPE`
- `SET NOT NULL` / `DROP NOT NULL`

Metadata Catalog Support

Data inlining is supported when using DuckDB, PostgreSQL or SQLite as the metadata catalog.

When using a non-DuckDB metadata catalog, nested types (`STRUCT`, `MAP` and `LIST`) are stored as `VARCHAR` strings in the inlined data table. DuckLake automatically casts the values back to the correct type when reading.

Inlining of `VARIANT` columns is only supported when using DuckDB as the metadata catalog. PostgreSQL and SQLite cannot inline `VARIANT` values because the type does not round-trip through a string representation without losing type information. Tables with `VARIANT` columns will not have their data inlined when using a non-DuckDB metadata catalog.

Flushing Inlined Data

Inlined data — both inlined inserts and inlined deletions — can be manually flushed to Parquet files by calling the `ducklake_flush_inlined_data` function.

Flush all inlined data in all schemas and tables:

```
CALL ducklake_flush_inlined_data('my_ducklake');
```

Flush inlined data only within a specific schema:

```
CALL ducklake_flush_inlined_data(
    'my_ducklake',
    schema_name => 'my_schema'
);
```

Flush inlined data for a specific table in the default `main` schema:

```
CALL ducklake_flush_inlined_data(
    'my_ducklake',
    table_name => 'my_table'
);
```

Flush inlined data for a specific table in a specific schema:

```
CALL ducklake_flush_inlined_data(
    'my_ducklake',
    schema_name => 'my_schema',
    table_name => 'my_table'
);
```

Return Values

`ducklake_flush_inlined_data` returns one row per table that had data flushed, with the following columns:

Column	Type	Description
`schema_name`	`VARCHAR`	Name of the schema containing the table
`table_name`	`VARCHAR`	Name of the table
`rows_flushed`	`BIGINT`	Number of rows flushed from inlined storage to Parquet

Tables with no inlined data are not included in the result. Example:

```
SELECT schema_name, table_name, rows_flushed
FROM ducklake_flush_inlined_data('my_ducklake');
```

Time Travel and Deletions

When flushing inlined inserts that have had rows deleted, DuckLake creates both the materialized Parquet data file and a *partial deletion file*. Rather than creating one deletion file per delete snapshot, the partial deletion file consolidates all deletions into a single Parquet file with an extra column that records the snapshot in which each row was deleted. This preserves full time-travel support while keeping the number of files minimal.

When flushing inlined deletions (rows deleted from existing Parquet data files), DuckLake writes a Parquet delete file for each affected data file. If a delete file already exists for that data file, the inlined deletions are merged into it. The resulting file is always a partial deletion file so that snapshot information is preserved for time-travel queries.

For example:

```
ATTACH 'ducklake:inlining.duckdb' AS my_ducklake (DATA_PATH 'data/', DATA_INLINING_ROW_LIMIT 10);
USE my_ducklake;

CREATE TABLE t1 (a INTEGER);
INSERT INTO t1 VALUES (1), (2), (3), (4), (5), (6), (7), (8);

DELETE FROM t1 WHERE a = 2;
DELETE FROM t1 WHERE a = 5;

-- Flush materializes data to Parquet and creates a deletion file with snapshot information
CALL ducklake_flush_inlined_data('ducklake');
```

After flushing, time travel to snapshots before the deletions still returns the deleted rows:

```
-- Returns all 8 original rows
SELECT * FROM t1 AT (VERSION => 1);
```

Sorted Flush

If a table has a sort order defined, `ducklake_flush_inlined_data` sorts the output by those keys before writing the resulting Parquet file. The sort order applied is the one currently active on the table at the time of the flush.

Interaction with `auto_compact`

If a table has `auto_compact` set to `false`, `ducklake_flush_inlined_data` will skip it when flushing the whole lake or a whole schema. The same applies when flushing via `CHECKPOINT`, since `CHECKPOINT` calls `ducklake_flush_inlined_data` internally.

> **Note** `auto_compact` does not cause flushing to happen automatically after writes. It only determines whether a table is eligible for flushing when a maintenance function is called.

Encryption

DuckLake supports an encrypted mode. In this mode, all files that are written to the data directory are encrypted using Parquet encryption. In order to use this mode, the ENCRYPTED flag must be passed when initializing the DuckLake catalog:

```
ATTACH 'ducklake:encrypted.ducklake'
    (DATA_PATH 'untrusted_location/', ENCRYPTED);
```

When enabled, all Parquet files that are written as part of DuckLake operations are automatically encrypted. The encryption keys for each file are automatically generated by the system when the files are written. New encryption keys are automatically generated for each write operation – such that each file is encrypted using their own encryption key. The generated keys are stored in the catalog, in the `encryption_key` field of the `ducklake_data_file` table.

When data is read from the encrypted files, the keys are read from the catalog server and automatically used to decrypt the files. This allows encrypted DuckLake databases to be interacted with in exactly the same manner as unencrypted databases.

Partitioning

DuckLake tables can be partitioned by a user-defined set of partition keys. When a DuckLake table has partitioning keys defined, any new data is split up into separate data files along the partitioning keys. During query planning, the partitioning keys are used to prune which files are scanned.

The partitioning keys defined on a table only affect new data written to the table. Previously written data will be kept partitioned by the keys the table had when that data was written. This allows the partition layout of a table to evolve over-time as needed.

The partitioning keys for a file are stored in DuckLake. These keys do not need to be necessarily stored within the files, or in the paths to the files.

Examples

> By default, DuckLake uses Hive partitioning. If you want to avoid this style of partitions, you can opt out globally via `CALL my_ducklake.set_option('hive_file_pattern', false)`, or per schema/table by passing the `schema` or `table_name` argument.

Set the partitioning keys of a table, such that new data added to the table is partitioned by these keys.

To partition on a column, use:

```
ALTER TABLE tbl SET PARTITIONED BY (part_key);
```

You can also partition using functions. For example, to partition based on the year/month of a timestamp, use:

```
ALTER TABLE tbl SET PARTITIONED BY (year(ts), month(ts));
```

To distribute rows into a fixed number of buckets using an Iceberg-compatible hash (Murmur3), use the `bucket` transform:

```
ALTER TABLE tbl SET PARTITIONED BY (bucket(8, user_id));
```

Bucket partitioning can be combined with other transforms:

```
ALTER TABLE tbl SET PARTITIONED BY (bucket(8, user_id), month(ts));
```

Remove the partitioning keys of a table, such that new data added to the table is no longer partitioned.

```
ALTER TABLE tbl RESET PARTITIONED BY;
```

DuckLake supports the following partition clauses:

Transform	Expression
identity	col_name
bucket	bucket(N, col_name)
year	year(ts)
month	month(ts)
day	day(ts)
hour	hour(ts)

Transactions

DuckLake has full support for ACID and offers snapshot isolation for all interactions with the database. All operations, including DDL statements such as `CREATE TABLE` or `ALTER TABLE`, have full transactional support. Transactions have all-or-nothing semantics and can be composed of multiple changes that are made to the database.

The extension also provides some syntax to be able to manage transactions. This is explained in the DuckDB documentation. Basically it comes down to this:

```
BEGIN TRANSACTION;
-- Some operation
-- Some other operation
COMMIT;
-- Or
ROLLBACK; -- ABORT will have the same behavior
```

In the context of DuckLake, one committed transaction (i.e., a `BEGIN-COMMIT` block) represents one snapshot.

If multiple transactions are being performed concurrently in one table, the `ducklake` extension has some default configurations for a retry mechanism. This default configurations can be overridden.

Row Lineage

Every row created in DuckLake has a unique row identifier, which can be queried as the `rowid` virtual column. This identifier is assigned when a row is first inserted into the system. The identifier is preserved when the row is moved between files – for example as part of `UPDATE` and compaction operations.

The `rowid` column can be used to track whether the addition of files actually introduces new rows into DuckLake, or whether the operation is simply moving files around. This is used internally in the data change feed to differentiate between update operations and delete + insert operations.

Macros

DuckLake supports both scalar and table macros. Macros can be created using the standard `CREATE MACRO` syntax.

Macros are stored in the DuckLake metadata across three catalog tables: `ducklake_macro`, `ducklake_macro_impl`, and `ducklake_macro_parameters`.

Macros support time travel, so attaching at a previous snapshot will reflect the macros that existed at that point in time.

Scalar Macros

Scalar macros return a single value.

```
CREATE MACRO add_values(a, b) AS a + b;
SELECT add_values(40, 2);
```

Table Macros

Table macros return a table.

```
CREATE MACRO filtered_table(threshold) AS TABLE
    SELECT *
    FROM my_table
    WHERE value > threshold;
SELECT * FROM filtered_table(100);
```

Default Parameters

Macros can define default values for parameters.

```
CREATE MACRO add_with_default(a, b := 10) AS a + b;
SELECT add_with_default(5);
```

Typed Parameters

Macros can specify types for their parameters.

```
CREATE MACRO typed_add(a INTEGER, b INTEGER) AS a + b;
```

Dropping Macros

Macros can be dropped using `DROP MACRO` or `DROP MACRO TABLE` for table macros.

```
DROP MACRO add_values;
DROP MACRO TABLE filtered_table;
```

Views

Views can be created using the standard `CREATE VIEW` syntax. The views are stored in the metadata, in the `ducklake_view` table.

Examples

Create a view.

```
CREATE VIEW v1 AS SELECT * FROM tbl;
```

Comments

Comments can be added to tables, views and columns using the `COMMENT ON` syntax. The comments are stored in the metadata, and can be modified in a transactional manner:

- For tables and views, comments are in the `ducklake_tag` table.
- For columns, comments are in the `ducklake_column_tag` table.

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';
```

Sorted Tables

DuckLake tables can be configured with a sort order. When a sort order is defined, data is physically sorted by the specified columns whenever it is written out as Parquet — during `INSERT`, file compaction and inlined data flushing.

Sorting data before writing improves the effectiveness of min/max statistics at query time, which allows the DuckDB query engine to skip data files whose value ranges do not overlap with a query's filter predicates.

Example Setup

The examples on this page use the following DuckLake instance:

```
ATTACH 'ducklake:sorted.duckdb' AS my_ducklake (DATA_PATH 'data/');
USE my_ducklake;
CREATE TABLE events (event_time TIMESTAMP, event_type VARCHAR, value DOUBLE);
```

Setting a Sort Order

Set the sort order for a table using `SET SORTED BY`:

```
ALTER TABLE events SET SORTED BY (event_time ASC);
```

Multiple sort keys are supported:

```
ALTER TABLE events SET SORTED BY (event_time ASC, event_type DESC);
```

ASC and DESC control the sort direction. `NULLS FIRST` and `NULLS LAST` are also supported to control null ordering:

```
ALTER TABLE events SET SORTED BY (event_time ASC NULLS LAST);
```

Expression-Based Sort Keys

Arbitrary expressions are supported in `SET SORTED BY`, not just column references. This includes function calls, casts, and DuckLake macros.

Sort by the hour extracted from a timestamp:

```
ALTER TABLE events SET SORTED BY (date_trunc('hour', event_time) ASC);
```

Sort by a DuckLake macro:

```
CREATE MACRO event_bucket(t) AS date_trunc('day', t);
ALTER TABLE events SET SORTED BY (event_bucket(event_time) ASC);
```

Expressions are validated when `SET SORTED BY` is executed — an error is returned if any referenced columns or functions cannot be resolved.

Removing a Sort Order

To remove the sort order from a table, use `RESET SORTED BY`:

```
ALTER TABLE events RESET SORTED BY;
```

After resetting, subsequent compactions and flushes will write data without sorting.

Effect on Insert

By default, `INSERT` statements automatically sort data according to the table's sort order before writing Parquet files. This behavior is controlled by the `sort_on_insert` option, which defaults to `true`.

To disable sorting on insert (e.g., when insertion speed is the primary concern):

```
CALL my_ducklake.set_option('sort_on_insert', false, table_name => 'events');
```

When `sort_on_insert` is disabled, data written to Parquet during compaction and inlined data flushing is still sorted according to the table's sort order.

Interaction with Data Inlining

When data inlining is enabled and `sort_on_insert` is `false`, data that exceeds the inlining row limit is still sorted before being written to Parquet. Inlined data (which stays in the metadata catalog) is not sorted at insert time — it will be sorted when it is flushed.

Effect on Compaction and Flush

Once a sort order is set, the **current** sort order is applied at the time of compaction or flush — not the sort order that was active when the source data was written.

When `ducklake_merge_adjacent_files` runs on a sorted table, the merged output file is sorted:

```
ALTER TABLE events SET SORTED BY (event_time ASC);
CALL ducklake_merge_adjacent_files('my_ducklake', 'events');
```

The same applies when flushing inlined data:

```
CALL ducklake_flush_inlined_data('my_ducklake', table_name => 'events');
```

Logging

DuckDB provides a logging framework that can be used to debug and monitor DuckLake operations. The DuckLake extension registers a dedicated log type for metadata queries, and DuckDB's built-in `QueryLog` type can be used to trace all SQL queries including those issued internally by the extension.

DuckLake Metadata Query Log

Metadata queries can be an important factor in performance, especially when using a remote catalog (e.g., PostgreSQL). The DuckLake extension registers the `DuckLakeMetadata` log type, which logs every metadata query together with timing information.

Enabling the Log

```
CALL enable_logging('DuckLakeMetadata');
```

Log Fields

`DuckLakeMetadata` is a structured log type. Use `duckdb_logs_parsed` to query the individual fields directly:

Field	Type	Description
`catalog`	`VARCHAR`	The name of the DuckLake catalog that issued the query
`query`	`VARCHAR`	The metadata SQL query that was executed against the catalog database
`elapsed_ms`	`BIGINT`	The time the query took to execute, in milliseconds

Example

```sql
ATTACH 'ducklake:catalog.db' AS my_ducklake (DATA_PATH 'data/');
USE my_ducklake;
CREATE TABLE t1 (a INTEGER, b VARCHAR);

-- Enable DuckLake metadata logging
CALL enable_logging('DuckLakeMetadata');

-- Run an operation that triggers metadata queries
INSERT INTO t1 VALUES (1, 'hello'), (2, 'world');

-- View the metadata queries with structured fields
SELECT catalog, elapsed_ms, query
FROM duckdb_logs_parsed('DuckLakeMetadata');
```

DuckDB Query Log

DuckDB's built-in `QueryLog` type logs every query executed on every connection, including the internal queries that the DuckLake extension issues against the catalog database. This can be useful for general debugging or to see the full picture of what DuckLake does under the hood.

Enabling the Log

```sql
CALL enable_logging('QueryLog');
```

Example

```sql
ATTACH 'ducklake:catalog.db' AS my_ducklake (DATA_PATH 'data/');
USE my_ducklake;
CREATE TABLE t1 (a INTEGER, b VARCHAR);

-- Enable query logging
CALL enable_logging('QueryLog');

-- Run an operation
INSERT INTO t1 VALUES (1, 'hello');

-- View all queries including internal DuckLake metadata queries
SELECT type, message
FROM duckdb_logs
WHERE type = 'QueryLog';
```

Combining Log Types

Both log types can be enabled simultaneously to get DuckLake-specific timing information alongside the full query trace:

```
CALL enable_logging(['DuckLakeMetadata', 'QueryLog']);
```

Logging to Different Storages

By default, logs are stored in an in-memory buffer and queried via the `duckdb_logs` view. DuckDB also supports logging to stdout or to a file:

```
-- Log to stdout
CALL enable_logging('DuckLakeMetadata', storage = 'stdout');

-- Log to a file
CALL enable_logging('DuckLakeMetadata', storage_path = '/tmp/ducklake_logs');
```

For more details on log storages and advanced configuration, see the DuckDB Logging documentation.

Metadata

List Files

The `ducklake_list_files` function can be used to list the data files and corresponding delete files that belong to a given table, optionally for a given snapshot.

Usage

List all files:

```
FROM ducklake_list_files('catalog', 'table_name');
```

Get list of files at a specific snapshot version:

```
FROM ducklake_list_files('catalog', 'table_name', snapshot_version => 2);
```

Get list of files at a specific point in time:

```
FROM ducklake_list_files('catalog', 'table_name', snapshot_time => '2025-06-16 15:24:30');
```

Get list of files of a table in a specific schema:

```
FROM ducklake_list_files('catalog', 'table_name', schema => 'main');
```

Parameter	Description	Default
`catalog`	Name of attached DuckLake catalog	
`table_name`	Name of table to fetch files from	
`schema`	Schema for the table	`main`
`snapshot_version`	If provided, fetch files for a given snapshot id	
`snapshot_time`	If provided, fetch files for a given timestamp	

Result

The function returns the following result set.

column_name	column_type
data_file	VARCHAR
data_file_size_bytes	UBIGINT
data_file_footer_size	UBIGINT
data_file_encryption_key	BLOB
delete_file	VARCHAR
delete_file_size_bytes	UBIGINT

column_name	column_type
delete_file_footer_size	UBIGINT
delete_file_encryption_key	BLOB

- If the file has delete files, the corresponding delete file is returned, otherwise these fields are `NULL`.
- If the database is encrypted, the encryption key must be used to read the file.
- The `footer_size` refers to the Parquet footer size – this is optionally provided.

Adding Files

The `ducklake_add_data_files` function can be used to register existing data files as new files in DuckLake. The files are not copied over – DuckLake is merely made aware of their existence, allowing them to be queried through DuckLake. Adding files in this manner supports regular transactional semantics.

> The ownership of the Parquet file is transferred to DuckLake. As such, compaction operations (such as those triggered through `merge_adjacent_files` or `expire_snapshots` followed by `cleanup_old_files`) can cause the files to be deleted by DuckLake.

Usage

Examples

Add the file `people.parquet` to the `people` table in `my_ducklake`:

```
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet');
```

Target a specific schema rather than the default `main`:

```
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet', schema => 'some_schema');
```

Add the file. Any columns that are present in the table but not in the file will have their default values used when reading:

```
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet', allow_missing => true);
```

Add the file. If the file has extra columns in the table they will be ignored (they will not be queryable through DuckLake):

```
CALL ducklake_add_data_files('my_ducklake', 'people', 'people.parquet', ignore_extra_columns => true);
```

Missing Columns

When adding files to a table, all columns that are present in the table must be present in the Parquet file, otherwise an error is thrown. The `allow_missing` option can be used to add the file anyway – in which case any missing columns will be substituted with the `initial_default` value of the column.

Extra Columns

When adding files to a table, if there are any columns present that are not present in the table, an error is thrown by default. The `ignore_extra_columns` option can be used to add the file anyway – any extra columns will be ignored and unreachable.

Type Mapping

In general, types of columns in the source Parquet file must match the type as defined in the table, otherwise an error is thrown. Types in the Parquet file can be narrower than the type defined in the table. Below is a supported mapping type:

Table type	Supported Parquet types
`bool`	`bool`
`int8`	`int8`
`int16`	`int[8/16]`, `uint8`
`int32`	`int[8/16/32]`, `uint[8/16]`
`int64`	`int[8/16/32/64]`, `uint[8/16/32]`
`uint8`	`uint8`
`uint16`	`uint[8/16]`
`uint32`	`uint[8/16/32]`
`uint64`	`uint[8/16/32/64]`
`float`	`float`
`double`	`float/double`
`decimal(P, S)`	`decimal(P',S')`, where `P' <= P` and `S' <= S`
`blob`	`blob`
`varchar`	`varchar`
`date`	`date`
`time`	`time`
`timestamp`	`timestamp`, `timestamp_ns`
`timestamp_ns`	`timestamp`, `timestamp_ns`
`timestamptz`	`timestamptz`

Migrations

DuckDB to DuckLake

Migrating from DuckDB to DuckLake is very simple to do with the DuckDB `ducklake` extension. However, if you are currently using some DuckDB features that are unsupported in DuckLake, this guide will definitely help you.

First Scenario: Everything Is Supported

If you are not using any of the unsupported features, migrating from DuckDB to DuckLake will be as simple as running the following commands:

```
ATTACH 'ducklake:my_ducklake.ducklake' AS my_ducklake;
ATTACH 'db.duckdb' AS my_duckdb;

COPY FROM DATABASE my_duckdb TO my_ducklake;
```

Note that it doesn't matter what catalog you are using as a metadata backend for DuckLake.

Second Scenario: Not Everything Is Supported

If you have been using DuckDB for a while, there is a chance you are using some very specific types, macros, default values that are not literals or even things like generated columns. If this is your case, then migrating will have some tradeoffs.

- Specific types need to be cast to a supported DuckLake type. User defined types that are created as a `STRUCT` can be interpreted as such and `ENUM` and `UNION` will be cast to `VARCHAR` and `VARINT` will be cast to `INT`.
- Macros can be migrated to a DuckDB persisted database. If you are using DuckDB as your catalog for DuckLake, then this will be the destination. If you are using other catalogs like PostgreSQL, SQLite or MySQL, DuckDB macros are not supported and therefore can't be migrated.
- Generated columns are the same as defaults that are not literals and therefore they need to be specified when inserting the data into the destination table. This means that the values will always be persisted (no `VIRTUAL` option).

Migration Script

The following Python script can be used to migrate from a DuckDB persisted database to DuckLake bypassing the unsupported features.

> Currently, only local migrations are supported by this script. The script will be adapted in the future to account for migrations to remote object storage such as S3 or GCS.

The script can be run in any Python environment with DuckDB installed. The usage is the following:

```
usage: migration.py [-h]
    --duckdb-catalog DUCKDB_CATALOG
    --duckdb-file DUCKDB_FILE
    --ducklake-catalog DUCKLAKE_CATALOG
    --catalog-type {duckdb,postgresql,sqlite}
    [--ducklake-file DUCKLAKE_FILE]
    --ducklake-data-path DUCKLAKE_DATA_PATH
```

If you are migrating to PostgreSQL, make sure that you provide the following environment variables for the PostgreSQL secret connection:

- `POSTGRES_HOST`
- `POSTGRES_PORT`
- `POSTGRES_DB`
- `POSTGRES_USER`
- `POSTGRES_PASSWORD`

Guides

Guides

The guides section contains compact how-to guides that are focused on achieving a single goal.

Access Control

While access control is not per se a feature of DuckLake, we can leverage the tools that DuckLake uses and their permission systems to implement schema- and table-level permissions in DuckLake.

Basic Principles

In this guide, we focus on three different roles regarding access control in DuckLake:

- The **DuckLake Superuser** can perform any DuckLake operation, most notably:
  - Initializing DuckLake (done the first time we run the `ATTACH` command).
  - Creating schemas.
  - `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT` on any DuckLake table.
- The **DuckLake Writer** can perform the following operations:
  - `ATTACH` to an existing DuckLake.
  - `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT` on any or a subset of DuckLake tables.
  - Optionally, `SELECT` on any or a subset of DuckLake tables.
  - Optionally, `CREATE` schema.
- The **DuckLake Reader** can perform the following operations:
  - `ATTACH` to an existing DuckLake. Both `READ_ONLY` and regular attaching modes will work.
  - `SELECT` on any or a subset of DuckLake tables.

> These roles are not actually implemented in DuckLake; they are constructs used in this guide, as they represent the most common types of roles present in data management systems.

DuckLake has two components: the metadata catalog, which resides in a SQL database, and the storage, which can be any filesystem backend. The roles mentioned above require different specific permissions at the **catalog level:**

- The DuckLake Superuser needs all permissions under the specified schema (`public` by default). Since this user initializes all tables, they also become the owner. Subsequent migrations between different version of the DuckLake specification must be carried out by this user.
- The DuckLake Writer only needs permissions to `INSERT`, `UPDATE`, `DELETE`, and `SELECT` at the catalog level. This is sufficient for any operation in DuckLake, including operations that expire snapshots.
- The DuckLake Reader only needs `SELECT` permissions at the catalog level.

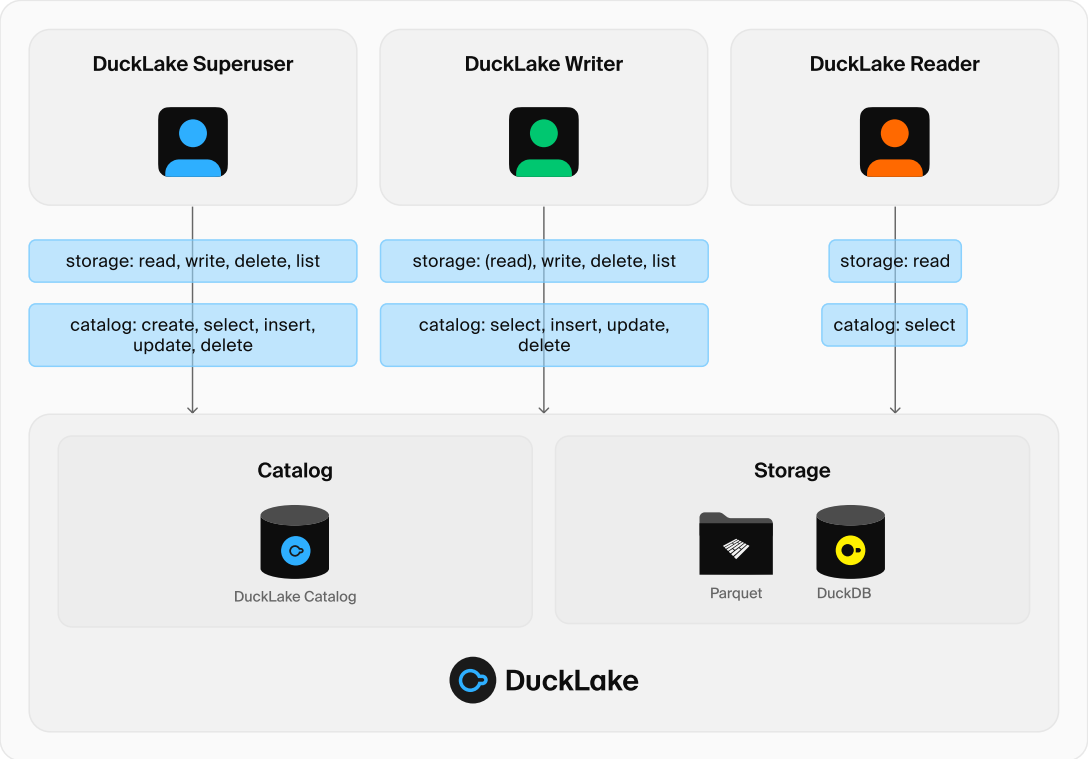
At the storage level, we can leverage the way DuckLake structures data paths for different tables, which uses the following convention:

```
/<schema>/<table>/<partition>/<data_file>.parquet
```

Using this convention and the policy mechanisms of certain filesystems (such as cloud-based object storage), we can establish access to certain paths at the schema, table, or even partition level.

> This will not work if we use `ducklake_add_data_files` and the added files do not follow the path convention; permissions at the path level will not apply to these files.

The following diagram shows how these roles and their necessary permissions work in DuckLake:



Access Control with S3 and PostgreSQL

The following is an example implementation of the basic principles described above, focusing on PostgreSQL as a DuckLake catalog and S3 as the storage backend.

PostgreSQL Requirements

In this section, we create the three roles described above in PostgreSQL. We create them as users for simplicity, but you may also create them as groups if you expect a specific role to be used by multiple users.

```
-- Setup initialization user, migrations, and writing, assuming the database is already created
CREATE USER ducklake_superuser WITH PASSWORD 'simple';
GRANT CREATE ON DATABASE access_control TO ducklake_superuser;
GRANT CREATE, USAGE ON SCHEMA public TO ducklake_superuser;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO ducklake_superuser;

-- Writer/reader
CREATE USER ducklake_writer WITH PASSWORD 'simple';
GRANT USAGE ON SCHEMA public TO ducklake_writer;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO ducklake_writer;

-- Reader only
CREATE USER ducklake_reader WITH PASSWORD 'simple';
GRANT USAGE ON SCHEMA public TO ducklake_reader;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO ducklake_reader;
```

S3 Requirements

In AWS, we create three users. The writer user will only have access to a specific schema, and the reader will only have access to a specific table. The policies needed for these users are as follows:

DuckLake Test

In this section, we connect to DuckLake using these different roles to demonstrate how the implementation works in practice using the DuckLake extension of DuckDB.

Let's initialize DuckLake and perform some basic operations with the **DuckLake Superuser**.

```
-- Using the credentials for the AWS DuckLake Superuser (other providers such as STS or SSO can also be
used)
CREATE OR REPLACE SECRET s3_ducklake_superuser (
  TYPE s3,
  PROVIDER config,
  KEY_ID '<key>',
  SECRET '<secret>',
  REGION 'eu-north-1'
);

-- Using the DuckLake Superuser credentials for Postgres
CREATE OR REPLACE SECRET postgres_secret_superuser (
  TYPE postgres,
  HOST 'localhost',
  DATABASE 'access_control',
  USER 'ducklake_superuser',
  PASSWORD 'simple'
);

-- DuckLake config secret
CREATE OR REPLACE SECRET ducklake_superuser_secret (
  TYPE ducklake,
  METADATA_PATH '',
  DATA_PATH 's3://ducklake-access-control/',
  METADATA_PARAMETERS MAP {'TYPE': 'postgres','SECRET': 'postgres_secret_superuser'}
);

-- This initializes DuckLake
ATTACH 'ducklake:ducklake_superuser_secret' AS ducklake_superuser;
USE ducklake_superuser;

-- Perform operations in DuckLake
CREATE SCHEMA IF NOT EXISTS some_schema;
CREATE TABLE IF NOT EXISTS some_schema.some_table (id INTEGER, name VARCHAR);
INSERT INTO some_schema.some_table VALUES (1, 'test');
```

Now let's use the **DuckLake Writer:**

```
-- Drop this to avoid the extension defaulting to this secret
DROP SECRET s3_ducklake_superuser;

-- Using the DuckLake Writer credentials for Postgres
CREATE OR REPLACE SECRET postgres_secret_writer (
  TYPE postgres,
  HOST 'localhost',
  DATABASE 'access_control',
  USER 'ducklake_writer',
```

```
  PASSWORD 'simple'
);

-- Using the credentials for the AWS DuckLake Writer
CREATE OR REPLACE SECRET s3_ducklake_schema_reader_writer (
  TYPE s3,
  PROVIDER config,
  KEY_ID '<key>',
  SECRET '<secret>',
  REGION 'eu-north-1'
);

-- DuckLake config secret
CREATE OR REPLACE SECRET ducklake_writer_secret (
  TYPE ducklake,
  METADATA_PATH '',
  DATA_PATH 's3://ducklake-access-control/',
  METADATA_PARAMETERS MAP {'TYPE': 'postgres','SECRET': 'postgres_secret_writer'}
);

ATTACH 'ducklake:ducklake_writer_secret' AS ducklake_writer;
USE ducklake_writer;

-- Perform operations
CREATE TABLE IF NOT EXISTS some_schema.another_table (id INTEGER, name VARCHAR);
INSERT INTO some_schema.another_table VALUES (1, 'test'); -- Works
INSERT INTO some_schema.some_table VALUES (2, 'test2'); -- Also works

-- Try to perform an unauthorized operation
CREATE TABLE other_table_in_main (id INTEGER, name VARCHAR); -- This unfortunately works
INSERT INTO other_table_in_main VALUES (1, 'test'); -- This doesn't work
```

In the last example, there are limitations to this approach. We can create an empty table, as this only inserts a new record in the metadata catalog—something the DuckLake Writer is allowed to do. The solution is to wrap table initializations in a transaction to ensure the table can't be created if there is no permission to insert data.

```
BEGIN TRANSACTION;
CREATE TABLE other_table_in_main (id INTEGER, name VARCHAR);
INSERT INTO other_table_in_main VALUES (1, 'test');
COMMIT;
```

This will throw the following error:

```
HTTP Error:
Unable to connect to URL "https://ducklake-access-control.s3.amazonaws.com/main/other_table_in_
main/ducklake-01992ec2-d9f7-745e-88e8-708e659a70be.parquet": 403 (Forbidden).

Authentication Failure - this is usually caused by invalid or missing credentials.
* No credentials are provided.
* See https://duckdb.org/docs/stable/extensions/httpfs/s3api.html
```

> The error message is the generic one used when DuckDB cannot access an object in S3; nothing specific to DuckLake.

The **DuckLake Reader** is the simplest role.

```
DROP SECRET s3_ducklake_schema_reader_writer;
CREATE OR REPLACE SECRET s3_ducklake_table_reader (
  TYPE s3,
  PROVIDER config,
  KEY_ID '<key_id>',
  SECRET '<secret_key>',
```

```
  REGION 'eu-north-1'
);
CREATE OR REPLACE SECRET postgres_secret_reader (
  TYPE postgres,
  HOST 'localhost',
  DATABASE 'access_control',
  USER 'ducklake_reader',
  PASSWORD 'simple'
);
CREATE OR REPLACE SECRET ducklake_reader_secret (
  TYPE ducklake,
  METADATA_PATH '',
  DATA_PATH 's3://ducklake-access-control/',
  METADATA_PARAMETERS MAP {'TYPE': 'postgres','SECRET': 'postgres_secret_reader'}
);
ATTACH 'ducklake:ducklake_reader_secret' AS ducklake_reader;
USE ducklake_reader;

SELECT * FROM some_schema.some_table; -- Works
SELECT * FROM some_schema.another_table; -- Fails
```

The last query will print the following error:

```
HTTP Error:
HTTP GET error on 'https://ducklake-access-control.s3.amazonaws.com/some_schema/another_
table/ducklake-019929c8-c9c9-77d7-91e6-bc3c6dc87605.parquet' (HTTP 403)
```

If we try to create a table, which is just a metadata operation, the error will be different, as it is imposed by a lack of permissions on the PostgreSQL side:

```
CREATE TABLE yet_another_table (a INT);
```

```
TransactionContext Error:
Failed to commit: Failed to commit DuckLake transaction: Failed to write new table to DuckLake: Failed
to prepare COPY "COPY "public"."ducklake_table" FROM STDIN (FORMAT BINARY)": ERROR:  permission denied
for table ducklake_table
```

Backup and Recovery

DuckLake has two components: catalog and storage. The catalog contains all of DuckLake's metadata, while the storage contains all of the data files in Parquet format. The catalog is a database, while the storage layer can be any filesystem backend supported by DuckDB. These two components have different backup strategies, so this document will address them separately.

> In this document, we will focus on disasters caused by human errors or application failures/malfunctions that result in data corruption or loss.

Catalog Backup and Recovery

Backup and recovery strategies depend on the SQL database you are using as a DuckLake catalog.

> Compaction and cleanup jobs should only be done before manual backups. These operations can re-write and remove data files, effectively changing the file layout for a specific snapshot.

DuckDB Catalog

For DuckDB, the best approach is to perform regular backups of the metadata database. If the original database is corrupted, tampered with, or even deleted, you can recover from this backup.

```
-- Backup
ATTACH 'db.duckdb' AS db (READ_ONLY);
ATTACH 'backup.duckdb' AS backup;
COPY FROM DATABASE db TO backup;

-- Recover
ATTACH 'db.duckdb' AS db;
ATTACH 'backup.duckdb' AS backup (READ_ONLY);
COPY FROM DATABASE backup TO db;
ATTACH 'ducklake:db.duckdb' AS my_ducklake;
```

It is very important to note that transactions committed to DuckLake after the metadata backup will not be tracked when recovering. The data from the transactions will exist in the data files, but the backup will point to a previous snapshot. If you are running batch jobs, make sure to always back up after the batch job. If you are regularly micro-batching or streaming data, then schedule periodic jobs to back up your metadata.

> **Tip.** If you want to make a backup with the current timestamp, you need to do this with a specific client. Right now ATTACH does not support functions, only strings. This is how it would look in Python:
>
> ```
> import duckdb
> import datetime
> con = duckdb.connection(f"backup_{datetime.datetime.now().strftime('%Y-%m-%d__%I_%M_%S')}.duckdb")
> ```

SQLite Catalog

For SQLite, the process is exactly the same as with DuckDB and has the same implications.

```
-- Backup
ATTACH 'sqlite:db.duckdb' AS db (READ_ONLY);
ATTACH 'sqlite:backup.duckdb' AS backup;
COPY FROM DATABASE db TO backup;

-- Recover
ATTACH 'sqlite:db.duckdb' AS db;
ATTACH 'sqlite:backup.duckdb' AS backup (READ_ONLY);
COPY FROM DATABASE backup TO db;
ATTACH 'ducklake:sqlite:db.duckdb' AS my_ducklake;
```

PostgreSQL Catalog

For PostgreSQL, there are two main approaches to backup and recovery:

- SQL dump: This approach is similar to the one mentioned for SQLite and DuckDB. This process can happen periodically and can only recover to a particular point in time (i.e., the time of the dump). For DuckLake, this will be a specific snapshot, and transactions after this snapshot will not be recorded.
- Continuous Archiving and Point-in-Time Recovery (PITR): This is a more complex approach but allows recovery to a specific point in time. For DuckLake, this means you can recover to a specific snapshot without losing any transactions.

Note that the SQL dump approach can also be managed by DuckDB using the `postgres` extension. In fact, the backup can be a DuckDB file.

> **Warning.** If your Postgres database has indexes, DuckDB will try to copy those over and fail.

```
-- Backup
ATTACH 'postgres:connection_string' AS db (READ_ONLY);
ATTACH 'duckdb:backup.duckdb' AS backup;
COPY FROM DATABASE db TO backup;
```

```
-- Recover
ATTACH 'postgres:connection_string' AS db;
ATTACH 'duckdb:backup.duckdb' AS backup (READ_ONLY);
COPY FROM DATABASE backup TO db;
ATTACH 'ducklake:postgres:connection_string' AS my_ducklake;
```

> Cloud-hosted PostgreSQL solutions may offer different mechanisms. We encourage you to check what your specific vendor recommends as a strategy for backup and recovery.

Storage Backup and Recovery

Backup and recovery of the data files also depend on the storage you are using. In this document, we will only focus on cloud-based object storage since it is the most common for lakehouse architectures.

S3

In S3, there are three main mechanisms that AWS offers to back up and/or restore data:

- Cross-bucket replication
- S3 backup service
- Enable S3 versioning

Both the S3 backup service and S3 object versioning will restore data files in the same bucket. On the other hand, cross-bucket replication will copy the files to a different bucket, and therefore your DuckLake initialization should change:

```
-- Before
ATTACH 'ducklake:some.duckdb' AS my_ducklake (DATA_PATH 's3://<og-bucket>/');

-- After
ATTACH 'ducklake:some.duckdb' AS my_ducklake (DATA_PATH 's3://<replication-bucket>/');
```

GCS

GCS has similar mechanisms to back up and/or restore data:

- Cross-bucket replication
- Backup and DR service
- Object versioning with soft deletes enabled

Regarding cross-bucket replication, repointing to the new bucket will be necessary.

```
-- Before
ATTACH 'ducklake:some.duckdb' AS my_ducklake (DATA_PATH 'gs://<og-bucket>/');

-- After
ATTACH 'ducklake:some.duckdb' AS my_ducklake (DATA_PATH 'gs://<replication-bucket>/');
```

Public DuckLake on Object Storage

This guide explains how to set up a **public read-only DuckLake** on object storage such as Amazon S3, Backblaze B2, Cloudflare R2, Leafcloud Object Storage, etc. Users can query this DuckLake through HTTPS **without authentication**.

> **Warning.** Please check the pricing models of the providers to understand the costs of serving DuckLakes.

> The setup described here is conceptually similar to Frozen DuckLakes but it is simpler to set up.

Steps

Creating the Bucket

Create a new public bucket:

- Amazon S3
- Backblaze B2
- Cloudflare R2

Make sure that the bucket is accessible through the internet. The exact settings for this vary from platform to platform.

Creating the DuckLake

Create a new DuckLake following the "Using a Remove Data Path" guide using DuckDB as the catalog database and set the data path to the `https://` URL that serves your bucket.

Uploading the DuckLake

Upload the `.ducklake` file and its data directory to the bucket. We recommend using Rclone, which supports all object storage platforms listed above.

```
rclone config
```

Create a new Rclone remote (see the details in the foldouts below). In our examples, we will call this `<your_rclone_remote>`.

To upload your data to the R2 bucket, run:

```
rclone cp -v <your_ducklake_catalog.ducklake> <your_rclone_remote>:<your_bucket_name>/<path>/
rclone sync -v <your_ducklake_directory> <your_rclone_remote>:<your_bucket_name>/<path>/
```

Cloudflare: Setting the CORS Policy for Browser Access

If you are using Cloudflare as your storage and try to query the dataset from another website – such as the online DuckDB shell –, you will get an error due to the CORS (Cross-Origin Resource Sharing) security mechanism:

```
IO Error: Failed to attach DuckLake MetaData "__ducklake_metadata_..." at path + "..."
Cannot open database "..." in read-only mode: database does not exist
```

or

```
Invalid Error: Failed to attach DuckLake MetaData "__ducklake_metadata..." at path + "..."
Opening file '...' failed with error:
NetworkError: Failed to execute 'send' on 'XMLHttpRequest': Failed to load '...'.
```

To allow querying the dataset, add a CORS policy to the Cloudflare configuration of the bucket. How to do this depends on whether you are serving directly from the bucket or through a Cloudflare Worker.

If you are serving through a Cloudflare Worker, edit the code of the Worker following the "CORS header proxy" and add the following to the JavaScript code of your `fetch` function:

```
const allowedOrigins = [
  "https://duckdb.org",
  "https://shell.duckdb.org",
];
```

```
const origin = request.headers.get("Origin");

let corsOrigin = "";

if (allowedOrigins.includes(origin)) {
  corsOrigin = origin;
}
return new Response(object.body, {
  headers: {
    "Content-Type": contentType,
    "Access-Control-Allow-Origin": corsOrigin,
    "Cache-Control": "public, max-age=3600",
  }
});
```

If you are serving directly from the bucket, navigate to its settings and add the following CORS Policy:

```
[
  {
    "AllowedOrigins": [
      "https://duckdb.org",
      "https://shell.duckdb.org"
    ],
    "AllowedMethods": [
      "GET"
    ]
  }
]
```

For the complete Cloudflare guide on CORS policies, see the "Configure CORS" page.

Using a Remote Data Path

This guide shows how to set up and load a DuckLake locally, where the DuckLake will be served on an `https://` endpoint.

> **Tip.** DuckLake currently does not allow you to change the persisted data path in the catalog. This is a known limitation that will be lifted in the future, rendering the future version of this guide almost trivial.

In this guide, we assume that we want to create a DuckLake at `https://blobs.duckdb.org/datalake/tpch-sf1/` to serve a read-only copy of the TPC-H SF1 dataset.

Initializing the DuckLake

We initialize the DataLake using the *remote URL as the data path* and immediately detach from the DuckLake:

```
ATTACH 'tpch-sf1.ducklake' AS tpch_sf1_ducklake (
    TYPE ducklake,
    DATA_PATH 'https://blobs.duckdb.org/datalake/tpch-sf1'
);
DETACH tpch_sf1_ducklake;
```

Generating the Data

Generate the data using the `tpchgen-cli` tool:

```
tpchgen-cli --scale-factor 1 --format parquet
```

Populating the DuckLake

Populating the DuckLake requires the following steps:

1. Attach to the DuckLake with a *local data path* using the `OVERRIDE_DATA_PATH true` flag:

```
ATTACH 'tpch-sf1.ducklake' AS tpch_sf1_ducklake (
    TYPE ducklake,
    DATA_PATH 'tpch-sf1',
    OVERRIDE_DATA_PATH true
);
USE tpch_sf1_ducklake;
```

2. Load the data into the DuckLake:

```
CREATE TABLE customer AS FROM 'customer.parquet';
CREATE TABLE lineitem AS FROM 'lineitem.parquet';
CREATE TABLE nation AS FROM 'nation.parquet';
CREATE TABLE orders AS FROM 'orders.parquet';
CREATE TABLE part AS FROM 'part.parquet';
CREATE TABLE partsupp AS FROM 'partsupp.parquet';
CREATE TABLE region AS FROM 'region.parquet';
CREATE TABLE supplier AS FROM 'supplier.parquet';
```

3. Flush the inlined data:

```
CALL ducklake_flush_inlined_data('tpch_sf1_ducklake');
```

4. Close the DuckDB session with `Ctrl + D` or `.quit`.

Uploading the DuckLake

Now, we have the `tpch-sf1.ducklake` file and the `tpch-sf1/` directory with the Parquet files:

```
tree tpch-sf1

tpch-sf1
└── main
    ├── customer
    │   └── ducklake-019a2726-0362-7eae-8a4d-1404ead2c506.parquet
    ├── lineitem
    │   └── ducklake-019a2726-03ad-7d79-b28a-6ae3f114fbd3.parquet
    ├── nation
    │   └── ducklake-019a2726-055d-7f55-914d-02787bda2eae.parquet
    ├── orders
    │   └── ducklake-019a2726-0579-72e2-be9e-0190b3f8f8af.parquet
    ├── part
    │   └── ducklake-019a2726-05fd-7d30-8051-a9172d75f815.parquet
    ├── partsupp
    │   └── ducklake-019a2726-0645-765b-9456-6ba632d8288b.parquet
    ├── region
    │   └── ducklake-019a2726-06a4-7b3a-9788-8f32006be0ad.parquet
    └── supplier
        └── ducklake-019a2726-06bb-72f4-97b3-131d7186787e.parquet
```

We upload both of these to `https://blobs.duckdb.org/datalake/`. This particular URL is served by Cloudflare and is based on the content of a public Cloudflare R2 bucket – but DuckLake works with any `http(s)://` URL.

Using the DuckLake

You can now attach to the remote DuckLake as follows:

```
ATTACH 'https://blobs.duckdb.org/datalake/tpch-sf1.ducklake' AS tpch_sf1_ducklake (TYPE ducklake);
USE tpch_sf1_ducklake;
```

Now you can use it like any other DuckDB database or DuckLake:

```
SELECT count(*) FROM lineitem;
```

count_star()
int64
6001215 (6.00 million)

Troubleshooting

This guide explains how to troubleshoot certain issues with DuckLake. Besides the issues described on this page, see also the "Unsupported Features" page.

Connecting to an Older DuckLake

If you try to connect to a DuckLake created by an older version of the `ducklake` extension, you get the following error message:

```
Invalid Input Error:
DuckLake catalog version mismatch: catalog version is 0.3, but the extension requires version 1.0.
To automatically migrate, set AUTOMATIC_MIGRATION to TRUE when attaching.
```

To work around this, add the `AUTOMATIC_MIGRATION` flag to the `ATTACH` command:

```
ATTACH '...' (AUTOMATIC_MIGRATION);
```

Connecting to a Read-Only DuckLake

If you try to connect to a read-only DuckLake, you get the following error message:

```
Invalid Input Error:
Failed to migrate DuckLake from v0.3 to v1.0:
Cannot execute statement of type "CREATE" on database "__ducklake_metadata_sf1" which is attached in
read-only mode!
```

It is not possible to connect to a read-only DuckLake with a pre-release version (v0.x). To migrate from such a DuckLake, use an older version of the `ducklake` extension and migrate into an intermediate format. For example, copy the data into a DuckLake that you have write access to, then connect to that DuckLake using the newer `ducklake` extension.

Unsupported Features

This page describes what is supported in DuckDB and DuckLake in relation to DuckDB standalone (i.e., `:memory:` or DuckDB file modes). We can make a distinction between:

- What is **currently** not supported by the DuckLake specification. These are features that are supported by DuckDB when using DuckDB's native database format but will not work with a DuckLake backend since the specification does not support them.
- What is **currently** not supported by the `ducklake` DuckDB extension. These are features that are supported by the DuckLake specification but are not (yet) implemented in the DuckDB extension.

Unsupported by the DuckLake Specification

Within this group, we are going to make a distinction between what is not supported now but is likely to be supported in the future and what is not supported and is unlikely to be supported.

Likely to Be Supported in the Future

The following features are currently not supported in DuckLake but are likely to be supported in future versions:

- User defined types.
- Fixed-size arrays, i.e., ARRAY type
- ENUM type
- CHECK constraints (not to be confused with Primary or Foreign Key constraints)
- Default values that are not literals. See the following example:

```
-- This is allowed
CREATE TABLE t1 (id INTEGER, d DATE DEFAULT '2025-08-08');

-- This is not allowed
CREATE TABLE t1 (id INTEGER, d DATE DEFAULT now());
```

- Dropping dependencies, such as views, when calling DROP `...` CASCADE. Note that this is also a DuckDB limitation.
- Generated columns

Unlikely to Be Supported in the Future

The following features are currently not supported in DuckLake and are unlikely to be supported in future versions:

- Indexes
- Primary key or enforced unique constraints and foreign key constraints are unlikely to be supported as these are prohibitively expensive to enforce in data lake setups. We may consider supporting unenforced primary keys, similar to BigQuery's implementation.
- Upserting is only supported via the `MERGE INTO` syntax since primary keys are not supported in DuckLake.
- Sequences
- `VARINT` type

- `BITSTRING` type
- `UNION` type

Acknowledgments

This document is built with Pandoc using the Eisvogel template. The scripts to build the document are available in the `ducklake-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.