Package 'ducklake'

September 9, 2026

Title Interact with 'DuckLake' from R

Version 0.6.0

Description A 'tidyverse'-friendly interface to 'DuckLake' <https://ducklake.select/>, the 'DuckDB' lakehouse format. Attach versioned data lakes from R and work with them using familiar 'dplyr' verbs, with support for ACID transactions, time travel queries, snapshot audit trails, data inlining, encrypted storage, multiple catalog backends ('DuckDB', 'PostgreSQL', 'SQLite', 'MySQL'), and remote access over the 'Quack' protocol from 'DuckDB'.

License MIT + file LICENSE

URL https://tgerke.github.io/ducklake-r/, https://github.com/tgerke/ducklake-r

BugReports https://github.com/tgerke/ducklake-r/issues

Encoding UTF-8

Depends R (>= 4.1)

Imports cli, DBI, dbplyr (>= 2.5.0), dplyr, duckdb (>= 1.5.1)

Suggests admiral, DiagrammeR, fs, ggplot2, jsonlite, knitr, lubridate, pharmaversesdtm, purrr, rmarkdown, spelling, stringr, testthat (>= 3.0.0), tidyr

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

Language en-US

NeedsCompilation no

Author Travis Gerke [aut, cre], Stefan Linner [ctb], Javier Orraca-Deatcu [ctb]

Maintainer Travis Gerke <travisgerke@gmail.com>

Repository CRAN

Date/Publication 2026-09-09 14:50:02 UTC

Contents

`add_data_files` *Register existing Parquet files with a DuckLake table*

Description

Adds Parquet files that already exist on disk (or object storage) to a DuckLake table without copying or rewriting them. This is the migration path for data that is already in Parquet: the files are recorded in the catalog in place.

Usage

```
add_data_files(
  table_name,
  files,
  schema_name = NULL,
  allow_missing = FALSE,
  ignore_extra_columns = FALSE,
  create = FALSE,
  ducklake_name = NULL
)
```

Arguments

table_name	The table to add the files to. Unless create = TRUE, it must already exist with a schema compatible with the files (see allow_missing and ignore_extra_columns for the permitted mismatches).
files	Character vector of Parquet file paths or URIs.
schema_name	Optional schema containing the table (defaults to the lake's main schema).
allow_missing	If TRUE, files may lack columns that exist in the table; missing columns read as the column's initial default. Default FALSE.
ignore_extra_columns	If TRUE, files may contain columns that the table does not have; the extra columns are inaccessible. Default FALSE.
create	If TRUE, create an empty target table from the registered Parquet schema. The table must not already exist. Default FALSE.
ducklake_name	Optional name of the attached DuckLake catalog. If NULL, the current database is used.

Details

Runs CALL ducklake_add_data_files(...) once per file. The complete vector is atomic: outside an existing transaction the function opens one, so the batch creates one snapshot and any failure rolls back every registration. Inside with_transaction() the registrations join the caller's snapshot.

With create = TRUE, the table schema is read from the complete file list with read_parquet() and created with zero rows before registration. Neither this path nor registration copies the data or materializes it in R.

Ownership of each file transfers to DuckLake: compaction (e.g. merge_adjacent_files()) may later rewrite and delete it, so do not add files that something else still relies on.

Value

Invisibly returns the character vector of files added.

See Also

list_ducklake_files(), create_table()

Other table operations: create_table(), create_view(), drop_view(), ducklake_exec(), get_ducklake_table(), get_metadata_table(), list_ducklake_tables(), replace_table(), show_ducklake_query()

Examples

```
lake_dir <- tempfile("addfiles_lake_")
dir.create(lake_dir)
attach_ducklake("addfiles_lake", lake_path = lake_dir)

# Write a couple of Parquet extracts to register. Cast explicitly: DuckDB
# reads a bare 1.5 as DECIMAL, which will not map onto a DOUBLE column.
extract_dir <- tempfile("extracts_")
```

```
dir.create(extract_dir)
conn <- get_ducklake_connection()
jan <- file.path(extract_dir, "jan.parquet")
feb <- file.path(extract_dir, "feb.parquet")
DBI::dbExecute(conn, sprintf(
  "COPY (SELECT 1::INTEGER AS id, 1.5::DOUBLE AS value)
   TO '%s' (FORMAT PARQUET);", jan
))
DBI::dbExecute(conn, sprintf(
  "COPY (SELECT 2::INTEGER AS id, 2.5::DOUBLE AS value)
   TO '%s' (FORMAT PARQUET);", feb
))

# Bring an existing Parquet extract into the lake without copying it
create_table(data.frame(id = integer(), value = numeric()), "readings")
add_data_files("readings", jan)

# Register another, tolerating a column the table doesn't have
add_data_files("readings", feb, ignore_extra_columns = TRUE)

# Create a new table and register an existing Parquet batch atomically
add_data_files("staging", feb, create = TRUE)

unlink(extract_dir, recursive = TRUE)

detach_ducklake("addfiles_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`add_table_column` *Add a column to a DuckLake table*

Description

Adds a column in place with `ALTER TABLE ... ADD COLUMN`. This is a metadata-only change: no data files are rewritten, history is preserved, and earlier snapshots still show the old schema. Compare `replace_table()`, which collects the table into R and rewrites it.

Usage

```
add_table_column(table_name, column_name, type, default = NULL)
```

Arguments

`table_name`	The table to change.
`column_name`	Name of the new column.
`type`	SQL type for the new column, e.g. `"INTEGER"`, `"DECIMAL(10,2)"`, or `"TIMESTAMP WITH TIME ZONE"`.

`default` Optional default value (an R scalar: character, numeric, logical, Date, or POSIXct). In DuckLake the default applies to existing rows as well as future inserts, so the new column appears filled everywhere. Without a default, the column reads `NA` for existing rows.

Value

Invisibly returns `NULL`.

See Also

`drop_table_column()`, `rename_table_column()`, `set_column_type()`, `rename_ducklake_table()`

Other schema evolution: `drop_table_column()`, `rename_ducklake_table()`, `rename_table_column()`, `set_column_type()`

Examples

```
lake_dir <- tempfile("addcol_lake_")
dir.create(lake_dir)
attach_ducklake("addcol_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

add_table_column("cars", "grade", "VARCHAR")
add_table_column("cars", "discount", "DECIMAL(5,2)", default = 0)

detach_ducklake("addcol_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

attach_ducklake *Create or attach a ducklake*

Description

Wrapper for the ducklake ATTACH command. Creates a new DuckLake if the specified name does not exist, or connects to an existing one. The lake can be detached with `detach_ducklake()`.

Usage

```
attach_ducklake(
  ducklake_name,
  lake_path,
  backend = c("duckdb", "postgres", "sqlite", "mysql"),
  catalog_connection_string = NULL,
  read_only = FALSE,
  override_data_path = FALSE,
  data_inlining_row_limit = NULL,
  encrypted = FALSE,
```

```
  snapshot_version = NULL,
  snapshot_time = NULL
)
```

Arguments

ducklake_name	Name for the ducklake, used as the database alias in DuckDB
lake_path	Directory path where the lake lives. For "duckdb" this is where the catalog file and Parquet data are stored. For other backends this sets the Parquet data location (DuckLake's DATA_PATH), which may also be an object-storage URI such as "s3://bucket/path" – register credentials first with create_storage_secret(). (The "duckdb" backend needs a local lake_path, since its catalog is a database file.)
backend	Catalog backend: "duckdb" (default), "postgres", "sqlite", or "mysql".
catalog_connection_string	Backend-specific connection string: "duckdb" Not required. Defaults to {ducklake_name}.ducklake. "postgres" libpq string, e.g. "dbname=mydb host=localhost". "sqlite" Path to the SQLite file, e.g. "metadata.sqlite". "mysql" MySQL connection string, e.g. "db=mydb host=localhost".
read_only	Attach in read-only mode (default FALSE).
override_data_path	Override the stored DATA_PATH in the catalog (default FALSE). Needed when restoring a backup to a different location.
data_inlining_row_limit	Optional integer. Sets the per-connection data inlining row limit. Inserts or deletes affecting fewer rows than this threshold are stored directly in the catalog instead of writing Parquet files. The default (when NULL) uses the DuckLake default of 10 rows. Set to 0 to disable inlining for this connection. This setting is not persisted; use set_inlining_row_limit() for persistent overrides.
encrypted	If TRUE, DuckLake encrypts the Parquet data files it writes. Encryption keys are stored in the catalog database, so anyone with access to the catalog can read the data – protect the catalog accordingly. Only applies when the lake is first created; an existing lake keeps the setting it was created with. The httpfs extension is loaded automatically: on some platforms (notably Windows) DuckDB's built-in crypto module is read-only and httpfs provides the writer. Default FALSE.
snapshot_version	Optional snapshot id. Attaches the lake pinned to that snapshot: queries see the lake exactly as it was then, and writes are rejected. Mutually exclusive with snapshot_time.
snapshot_time	Optional POSIXct or UTC timestamp string. Attaches the lake pinned to its state at that moment. Mutually exclusive with snapshot_version.

Details

By default DuckDB is used as the catalog database. Alternative backends (PostgreSQL, SQLite, MySQL) can be selected with the backend parameter, which enables concurrent multi-client access. See https://ducklake.select/docs/stable/duckdb/usage/choosing_a_catalog_database.

For credential management with PostgreSQL or MySQL, consider DuckDB's built-in secrets manager instead of embedding credentials in the connection string:

```
conn <- get_ducklake_connection()
DBI::dbExecute(conn, "CREATE SECRET (
    TYPE postgres,
    HOST '127.0.0.1',
    PORT 5432,
    DATABASE ducklake_catalog,
    USER 'analyst',
    PASSWORD 'secret'
)")
```

Then pass an empty or partial `catalog_connection_string`; DuckDB fills in the rest from the secret. See https://duckdb.org/docs/stable/configuration/secrets_manager.

Windows limitation: The `postgres` and `mysql` DuckDB extensions are not available on Windows (MinGW toolchain). Only `duckdb` and `sqlite` backends work there. Use Linux, macOS, or WSL for PostgreSQL/MySQL backends. See https://github.com/duckdb/duckdb/issues/7892.

Value

Invisibly, `NULL`. Called for its side effect of attaching the DuckLake catalog to the package's DuckDB connection.

See Also

`detach_ducklake()`, `install_ducklake()`, `create_storage_secret()`

Other connection management: `create_storage_secret()`, `detach_ducklake()`, `ducklake_extension_available()`, `get_ducklake_backend()`, `get_ducklake_connection()`, `install_ducklake()`, `set_ducklake_connection()`

Examples

```
# DuckDB catalog (default)
lake_dir <- tempfile("my_lake_")
dir.create(lake_dir)
attach_ducklake("my_lake", lake_path = lake_dir)
detach_ducklake("my_lake")

# Custom inlining threshold for a streaming workload
stream_dir <- tempfile("streaming_lake_")
dir.create(stream_dir)
attach_ducklake(
  "streaming_lake",
  lake_path = stream_dir,
  data_inlining_row_limit = 100
```

```
)

detach_ducklake("streaming_lake", shutdown = TRUE)
unlink(c(lake_dir, stream_dir), recursive = TRUE)

# The remaining forms need a catalog server, or extensions that are
# downloaded on first use, so they are not run here.
## Not run:
# PostgreSQL catalog
attach_ducklake(
  "my_lake",
  backend = "postgres",
  catalog_connection_string = "dbname=ducklake_catalog host=localhost",
  lake_path = "/shared/lake/data/"
)

# SQLite catalog
attach_ducklake(
  "my_lake",
  backend = "sqlite",
  catalog_connection_string = "metadata.sqlite",
  lake_path = "data_files/"
)

# MySQL catalog
attach_ducklake(
  "my_lake",
  backend = "mysql",
  catalog_connection_string = "db=ducklake_catalog host=localhost",
  lake_path = "data_files/"
)

# Encrypted Parquet files (keys live in the catalog); needs httpfs
attach_ducklake("secure_lake", lake_path = "path/to/lake", encrypted = TRUE)

# A frozen view of the lake as of snapshot 12, e.g. to reproduce a report
attach_ducklake("lake_v12", lake_path = "path/to/lake", snapshot_version = 12)

## End(Not run)
```

`attach_quack` *Connect to a remote Quack server*

Description

Attaches a remote Quack server as a catalog in the current session. Tables in the server's default database are then reachable as `quack_name.table_name` and can be queried with `get_ducklake_table()` or `dplyr::tbl()`.

Usage

```
attach_quack(quack_name, uri, token = NULL, disable_ssl = FALSE)
```

Arguments

quack_name	Name for the attached remote catalog, used as the database alias in DuckDB.
uri	Address of the Quack server, for example "quack:localhost" or "quack:data.example.org:9494". A bare host such as "localhost" is prefixed with quack: automatically. The default port is 9494.
token	Authentication token expected by the server. If NULL, the token is taken from a Quack secret (see CREATE SECRET) if one exists.
disable_ssl	Connect over plain HTTP instead of HTTPS (default FALSE). Only appropriate on a trusted network.

Details

A DuckLake served over Quack lives in its own catalog on the server rather than in the default database, so its tables are not exposed under quack_name. Query a served DuckLake with quack_query(), naming the lake's catalog, for example quack_query(uri, "SELECT * FROM trial.adsl").

Value

Invisibly, NULL. Called for its side effect of attaching the remote catalog to the package's DuckDB connection.

See Also

detach_quack(), quack_query(), quack_serve()

Other quack: detach_quack(), install_quack(), quack_query(), quack_serve(), quack_stop()

Examples

```
## Not run:
# A remote DuckDB database, queried through the attached catalog
attach_quack("warehouse", "quack:data.example.org", token = "super_secret")

get_ducklake_table("warehouse.sales") |>
  dplyr::filter(region == "EMEA") |>
  dplyr::collect()

detach_quack("warehouse")

## End(Not run)
```

backup_ducklake *Create a DuckLake backup*

Description

Creates a timestamped backup of the Parquet data files and, for file-based backends (DuckDB, SQLite), the catalog database file. For PostgreSQL/MySQL backends only data files are copied; use `pg_dump` / `mysqldump` for the catalog.

Usage

```
backup_ducklake(ducklake_name, lake_path, backup_path)
```

Arguments

`ducklake_name`	Name of the attached DuckLake
`lake_path`	Path to the DuckLake directory containing the data files (and catalog file for DuckDB/SQLite backends)
`backup_path`	Directory where backups should be stored. A timestamped subdirectory will be created within this path.

Details

For file-based backends the DuckLake is temporarily detached during backup to release file locks and ensure a consistent copy. It is automatically re-attached afterwards.

Important notes:

- Transactions committed after a backup won't be tracked when recovering. The data will exist in the Parquet files, but the backup will point to an earlier snapshot.
- Consider coordinating backups with maintenance operations (compaction and cleanup) for optimal storage efficiency.
- For production systems, schedule backups using `{cronR}` or `{taskscheduleR}`.

Value

Invisibly returns the path to the created backup directory

See Also

Other maintenance: `checkpoint_ducklake()`, `cleanup_old_files()`, `delete_orphaned_files()`, `expire_snapshots()`, `flush_inlined_data()`, `get_table_info()`, `list_ducklake_files()`, `merge_adjacent_files()`, `plot_table_files()`, `rewrite_data_files()`

Examples

```
# Create a DuckLake
lake_dir <- tempfile("my_lake")
dir.create(lake_dir)
attach_ducklake("my_lake", lake_path = lake_dir)

# Add some data
with_transaction(
  create_table(mtcars, "cars"),
  author = "User",
  commit_message = "Initial data"
)

# Create a backup
backup_dir <- backup_ducklake(
  ducklake_name = "my_lake",
  lake_path = lake_dir,
  backup_path = file.path(lake_dir, "backups")
)

# Restore (override_data_path needed when location differs):
# detach_ducklake("my_lake")
# attach_ducklake("my_lake", lake_path = backup_dir, override_data_path = TRUE)

detach_ducklake("my_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

begin_transaction *Begin a transaction*

Description

Starts a new transaction in the DuckDB connection. All subsequent operations will be part of this transaction until it is committed or rolled back.

Usage

```
begin_transaction(conn = NULL)
```

Arguments

conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

Transactions allow you to group multiple operations together and ensure they either all succeed or all fail. Use `commit_transaction()` to apply the changes or `rollback_transaction()` to discard them.

DuckDB supports full ACID transactions with multiple isolation levels.

Value

Invisibly returns TRUE on success

See Also

Other transactions: `commit_transaction()`, `rollback_transaction()`, `set_snapshot_metadata()`, `with_transaction()`

Examples

```
lake_dir <- tempfile("begin_lake_")
dir.create(lake_dir)
attach_ducklake("begin_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, status = "pending"), "jobs")

# Start a transaction
begin_transaction()

# Make some changes
get_ducklake_table("jobs") |>
  dplyr::filter(status == "pending") |>
  dplyr::mutate(status = "processed") |>
  ducklake_exec()

# Commit if everything looks good
commit_transaction()

# Or rollback if something went wrong
# rollback_transaction()

detach_ducklake("begin_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

checkpoint_ducklake *Run a DuckLake checkpoint*

Description

Runs all maintenance operations on the DuckLake catalog: flushes inlined data, expires old snapshots, merges small files, and cleans up unreferenced files.

Usage

```
checkpoint_ducklake(ducklake_name = NULL)
```

Arguments

ducklake_name	Name of the attached DuckLake catalog. If NULL, the current database is used.

Details

CHECKPOINT is the recommended one-stop maintenance command. It internally calls flush_inlined_data() along with compaction, snapshot expiration, and file cleanup.

Run checkpoints periodically (e.g., after a batch of streaming inserts) to consolidate inlined data and keep query performance optimal.

Value

Invisibly returns NULL.

Note

On Windows with a DuckDB-file catalog, the file-cleanup step of CHECKPOINT can fail because Windows does not allow the catalog file to be opened a second time while the lake is attached (a current DuckDB limitation). flush_inlined_data() is unaffected; on Windows, prefer it for routine use and run full checkpoints from a fresh session, or use a PostgreSQL/SQLite catalog.

See Also

flush_inlined_data(), set_inlining_row_limit()

Other data inlining: flush_inlined_data(), get_inlining_row_limit(), set_inlining_row_limit()

Other maintenance: backup_ducklake(), cleanup_old_files(), delete_orphaned_files(), expire_snapshots(), flush_inlined_data(), get_table_info(), list_ducklake_files(), merge_adjacent_files(), plot_table_files(), rewrite_data_files()

Examples

```
lake_dir <- tempfile("checkpoint_lake_")
dir.create(lake_dir)
attach_ducklake("checkpoint_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Run all maintenance
checkpoint_ducklake()

# Or specify a named lake
checkpoint_ducklake("checkpoint_lake")

detach_ducklake("checkpoint_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

cleanup_old_files *Delete files scheduled for removal*

Description

Physically deletes data files that are no longer referenced by any snapshot – typically files orphaned by `expire_snapshots()` or replaced by `merge_adjacent_files()`.

Usage

```
cleanup_old_files(
  ducklake_name = NULL,
  older_than = NULL,
  cleanup_all = FALSE,
  dry_run = FALSE
)
```

Arguments

`ducklake_name`	Name of the attached DuckLake catalog. If `NULL`, the current database is used.
`older_than`	Only delete files scheduled for deletion before this timestamp (POSIXct, converted to UTC, or character already in UTC). One of `older_than` or `cleanup_all` is required.
`cleanup_all`	If `TRUE`, delete all scheduled files regardless of when they were scheduled.
`dry_run`	If `TRUE`, only lists the files that would be deleted.

Details

As an alternative to calling this manually, a retention policy can be set once on the catalog with `DBI::dbExecute(get_ducklake_connection(), "CALL my_lake.set_option('delete_older_than', '1 week')")`, after which DuckLake cleans up eligible files automatically.

Value

A data frame listing the deleted (or deletable) files.

See Also

`expire_snapshots()`, `delete_orphaned_files()`, `checkpoint_ducklake()`

Other maintenance: `backup_ducklake()`, `checkpoint_ducklake()`, `delete_orphaned_files()`, `expire_snapshots()`, `flush_inlined_data()`, `get_table_info()`, `list_ducklake_files()`, `merge_adjacent_files()`, `plot_table_files()`, `rewrite_data_files()`

Examples

```
lake_dir <- tempfile("cleanup_lake_")
dir.create(lake_dir)
attach_ducklake("cleanup_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

expire_snapshots(older_than = Sys.time())

# Preview, then delete everything that is scheduled
cleanup_old_files(dry_run = TRUE, cleanup_all = TRUE)
cleanup_old_files(cleanup_all = TRUE)

# Only delete files scheduled more than a week ago
cleanup_old_files(older_than = Sys.time() - 7 * 24 * 60 * 60)

detach_ducklake("cleanup_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

commit_transaction *Commit a transaction*

Description

Commits the current transaction, making all changes permanent. Optionally adds metadata (author, commit message, and extra info) to the snapshot.

Usage

```
commit_transaction(
  conn = NULL,
  author = NULL,
  commit_message = NULL,
  commit_extra_info = NULL
)
```

Arguments

conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.
author	Optional author name to associate with the snapshot
commit_message	Optional commit message describing the changes
commit_extra_info	Optional extra information about the commit

Details

This function commits all changes made since `begin_transaction()` was called, making them permanent in the database.

If `author`, `commit_message`, or `commit_extra_info` are provided, they will be set using `CALL ducklake.set_commit_message()` within the transaction before the `COMMIT` statement, as required by the DuckLake v1.0 specification.

Value

Invisibly returns TRUE on success

See Also

Other transactions: `begin_transaction()`, `rollback_transaction()`, `set_snapshot_metadata()`, `with_transaction()`

Examples

```
lake_dir <- tempfile("commit_lake_")
dir.create(lake_dir)
attach_ducklake("commit_lake", lake_path = lake_dir)

# Basic commit
begin_transaction()
create_table(iris, "flowers")
commit_transaction()

# Commit with metadata
begin_transaction()
create_table(mtcars, "cars")
commit_transaction(
  author = "John Doe",
  commit_message = "Add cars dataset"
)

detach_ducklake("commit_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`create_storage_secret` *Store object storage credentials for a session*

Description

Registers credentials with DuckDB's secrets manager so a DuckLake can read and write data files on object storage (`lake_path = "s3://..."` and friends). Wraps `CREATE SECRET`.

Usage

```
create_storage_secret(
  type = c("s3", "gcs", "r2", "azure"),
  ...,
  provider = NULL,
  scope = NULL,
  name = NULL,
  persistent = FALSE
)
```

Arguments

type	Storage type: "s3" (also for S3-compatible stores), "gcs" (Google Cloud Storage), "r2" (Cloudflare R2), or "azure".
...	Named secret parameters passed through to CREATE SECRET, e.g. key_id, secret, region, session_token, endpoint, url_style, account_id (R2), or connection_string (Azure). Character values are quoted; logicals become true/false.
provider	Optional credential provider. The common one is "credential_chain", which picks up credentials the way AWS SDKs do (environment variables, profiles, instance metadata) so no key needs to be passed in code.
scope	Optional URI prefix (e.g. "s3://my-bucket") limiting which paths the secret applies to. Useful when different buckets need different credentials.
name	Optional name for the secret. Named secrets can be replaced and dropped individually; unnamed ones act as the default for their type.
persistent	If TRUE, the secret is written (unencrypted) to ~/.duckdb/stored_secrets and survives the session. The default FALSE keeps it in memory only, which is the right choice for credentials supplied from a vault or environment variable.

Details

The httpfs extension (or the azure extension for type = "azure") is loaded automatically.

Prefer provider = "credential_chain" over embedding long-lived keys in scripts. The secret's values are visible in the session via duckdb_secrets() (redacted) and travel with persistent storage unencrypted, so treat persistent = TRUE with the same care as a credentials file.

Value

Invisibly returns the secret's name (or NA_character_ for an unnamed secret).

See Also

attach_ducklake()

Other connection management: attach_ducklake(), detach_ducklake(), ducklake_extension_available(), get_ducklake_backend(), get_ducklake_connection(), install_ducklake(), set_ducklake_connection()

Examples

```
## Not run:
# Explicit keys, scoped to one bucket
create_storage_secret(
  "s3",
  key_id = Sys.getenv("AWS_ACCESS_KEY_ID"),
  secret = Sys.getenv("AWS_SECRET_ACCESS_KEY"),
  region = "us-east-1",
  scope = "s3://my-trial-lake"
)

# Let the AWS credential chain find credentials
create_storage_secret("s3", provider = "credential_chain")

# Then attach a lake whose data lives on S3
attach_ducklake("trial_lake", lake_path = "s3://my-trial-lake/data")

## End(Not run)
```

---

create_table *Create a DuckLake table*

Description

Create a DuckLake table

Usage

```
create_table(data_source, table_name, labels = TRUE)
```

Arguments

data_source	Raw data source. Can be: • A URL (http:// or https://) • A file path (e.g., "data.csv", "data.parquet") • An R data.frame or tibble • A lazy table (tbl_duckdb_connection or tbl_lazy)
table_name	Name of the new table
labels	When TRUE (the default) and the data has haven/labelled variable labels (label attributes on columns), store them in the lake as column comments – in the same transaction as the table creation, so both land as one snapshot. Collecting the table later restores the labels (see get_table_comments()), and every other client of the lake can read them too. Set to FALSE to skip.

Value

Invisibly, NULL. Called for its side effect of creating the table in the lake.

See Also

Other table operations: add_data_files(), create_view(), drop_view(), ducklake_exec(), get_ducklake_table(), get_metadata_table(), list_ducklake_tables(), replace_table(), show_ducklake_query()

Examples

```
lake_dir <- tempfile("create_lake_")
dir.create(lake_dir)
attach_ducklake("create_lake", lake_path = lake_dir)

# From data.frame
create_table(mtcars, "cars")

# From a local file
csv_path <- tempfile(fileext = ".csv")
utils::write.csv(mtcars, csv_path, row.names = FALSE)
create_table(csv_path, "cars_from_csv")

# From a lazy table (pipe-friendly)
get_ducklake_table("cars") |>
  dplyr::filter(cyl > 4) |>
  create_table("big_cars")

# From a URL -- needs network access and the httpfs extension
## Not run:
create_table("https://example.com/data.csv", "remote_table")

## End(Not run)

unlink(csv_path)

detach_ducklake("create_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

create_view *Create a DuckLake view from a dplyr pipeline*

Description

Stores a dplyr pipeline in the lake as a SQL view: the query runs fresh every time the view is read, so it always reflects the current data. Views live in the DuckLake catalog itself, which makes them a good home for shared business logic – a Python or SQL client of the same lake sees exactly the same definition.

Usage

```
create_view(.data, view_name, replace = TRUE)
```

Arguments

.data	A lazy table (a dplyr pipeline built on get_ducklake_table()). Not a data frame: a view stores a query, not data – use create_table() to store data.
view_name	Name for the view.
replace	Replace an existing view of the same name (default TRUE).

Details

Read a view back with get_ducklake_table(), which works for views and tables alike, and keep piping dplyr verbs onto it. Like tables, views are versioned: dropping or replacing one is a snapshot like any other.

Value

Invisibly returns NULL.

See Also

drop_view(), list_ducklake_tables(), replace_table() to materialize a pipeline as data instead.

Other table operations: add_data_files(), create_table(), drop_view(), ducklake_exec(), get_ducklake_table(), get_metadata_table(), list_ducklake_tables(), replace_table(), show_ducklake_query()

Examples

```
lake_dir <- tempfile("view_lake_")
dir.create(lake_dir)
attach_ducklake("view_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Encapsulate filtering logic the whole team should share
get_ducklake_table("cars") |>
  dplyr::filter(cyl == 4) |>
  dplyr::select(mpg, cyl, gear) |>
  create_view("v_efficient_cars")

# Reads run the stored query against current data
get_ducklake_table("v_efficient_cars") |> dplyr::collect()

detach_ducklake("view_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`delete_orphaned_files` *Delete orphaned files*

Description

Deletes files sitting in the lake's data path that are not tracked in the DuckLake metadata at all – for example, leftovers from a crashed write.

Usage

```
delete_orphaned_files(
  ducklake_name = NULL,
  older_than = NULL,
  cleanup_all = FALSE,
  dry_run = FALSE
)
```

Arguments

`ducklake_name`	Name of the attached DuckLake catalog. If `NULL`, the current database is used.
`older_than`	Only delete files scheduled for deletion before this timestamp (POSIXct, converted to UTC, or character already in UTC). One of `older_than` or `cleanup_all` is required.
`cleanup_all`	If `TRUE`, delete all scheduled files regardless of when they were scheduled.
`dry_run`	If `TRUE`, only lists the files that would be deleted.

Details

This differs from `cleanup_old_files()`, which removes files that *were* tracked but are scheduled for deletion.

Always run with `dry_run = TRUE` first and check the file list. Anything in the data path that DuckLake does not recognise is fair game, and the comparison is by exact path string: a data path registered with an irregularity such as a doubled slash (as R's `tempdir()` produces on macOS) makes *live* files look orphaned, and deleting them breaks the lake.

Value

A data frame listing the deleted (or deletable) files.

See Also

`cleanup_old_files()`, `checkpoint_ducklake()`

Other maintenance: `backup_ducklake()`, `checkpoint_ducklake()`, `cleanup_old_files()`, `expire_snapshots()`, `flush_inlined_data()`, `get_table_info()`, `list_ducklake_files()`, `merge_adjacent_files()`, `plot_table_files()`, `rewrite_data_files()`

Examples

```
lake_dir <- tempfile("orphan_lake_")
dir.create(lake_dir)
attach_ducklake("orphan_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Always preview orphan deletion first
delete_orphaned_files(dry_run = TRUE, cleanup_all = TRUE)

detach_ducklake("orphan_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

detach_ducklake *Detach from a ducklake*

Description

Detaches the DuckLake database but keeps the DuckDB connection alive by default. Use `shutdown = TRUE` to also close the connection and release file locks.

Usage

```
detach_ducklake(ducklake_name = NULL, shutdown = FALSE)
```

Arguments

`ducklake_name`	Optional name of the ducklake to detach.
`shutdown`	If `TRUE`, shut down the DuckDB connection after detaching. Only applies to the connection ducklake created itself; a connection registered with `set_ducklake_connection()` is never closed for you.

Value

Invisibly, `NULL`. Called for its side effect of detaching the catalog from the package's DuckDB connection.

See Also

Other connection management: `attach_ducklake()`, `create_storage_secret()`, `ducklake_extension_available()`, `get_ducklake_backend()`, `get_ducklake_connection()`, `install_ducklake()`, `set_ducklake_connection()`

Examples

```
lake_dir <- tempfile("detach_lake_")
dir.create(lake_dir)
attach_ducklake("detach_lake", lake_path = lake_dir)
# ... do work ...
detach_ducklake("detach_lake")

# Full shutdown when completely done
attach_ducklake("detach_lake", lake_path = lake_dir)
detach_ducklake("detach_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

detach_quack *Disconnect from a remote Quack server*

Description

Detaches a remote catalog previously attached with `attach_quack()`. The DuckDB connection itself stays alive.

Usage

```
detach_quack(quack_name = NULL)
```

Arguments

quack_name	Name of the remote catalog to detach. If `NULL`, nothing is detached.

Value

Invisibly, `NULL`. Called for its side effect of detaching the remote catalog.

See Also

`attach_quack()`

Other quack: `attach_quack()`, `install_quack()`, `quack_query()`, `quack_serve()`, `quack_stop()`

Examples

```
## Not run:
detach_quack("team")

## End(Not run)
```

drop_table_column *Drop a column from a DuckLake table*

Description

Removes a column in place with `ALTER TABLE ... DROP COLUMN`. This is a metadata-only change: the column disappears from the current schema, but earlier snapshots still contain it and remain queryable through time travel.

Usage

```
drop_table_column(table_name, column_name)
```

Arguments

`table_name`	The table to change.
`column_name`	Name of the column to drop.

Value

Invisibly returns `NULL`.

See Also

`add_table_column()`, `rename_table_column()`, `get_ducklake_table_version()` to read snapshots that still have the column

Other schema evolution: `add_table_column()`, `rename_ducklake_table()`, `rename_table_column()`, `set_column_type()`

Examples

```
lake_dir <- tempfile("dropcol_lake_")
dir.create(lake_dir)
attach_ducklake("dropcol_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

drop_table_column("cars", "carb")

detach_ducklake("dropcol_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

drop_view *Drop a DuckLake view*

Description

Removes a view from the lake with `DROP VIEW`. Only the stored query is dropped; the tables it reads are untouched.

Usage

```
drop_view(view_name)
```

Arguments

`view_name`	The view to drop.

Value

Invisibly returns `NULL`.

See Also

`create_view()`

Other table operations: `add_data_files()`, `create_table()`, `create_view()`, `ducklake_exec()`, `get_ducklake_table()`, `get_metadata_table()`, `list_ducklake_tables()`, `replace_table()`, `show_ducklake_query()`

Examples

```
lake_dir <- tempfile("dropview_lake_")
dir.create(lake_dir)
attach_ducklake("dropview_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

get_ducklake_table("cars") |>
  dplyr::filter(cyl == 4) |>
  create_view("v_efficient_cars")

drop_view("v_efficient_cars")

detach_ducklake("dropview_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

ducklake_exec *Execute DuckLake operations from dplyr queries*

Description

Execute DuckLake operations from dplyr queries

Usage

```
ducklake_exec(.data, table_name = NULL, .quiet = TRUE)
```

Arguments

.data	A dplyr query object (tbl_lazy) with accumulated operations
table_name	The target table name for the operation. If not provided, will be extracted from the table attribute (set by get_ducklake_table())
.quiet	Logical, whether to suppress debug output (default TRUE)

Details

This function automatically detects the type of operation based on dplyr verbs:

- Filter-only queries on `table_name` generate DELETE operations (removes rows that DON'T match filter)
- Queries with mutate() on `table_name` generate UPDATE operations
- Reads from *other* tables generate INSERT operations, appending their result into `table_name` with columns matched by name; `filter()` and joins are fine here, since the whole query just feeds the INSERT

A plain read from `table_name` itself is refused, since inserting a table's own rows back into it would duplicate them. Pipelines that compile to a subquery over `table_name` (grouped filters, `mutate()` followed by `filter()`) are also refused rather than mistranslated. Use `show_ducklake_query()` to preview the generated SQL without running it.

Value

The result from db_execute()

See Also

Other table operations: `add_data_files()`, `create_table()`, `create_view()`, `drop_view()`, `get_ducklake_table()`, `get_metadata_table()`, `list_ducklake_tables()`, `replace_table()`, `show_ducklake_query()`

Examples

```
lake_dir <- tempfile("exec_lake_")
dir.create(lake_dir)
attach_ducklake("exec_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, status = "pending"), "jobs")

# Update specific rows (table name inferred)
get_ducklake_table("jobs") |>
  dplyr::filter(id == 1) |>
  dplyr::mutate(status = "updated") |>
  ducklake_exec()

# Delete rows matching a filter
get_ducklake_table("jobs") |>
  dplyr::filter(status == "pending") |>
  ducklake_exec()

# Or provide the table name explicitly
get_ducklake_table("jobs") |>
  dplyr::mutate(status = "done") |>
  ducklake_exec("jobs")

detach_ducklake("exec_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

ducklake_extension_available

Is the ducklake DuckDB extension usable?

Description

Reports whether the ducklake DuckDB extension is already installed and can be loaded. The check opens a throwaway in-memory DuckDB connection with automatic extension installation turned off, so it never downloads anything and never writes to the extension cache in your home directory. It also leaves the package's own connection untouched.

Usage

```
ducklake_extension_available()
```

Details

Use it to guard code that should degrade gracefully when the extension is absent: examples, vignettes, tests, and conditional branches in scripts. The package's own examples are gated on it. To install the extension, call `install_ducklake()`.

The result is cached for the rest of the session, since spinning up DuckDB to re-answer the same question is wasteful when dozens of examples ask it. `install_ducklake()` clears the cache, so a `FALSE` answer becomes `TRUE` as soon as you install.

Value

A single logical: TRUE when the extension loads, FALSE otherwise (including when a connection cannot be opened).

See Also

Other connection management: attach_ducklake(), create_storage_secret(), detach_ducklake(), get_ducklake_backend(), get_ducklake_connection(), install_ducklake(), set_ducklake_connection()

Examples

```
if (ducklake_extension_available()) {
  message("ducklake extension is ready to use")
} else {
  message("run install_ducklake() first")
}
```

---

expire_snapshots *Expire old snapshots*

Description

Removes old snapshots from the DuckLake catalog. Expiring snapshots gives up the ability to time-travel to them, and schedules the data files that only they referenced for deletion.

Usage

```
expire_snapshots(
  ducklake_name = NULL,
  older_than = NULL,
  versions = NULL,
  dry_run = FALSE
)
```

Arguments

ducklake_name	Name of the attached DuckLake catalog. If NULL, the current database is used.
older_than	Expire all snapshots older than this timestamp (POSIXct or character in ISO 8601 format). POSIXct values are converted to UTC, which is how DuckLake records snapshot times; character values must already be UTC. At least one of older_than or versions must be provided.
versions	Integer vector of specific snapshot ids to expire (see list_table_snapshots()).
dry_run	If TRUE, only lists the snapshots that would be expired without expiring them.

Details

Expiring snapshots does not delete any files by itself: files that are no longer referenced are merely scheduled for deletion. Run `cleanup_old_files()` afterwards to reclaim the storage, or let `checkpoint_ducklake()` handle both steps.

The most recent snapshot can never be expired.

Value

A data frame listing the expired (or, with `dry_run = TRUE`, expirable) snapshots.

See Also

`cleanup_old_files()`, `checkpoint_ducklake()`, `list_table_snapshots()`

Other maintenance: `backup_ducklake()`, `checkpoint_ducklake()`, `cleanup_old_files()`, `delete_orphaned_files()`, `flush_inlined_data()`, `get_table_info()`, `list_ducklake_files()`, `merge_adjacent_files()`, `plot_table_files()`, `rewrite_data_files()`

Examples

```
lake_dir <- tempfile("expire_lake_")
dir.create(lake_dir)
attach_ducklake("expire_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

rows_insert(
  get_ducklake_table("cars"),
  data.frame(mpg = 30, cyl = 4),
  by = "mpg"
)

# Preview what a one-week retention policy would remove
expire_snapshots(older_than = Sys.time() - 7 * 24 * 60 * 60, dry_run = TRUE)

# Expire everything older than now, then reclaim the storage
expire_snapshots(older_than = Sys.time())
cleanup_old_files(cleanup_all = TRUE)

detach_ducklake("expire_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

flush_inlined_data *Flush inlined data to Parquet files*

Description

Materialises data that has been stored inline in the catalog database into Parquet files on the data path. This includes both inlined inserts and inlined deletions.

Usage

```
flush_inlined_data(ducklake_name = NULL, table_name = NULL, schema_name = NULL)
```

Arguments

ducklake_name	Name of the attached DuckLake catalog. If NULL, the current database is used.
table_name	Optional table name. When provided, only flushes inlined data for that table.
schema_name	Optional schema name. When provided, only flushes inlined data for tables in that schema.

Details

Flushing writes inlined rows to consolidated Parquet files and cleans up the inlined data tables. Time-travel information is preserved: flushed rows that had been deleted will produce a partial deletion file with snapshot metadata.

Tables with auto_compact set to FALSE are skipped when flushing an entire lake or schema. Use an explicit table_name to flush those tables.

If a table has a sort order defined, the flushed Parquet file will be sorted by those keys.

Value

A data frame with columns schema_name, table_name, and rows_flushed. Tables with no inlined data are omitted.

See Also

set_inlining_row_limit(), checkpoint_ducklake()

Other data inlining: checkpoint_ducklake(), get_inlining_row_limit(), set_inlining_row_limit()

Other maintenance: backup_ducklake(), checkpoint_ducklake(), cleanup_old_files(), delete_orphaned_files(), expire_snapshots(), get_table_info(), list_ducklake_files(), merge_adjacent_files(), plot_table_files(), rewrite_data_files()

Examples

```
lake_dir <- tempfile("flush_lake_")
dir.create(lake_dir)
attach_ducklake("flush_lake", lake_path = lake_dir,
                data_inlining_row_limit = 10)
create_table(head(mtcars, 3), "cars")

# Flush everything
flush_inlined_data()

# Flush a specific table
flush_inlined_data(table_name = "cars")

detach_ducklake("flush_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`get_ducklake_backend` *Get the catalog backend type of an attached lake*

Description

Get the catalog backend type of an attached lake

Usage

```
get_ducklake_backend(ducklake_name = NULL)
```

Arguments

`ducklake_name`	Name of the lake to look up. When `NULL` (the default), the lake the session is currently `USE`ing is looked up.

Value

One of `"duckdb"`, `"postgres"`, `"sqlite"`, or `"mysql"`. Defaults to `"duckdb"` when the lake is unknown.

See Also

Other connection management: `attach_ducklake()`, `create_storage_secret()`, `detach_ducklake()`, `ducklake_extension_available()`, `get_ducklake_connection()`, `install_ducklake()`, `set_ducklake_connection(`

Examples

```
lake_dir <- tempfile("backend_lake_")
dir.create(lake_dir)
attach_ducklake("backend_lake", lake_path = lake_dir)

get_ducklake_backend()

# With several lakes attached, look one up by name
get_ducklake_backend("backend_lake")

detach_ducklake("backend_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

get_ducklake_connection

Get the DuckDB connection used by ducklake

Description

Returns the DuckDB connection that all ducklake functions share. The first call creates the connection automatically, so you never need to set one up yourself. If you want ducklake to use a connection you have created (for example, one shared with other tools), register it first with `set_ducklake_connection()`.

Usage

```
get_ducklake_connection()
```

Details

The automatically created connection is backed by a temporary database file (not `:memory:`) with a spill directory configured, so larger-than-memory operations work out of the box. It is closed automatically when the R session ends.

Value

A DuckDB connection object (a `duckdb_connection`).

See Also

Other connection management: `attach_ducklake()`, `create_storage_secret()`, `detach_ducklake()`, `ducklake_extension_available()`, `get_ducklake_backend()`, `install_ducklake()`, `set_ducklake_connection()`

Examples

```
conn <- get_ducklake_connection()
DBI::dbGetQuery(conn, "SELECT version()")

detach_ducklake(shutdown = TRUE)
```

get_ducklake_options *List the options set on a DuckLake*

Description

Reads the configuration options recorded in the metadata catalog, including their scope (global, schema, or table).

Usage

```
get_ducklake_options(ducklake_name = NULL)
```

Arguments

ducklake_name	Optional name of the attached DuckLake catalog. If NULL, the current database is used.

Value

A data frame with one row per option setting, including option_name, value, scope (GLOBAL, SCHEMA, or TABLE), and scope_entry. Options left at their defaults are not listed.

See Also

set_ducklake_option()

Other options: set_ducklake_option()

Examples

```
lake_dir <- tempfile("getopt_lake_")
dir.create(lake_dir)
attach_ducklake("getopt_lake", lake_path = lake_dir)

set_ducklake_option("parquet_compression", "zstd")
get_ducklake_options()

detach_ducklake("getopt_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

get_ducklake_table *Get a DuckLake table*

Description

Returns a lazy reference to a table in the attached DuckLake. Like dplyr::tbl(), nothing is read until you collect(): build up your filter()/mutate()/summarise() pipeline first and DuckDB executes it as a single query, only pulling the rows you asked for into R.

Usage

```
get_ducklake_table(tbl_name)
```

Arguments

tbl_name	Character string, name of the table to retrieve.

Value

A lazy table (class `tbl_ducklake`) that works with dplyr verbs. The table name is stored in the `ducklake_table_name` attribute.

See Also

`create_table()` to create tables, `get_ducklake_table_asof()` and `get_ducklake_table_version()` for time-travel reads.

Other table operations: `add_data_files()`, `create_table()`, `create_view()`, `drop_view()`, `ducklake_exec()`, `get_metadata_table()`, `list_ducklake_tables()`, `replace_table()`, `show_ducklake_query()`

Examples

```
lake_dir <- tempfile("cars_lake_")
dir.create(lake_dir)
attach_ducklake("cars_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Query lazily with dplyr, then collect
get_ducklake_table("cars") |>
  dplyr::filter(cyl > 4) |>
  dplyr::summarise(avg_mpg = mean(mpg), .by = cyl) |>
  dplyr::collect()

detach_ducklake("cars_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

get_ducklake_table_asof

Query a table at a specific timestamp (time travel)

Description

Retrieves data from a DuckLake table as it existed at a specific point in time using DuckLake's AT (TIMESTAMP => ...) syntax.

Usage

```
get_ducklake_table_asof(table_name, timestamp, conn = NULL)
```

Arguments

`table_name`	The name of the table to query
`timestamp`	A POSIXct timestamp (converted to UTC, which is how DuckLake records snapshot times) or character string in ISO 8601 format already in UTC (e.g., "2024-01-15 10:30:00")
`conn`	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

DuckLake supports time-travel queries, allowing you to query historical data as it existed at a specific timestamp. This uses the syntax: `SELECT * FROM table AT (TIMESTAMP => 'timestamp')`

This is useful for:

- Auditing changes over time
- Recovering accidentally deleted or modified data
- Comparing data states across different time points
- Regulatory compliance and data lineage documentation

The timestamp must be within the range of available snapshots for the table. Use `list_table_snapshots()` to see available snapshot times.

Important: When querying at a snapshot's exact timestamp, you may need to add a small time buffer (e.g., +1 second) to ensure the snapshot is found. This is because the time-travel query looks for snapshots created at or before the specified timestamp.

Value

A dplyr lazy query object (tbl_lazy) that can be further manipulated with dplyr verbs

See Also

Other time travel: `get_ducklake_table_version()`, `get_table_changes()`, `list_table_snapshots()`, `plot_snapshots()`, `plot_table_changes()`, `restore_table_version()`

Examples

```
lake_dir <- tempfile("asof_lake_")
dir.create(lake_dir)
attach_ducklake("asof_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, amount = c(10, 20, 30)), "orders")

rows_insert(
  get_ducklake_table("orders"),
  data.frame(id = 4L, amount = 40),
  by = "id"
)

# Query data at a specific snapshot time
snapshots <- list_table_snapshots("orders")
# Add 1 second to ensure the snapshot is found
get_ducklake_table_asof("orders", snapshots$snapshot_time[1] + 1) |>
  dplyr::summarise(total = sum(amount)) |>
  dplyr::collect()

detach_ducklake("asof_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

get_ducklake_table_version *Query a table at a specific version/snapshot*

Description

Retrieves data from a DuckLake table at a specific snapshot ID using DuckLake's AT (VERSION => ...) syntax.

Usage

```
get_ducklake_table_version(table_name, version, conn = NULL)
```

Arguments

table_name	The name of the table to query
version	The snapshot_id to query (get this from list_table_snapshots())
conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

This function allows you to query a specific snapshot of a table using its snapshot_id. This uses the syntax: SELECT * FROM table AT (VERSION => snapshot_id)

Each time you create or modify a table within a transaction, DuckLake creates a new snapshot with a unique snapshot_id. Note that snapshot_id and schema_version are typically the same value - both represent the snapshot identifier.

Use list_table_snapshots(table_name) to see all available snapshots and their IDs.

Value

A dplyr lazy query object (tbl_lazy) that can be further manipulated with dplyr verbs

See Also

Other time travel: get_ducklake_table_asof(), get_table_changes(), list_table_snapshots(), plot_snapshots(), plot_table_changes(), restore_table_version()

Examples

```
lake_dir <- tempfile("version_lake_")
dir.create(lake_dir)
attach_ducklake("version_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, amount = c(10, 20, 30)), "orders")

rows_insert(
  get_ducklake_table("orders"),
```

```
  data.frame(id = 4L, amount = 40),
  by = "id"
)

# Get available snapshots
snapshots <- list_table_snapshots("orders")

# Query the first snapshot version
get_ducklake_table_version("orders", snapshots$snapshot_id[1]) |>
  dplyr::collect()

detach_ducklake("version_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`get_inlining_row_limit` *Get the current data inlining row limit*

Description

Returns the effective data inlining row limit. When no table- or schema-level override is configured, the global DuckDB default is returned.

Usage

```
get_inlining_row_limit(
  table_name = NULL,
  schema_name = NULL,
  ducklake_name = NULL
)
```

Arguments

`table_name`	Optional table name to query the table-level override.
`schema_name`	Optional schema name to query the schema-level override.
`ducklake_name`	Optional name of the attached DuckLake catalog. If `NULL`, the current database is used.

Value

An integer: the effective inlining row limit.

See Also

`set_inlining_row_limit()`

Other data inlining: `checkpoint_ducklake()`, `flush_inlined_data()`, `set_inlining_row_limit()`

Examples

```
lake_dir <- tempfile("inline_get_lake_")
dir.create(lake_dir)
attach_ducklake("inline_get_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

set_inlining_row_limit(100, table_name = "cars")

# Global default
get_inlining_row_limit()

# Table-specific limit
get_inlining_row_limit(table_name = "cars")

detach_ducklake("inline_get_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

get_metadata_table *Get a DuckLake metadata table*

Description

DuckLake keeps all of its bookkeeping – snapshots, table schemas, data file locations, and more – in ordinary tables inside the catalog database. This function gives you a lazy reference to any of them, which is handy for auditing and for understanding how your lake evolves.

Usage

```
get_metadata_table(tbl_name, ducklake_name = NULL)
```

Arguments

`tbl_name`	Character string, name of the metadata table to retrieve (e.g., `"ducklake_snapshot"`).
`ducklake_name`	Character string, name of the ducklake database (optional, defaults to the currently active ducklake).

Details

Commonly useful tables include `ducklake_snapshot` (one row per snapshot), `ducklake_table` (registered tables), and `ducklake_data_file` (the Parquet files backing each table). The full list is in the DuckLake specification.

Value

A lazy table that works with dplyr verbs.

See Also

`list_table_snapshots()` for a friendlier view of snapshot history.

Other table operations: `add_data_files()`, `create_table()`, `create_view()`, `drop_view()`, `ducklake_exec()`, `get_ducklake_table()`, `list_ducklake_tables()`, `replace_table()`, `show_ducklake_query()`

Examples

```
lake_dir <- tempfile("meta_lake_")
dir.create(lake_dir)
attach_ducklake("meta_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Every snapshot ever taken
get_metadata_table("ducklake_snapshot") |> dplyr::collect()

# Which Parquet files back the lake?
get_metadata_table("ducklake_data_file") |>
  dplyr::select(data_file_id, path) |>
  dplyr::collect()

detach_ducklake("meta_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

get_table_changes *Get the changes made to a table between two snapshots*

Description

Returns the exact rows that were inserted, deleted, or updated in a table between two snapshots (inclusive), using DuckLake's data change feed. Useful for auditing and for change-data-capture style pipelines.

Usage

```
get_table_changes(table_name, start, end, ducklake_name = NULL, conn = NULL)
```

Arguments

`table_name`	The name of the table to inspect.
`start`	The first snapshot to include: either a snapshot id (see `list_table_snapshots()`) or a timestamp (POSIXct or character).
`end`	The last snapshot to include, in the same form as `start`.
`ducklake_name`	Optional name of the attached DuckLake catalog. If `NULL`, the current database is used.
`conn`	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

Both bounds must be of the same kind: two snapshot ids or two timestamps. POSIXct bounds are converted to UTC, matching the snapshot times DuckLake records; character bounds are passed through as-is and must already be in UTC. Bounds before the lake's first snapshot are rejected by DuckLake, so prefer snapshot times from `list_table_snapshots()`. Updates appear as two rows – the row as it looked before the change (`update_preimage`) and after it (`update_postimage`).

This wraps DuckLake's `table_changes()` function.

Value

A dplyr lazy query object (tbl_lazy). In addition to the table's own columns it carries `snapshot_id` (the snapshot that made the change), `rowid` (the changed row's identifier), and `change_type` (`"insert"`, `"delete"`, `"update_preimage"`, or `"update_postimage"`).

See Also

`list_table_snapshots()`, `get_ducklake_table_version()`, `get_ducklake_table_asof()`

Other time travel: `get_ducklake_table_asof()`, `get_ducklake_table_version()`, `list_table_snapshots()`, `plot_snapshots()`, `plot_table_changes()`, `restore_table_version()`

Examples

```
lake_dir <- tempfile("changes_lake_")
dir.create(lake_dir)
attach_ducklake("changes_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, amount = c(10, 20, 30)), "orders")

rows_delete(
  get_ducklake_table("orders"),
  data.frame(id = 1L),
  by = "id"
)
snaps <- list_table_snapshots("orders")

# What changed in the most recent snapshot?
latest <- max(snaps$snapshot_id)
get_table_changes("orders", latest, latest) |> dplyr::collect()

# Every change across the table's full history, by timestamp
get_table_changes(
  "orders",
  min(snaps$snapshot_time), max(snaps$snapshot_time) + 1
) |>
  dplyr::filter(change_type == "delete") |>
  dplyr::collect()

detach_ducklake("changes_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

get_table_comments *Read table, view, and column comments from a DuckLake catalog*

Description

Returns the current comments stored in the lake – via `set_table_comment()`, `set_column_comments()`, `create_table()`'s label sync, or any other client – as a tidy data frame.

Usage

```
get_table_comments(table_name = NULL, ducklake_name = NULL)
```

Arguments

table_name	Optional table (or view) name to filter to.
ducklake_name	Optional name of the attached DuckLake catalog. If NULL, the current database is used.

Value

A data frame with one row per comment: `object_type` (`"table"`, `"view"`, or `"column"`), `table_name`, `column_name` (`NA` for tables and views), and `comment`. Zero rows when nothing is commented.

See Also

`get_metadata_table()` for the raw `ducklake_tag` and `ducklake_column_tag` catalog tables.

Other table documentation: `set_column_comments()`, `set_table_comment()`

Examples

```
lake_dir <- tempfile("readcomment_lake_")
dir.create(lake_dir)
attach_ducklake("readcomment_lake", lake_path = lake_dir)
create_table(mtcars, "cars")
set_table_comment("cars", "Motor Trend road tests")
set_column_comments("cars", mpg = "Miles per US gallon")

# Everything documented in the lake
get_table_comments()

# One table's documentation
get_table_comments("cars")

detach_ducklake("readcomment_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

get_table_info *Get file statistics for the tables in a lake*

Description

Returns per-table storage statistics from the DuckLake catalog: how many Parquet data files each table has and their total size, plus the same for delete files.

Usage

```
get_table_info(table_name = NULL, ducklake_name = NULL, conn = NULL)
```

Arguments

table_name	Optional table name. When provided, only that table's row is returned.
ducklake_name	Name of the attached DuckLake catalog. If NULL, the current database is used.
conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

These statistics are the raw material for storage maintenance decisions: many small files are worth compacting with merge_adjacent_files(), and a growing delete-file share is a sign to run rewrite_data_files(). Rows that are still inlined in the catalog (see set_inlining_row_limit()) are not in any Parquet file yet, so small recent writes may not show up in the counts until flush_inlined_data() writes them out.

This wraps DuckLake's ducklake_table_info() function.

Value

A data frame with one row per table and columns table_name, schema_id, table_id, table_uuid, file_count, file_size_bytes, delete_file_count, and delete_file_size_bytes.

See Also

plot_table_files(), merge_adjacent_files(), rewrite_data_files()

Other maintenance: backup_ducklake(), checkpoint_ducklake(), cleanup_old_files(), delete_orphaned_files(), expire_snapshots(), flush_inlined_data(), list_ducklake_files(), merge_adjacent_files(), plot_table_files(), rewrite_data_files()

Examples

```
lake_dir <- tempfile("info_lake_")
dir.create(lake_dir)
attach_ducklake("info_lake", lake_path = lake_dir)
create_table(mtcars, "cars")
```

```
# File statistics for every table in the lake
get_table_info()

# Just one table
get_table_info("cars")

detach_ducklake("info_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`get_table_partitions` *List the partitioning keys of tables in a lake*

Description

Reads the current partitioning keys from the DuckLake metadata catalog.

Usage

```
get_table_partitions(table_name = NULL, ducklake_name = NULL)
```

Arguments

`table_name`	Optional table name to filter to a single table.
`ducklake_name`	Optional name of the attached DuckLake catalog. If `NULL`, the current database is used.

Value

A data frame with one row per partition key: `table_name`, `partition_key_index`, `column_name`, and `transform` (e.g. `"identity"` or `"year"`). Zero rows when nothing is partitioned.

See Also

`set_table_partitioning()`, `get_metadata_table()`

Other partitioning: `reset_table_partitioning()`, `set_table_partitioning()`

Examples

```
lake_dir <- tempfile("getpart_lake_")
dir.create(lake_dir)
attach_ducklake("getpart_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

set_table_partitioning("cars", "cyl")

# All partitioned tables in the lake
get_table_partitions()
```

```
# Keys for one table
get_table_partitions("cars")

detach_ducklake("getpart_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`install_ducklake` *Install the ducklake extension to duckdb*

Description

Installs the ducklake DuckDB extension and optionally the extensions for alternative catalog backends (postgres, sqlite, mysql). Only needs to be run once per DuckDB version.

Usage

```
install_ducklake(backend = NULL)
```

Arguments

backend	Optional character vector of backends to install. The ducklake extension is always installed. Pass `"postgres"`, `"sqlite"`, and/or `"mysql"` to install the corresponding backend extensions.

Value

Invisibly, `NULL`. Called for its side effect of installing the DuckDB extensions into the local extension cache.

Note

On Windows the `postgres` and `mysql` extensions are not available (MinGW toolchain). See `attach_ducklake()` for details.

See Also

Other connection management: `attach_ducklake()`, `create_storage_secret()`, `detach_ducklake()`, `ducklake_extension_available()`, `get_ducklake_backend()`, `get_ducklake_connection()`, `set_ducklake_connection()`

Examples

```
## Not run:
install_ducklake()
install_ducklake(backend = "postgres")
install_ducklake(backend = c("postgres", "sqlite", "mysql"))

## End(Not run)
```

install_quack *Install the Quack extension*

Description

Installs the Quack DuckDB extension, which provides the `quack:` client-server protocol. Quack is a core extension from DuckDB 1.5.3 onward, so it is autoloaded the first time it is used. Installing it ahead of time is useful on machines that have no internet access at query time.

Usage

```
install_quack(load = TRUE)
```

Arguments

`load`	If `TRUE` (the default), load the extension after installing it.

Value

Invisibly, `NULL`. Called for its side effect of installing the quack DuckDB extension into the local extension cache.

See Also

`attach_quack()`, `quack_serve()`

Other quack: `attach_quack()`, `detach_quack()`, `quack_query()`, `quack_serve()`, `quack_stop()`

Examples

```
## Not run:
install_quack()

## End(Not run)
```

list_ducklake_files *List the data files backing a DuckLake table*

Description

Returns the Parquet data files (and any delete files) that make up a table, optionally as of a past snapshot.

Usage

```
list_ducklake_files(
  table_name,
  schema_name = NULL,
  snapshot_version = NULL,
  snapshot_time = NULL,
  ducklake_name = NULL
)
```

Arguments

table_name	The table whose files to list.
schema_name	Optional schema containing the table (defaults to the lake's main schema).
snapshot_version	Optional snapshot id: list the files as of that snapshot. Mutually exclusive with snapshot_time.
snapshot_time	Optional POSIXct or UTC timestamp string: list the files as of that moment. Mutually exclusive with snapshot_version.
ducklake_name	Optional name of the attached DuckLake catalog. If NULL, the current database is used.

Details

Wraps ducklake_list_files(). For per-table file counts and sizes across the whole lake, see get_table_info(); for a picture of storage layout, see plot_table_files().

Value

A data frame with one row per data file, including data_file, data_file_size_bytes, and the associated delete_file columns (NA when a file has no deletes).

See Also

add_data_files(), get_table_info()

Other maintenance: backup_ducklake(), checkpoint_ducklake(), cleanup_old_files(), delete_orphaned_files(), expire_snapshots(), flush_inlined_data(), get_table_info(), merge_adjacent_files(), plot_table_files(), rewrite_data_files()

Examples

```
lake_dir <- tempfile("listfiles_lake_")
dir.create(lake_dir)
attach_ducklake("listfiles_lake", lake_path = lake_dir)
create_table(mtcars, "cars")
rows_insert(
  get_ducklake_table("cars"),
  data.frame(mpg = 30, cyl = 4),
  by = "mpg"
```

```
)

# Files behind a table right now
list_ducklake_files("cars")

# Files as of an earlier snapshot
first <- min(list_table_snapshots("cars")$snapshot_id)
list_ducklake_files("cars", snapshot_version = first)

detach_ducklake("listfiles_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`list_ducklake_tables` *List the tables and views in a DuckLake catalog*

Description

Returns every table and view in the lake as a tidy data frame – the quick answer to "what is in here?".

Usage

```
list_ducklake_tables(ducklake_name = NULL)
```

Arguments

`ducklake_name`	Optional name of the attached DuckLake catalog. If `NULL`, the current database is used.

Value

A data frame with one row per object: `schema_name`, `table_name`, and `type` (`"table"` or `"view"`).

See Also

`get_table_info()` for per-table file statistics, `get_table_comments()` for stored documentation, `get_ducklake_table()` to read any listed object.

Other table operations: `add_data_files()`, `create_table()`, `create_view()`, `drop_view()`, `ducklake_exec()`, `get_ducklake_table()`, `get_metadata_table()`, `replace_table()`, `show_ducklake_query()`

Examples

```
lake_dir <- tempfile("list_lake_")
dir.create(lake_dir)
attach_ducklake("list_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

list_ducklake_tables()
```

```
detach_ducklake("list_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`list_table_snapshots` *List available snapshots for a table*

Description

Retrieves information about available snapshots/versions for a table.

Usage

```
list_table_snapshots(table_name = NULL, ducklake_name = NULL, conn = NULL)
```

Arguments

`table_name`	The name of the table to query
`ducklake_name`	The name of the ducklake (database) to query. If NULL, will attempt to infer from current database.
`conn`	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

This function queries the snapshot history of a table, showing available versions and their timestamps. This is useful for understanding what historical versions are available for time-travel queries.

Value

A data frame with snapshot information (version, timestamp, etc.)

See Also

Other time travel: `get_ducklake_table_asof()`, `get_ducklake_table_version()`, `get_table_changes()`, `plot_snapshots()`, `plot_table_changes()`, `restore_table_version()`

Examples

```
lake_dir <- tempfile("snaplist_lake_")
dir.create(lake_dir)
attach_ducklake("snaplist_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, amount = c(10, 20, 30)), "orders")

# List all snapshots for a table
list_table_snapshots("orders")
```

```
detach_ducklake("snaplist_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`merge_adjacent_files` *Merge adjacent Parquet files*

Description

Compacts small adjacent Parquet files into larger ones. Frequent small inserts each write their own file; merging keeps file counts down and scans fast.

Usage

```
merge_adjacent_files(
  ducklake_name = NULL,
  table_name = NULL,
  schema_name = NULL,
  max_compacted_files = NULL,
  min_file_size = NULL,
  max_file_size = NULL
)
```

Arguments

`ducklake_name`	Name of the attached DuckLake catalog. If `NULL`, the current database is used.
`table_name`	Optional table name. When provided, only that table is compacted.
`schema_name`	Optional schema name. When provided, only tables in that schema are compacted.
`max_compacted_files`	Optional cap on the number of compaction operations per table in a single call.
`min_file_size`	Optional minimum file size in bytes; smaller files are excluded from merging.
`max_file_size`	Optional maximum file size in bytes; files at or above this size are excluded. Defaults to the lake's target file size.

Details

Merging does not delete the original small files – they may still be referenced by older snapshots. They are scheduled for deletion once no snapshot references them; run `cleanup_old_files()` to remove them.

Value

A data frame with one row per output file (columns `schema_name`, `table_name`, `files_processed`, `files_created`).

See Also

cleanup_old_files(), checkpoint_ducklake(), flush_inlined_data()

Other maintenance: backup_ducklake(), checkpoint_ducklake(), cleanup_old_files(), delete_orphaned_files(), expire_snapshots(), flush_inlined_data(), get_table_info(), list_ducklake_files(), plot_table_files(), rewrite_data_files()

Examples

```
lake_dir <- tempfile("merge_files_lake_")
dir.create(lake_dir)
attach_ducklake("merge_files_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

rows_insert(
  get_ducklake_table("cars"),
  data.frame(mpg = 30, cyl = 4),
  by = "mpg"
)

# Compact the whole lake
merge_adjacent_files()

# Compact one table, only touching files under 10 MB
merge_adjacent_files(table_name = "cars", max_file_size = 10e6)

detach_ducklake("merge_files_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

merge_into *Merge a source table into a DuckLake table*

Description

Runs a SQL `MERGE INTO` statement: rows of `target` are matched against rows of `source` on the `by` columns, then updated, deleted, or left alone, while unmatched source rows can be inserted and target rows missing from the source can be removed. The whole operation is atomic – one snapshot, with row lineage preserved in the change feed.

Usage

```
merge_into(
  target,
  source,
  by,
  when_matched = c("update", "delete", "nothing"),
  when_not_matched = c("insert", "nothing"),
  matched_condition = NULL,
```

```
  not_matched_condition = NULL,
  delete_missing = FALSE,
  .quiet = TRUE
)
```

Arguments

target	The table to modify: a table from get_ducklake_table() or a table name.
source	The rows to merge in: a data frame or a lazy table on the same connection.
by	Character vector of key column(s) to match on. Rows with NULL key values never match.
when_matched	What to do with target rows that match a source row: "update" (default) sets the columns the two tables share, "delete" removes the row, "nothing" leaves it alone.
when_not_matched	What to do with source rows that match no target row: "insert" (default) adds them, "nothing" skips them.
matched_condition	Optional SQL expression further restricting the when_matched action, written against the aliases target and source, e.g. "source.amt > target.amt".
not_matched_condition	Optional SQL expression further restricting the when_not_matched action.
delete_missing	Also delete target rows that have no match in the source (WHEN NOT MATCHED BY SOURCE THEN DELETE). Combined with the update action this synchronizes target to source.
.quiet	Logical, whether to suppress the row-count message (default TRUE).

Details

This is not a join. Joins (left_join() and friends) read from the lake and build a new result without touching either table; merge_into() changes the rows of target in place. For the common update-or-insert case, reach for rows_upsert() first – merge_into() is for the cases it cannot express: conditional clauses, deletes of matched rows, and synchronizing a table with a staging source.

Choosing how to change a table:

- To look up or combine data for analysis, use dplyr joins (left_join() and friends). Joins read from the lake and build a new result; they never modify a lake table.
- To append, correct, or remove specific rows, use rows_insert(), rows_update(), or rows_delete(). Each call is a single SQL statement against the existing table – no data leaves the database, and with data inlining enabled (DuckLake's default) small changes land in the catalog without creating tiny Parquet files.
- To update rows that exist and insert the ones that don't in one atomic statement, use rows_upsert().
- For conditional merge logic or deletes driven by a staging table, use merge_into().
- To change a table's shape without touching its data – add, drop, or rename columns, widen a type – use the schema evolution family (add_table_column() and friends): metadata-only changes that rewrite nothing.

- For bulk transformations that touch most rows, use `replace_table()`. It collects the transformed data into R and rewrites the whole table – heavier than the row operations, and it resets the row lineage that the in-place operations preserve in the change feed.

A tempting alternative – joining the source to the table and calling `replace_table()` on the result – rewrites every row and records the change as a wholesale replacement. `merge_into()` touches only the affected rows, so `get_table_changes()` afterward shows exactly which rows were inserted, updated, or deleted.

Conditions are SQL:

`matched_condition` and `not_matched_condition` are raw SQL expressions, not dplyr code. They are pasted into the statement as written (only a `;` is rejected), so build them from trusted input only.

DuckLake-specific behavior:

- Update sets the columns present in both tables (minus `by`); inserted rows receive the column default (`NULL` when none is defined) in target columns the source lacks.
- `MERGE ... RETURNING` is not implemented by DuckLake.
- DuckLake currently supports one update/delete action per MERGE statement. `delete_missing = TRUE` together with a `when_matched` action therefore runs as a MERGE plus a `DELETE` of the unmatched rows, wrapped in one transaction (a single snapshot). When you already opened a transaction, both statements simply join it.
- When several source rows share a `by` key, matched updates apply in an unspecified order; keep source keys unique.

Value

The number of affected rows, invisibly.

See Also

`rows_upsert()` for the plain update-or-insert case; `get_table_changes()` to inspect what a merge did.

Other row operations: `rows_delete()`, `rows_insert()`, `rows_update()`, `rows_upsert()`

Examples

```
lake_dir <- tempfile("merge_lake_")
dir.create(lake_dir)
attach_ducklake("merge_lake", lake_path = lake_dir)
create_table(data.frame(sls_id = 1:3, sls_amt = c(10, 20, 30)), "sales")

# Update only when the source amount is higher; insert new ids
new_sales <- data.frame(sls_id = c(2L, 4L), sls_amt = c(25, 40))
merge_into(
  get_ducklake_table("sales"), new_sales, by = "sls_id",
  matched_condition = "source.sls_amt > target.sls_amt"
)

# Synchronize to a staging table: upsert + drop rows gone from the source
```

```
staging_sales <- data.frame(sls_id = c(1L, 2L), sls_amt = c(11, 26))
merge_into("sales", staging_sales, by = "sls_id", delete_missing = TRUE)

# Remove rows flagged in the source
withdrawn <- data.frame(sls_id = 1L, sls_amt = 11)
merge_into(
  "sales", withdrawn, by = "sls_id",
  when_matched = "delete", when_not_matched = "nothing"
)

detach_ducklake("merge_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`plot_snapshots` *Plot the snapshot history of a table or lake*

Description

Draws snapshot history in one of two layouts. With a `table_name`, a commit-log timeline: one row per snapshot (newest at top) on an ordinal spine, with the timestamp, author, and commit message as aligned text and long idle stretches marked inline (e.g. "103 days later") instead of stretching an axis. Without a `table_name`, a lake-wide swimlane: one row per table, one point per snapshot, evenly spaced in snapshot order, so active and stale tables read at a glance.

Usage

```
plot_snapshots(table_name = NULL, ducklake_name = NULL, conn = NULL)
```

Arguments

`table_name`	The name of the table to plot. If NULL, plots all snapshots in the ducklake.
`ducklake_name`	The name of the ducklake (database) to query. If NULL, will attempt to infer from current database.
`conn`	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

Requires the ggplot2 package (listed in Suggests). Snapshot data comes from `list_table_snapshots()`; each snapshot is classified from its `changes` column into one of: created, schema change, data change, maintenance, or other. Authors and commit messages appear where they were recorded (see `set_snapshot_metadata()` and `commit_transaction()`).

Both layouts position snapshots by order rather than by clock time, so a history with months of silence between bursts of activity stays readable. In the swimlane, snapshots that touch no table (like the initial schema creation) appear in a `(lake)` lane, and the x axis labels show each snapshot's date.

Value

A ggplot object, which can be further customized with ggplot2 functions

See Also

Other time travel: get_ducklake_table_asof(), get_ducklake_table_version(), get_table_changes(), list_table_snapshots(), plot_table_changes(), restore_table_version()

Examples

```
lake_dir <- tempfile("plotsnap_lake_")
dir.create(lake_dir)
attach_ducklake("plotsnap_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

create_table(iris, "flowers")

# Commit-log timeline of one table's history
plot_snapshots("cars")

# Swimlane of every table in the lake
plot_snapshots()

# Customize the result like any ggplot
plot_snapshots("cars") +
  ggplot2::labs(title = "Audit trail")

detach_ducklake("plotsnap_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

plot_table_changes *Plot the rows changed in each snapshot of a table*

Description

Draws a table's change volume as a diverging bar chart: one bar per snapshot, with rows inserted or updated above the axis and rows deleted below it. A companion to plot_snapshots(), which shows *when* and *what kind* of changes happened; this shows *how much* changed each time.

Usage

```
plot_table_changes(table_name, ducklake_name = NULL, conn = NULL)
```

Arguments

table_name	The name of the table to plot.
ducklake_name	The name of the ducklake (database) to query. If NULL, will attempt to infer from current database.
conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

Requires the ggplot2 package (listed in Suggests). Row counts come from DuckLake's data change feed via get_table_changes(). An update appears in the feed as a before and an after image of the row; it is counted once here. Snapshots that touched the table without changing rows (a schema change, for example) keep their slot on the axis with no bar.

Value

A ggplot object, which can be further customized with ggplot2 functions

See Also

Other time travel: get_ducklake_table_asof(), get_ducklake_table_version(), get_table_changes(), list_table_snapshots(), plot_snapshots(), restore_table_version()

Examples

```
lake_dir <- tempfile("plotchg_lake_")
dir.create(lake_dir)
attach_ducklake("plotchg_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

rows_update(
  get_ducklake_table("cars"),
  data.frame(mpg = 21, gear = 5),
  by = "mpg"
)

# Rows inserted, updated, and deleted per snapshot
plot_table_changes("cars")

# Customize the result like any ggplot
plot_table_changes("cars") +
  ggplot2::theme_classic()

detach_ducklake("plotchg_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

plot_table_files *Plot the file layout of a lake*

Description

Draws each table's storage footprint as a horizontal bar: total bytes of Parquet data files, with delete files stacked in a second color, and a label giving the file count and average file size. Useful for spotting fragmentation – many small files – before it slows scans down.

Usage

```
plot_table_files(ducklake_name = NULL, conn = NULL)
```

Arguments

ducklake_name	The name of the ducklake (database) to query. If NULL, will attempt to infer from current database.
conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

Requires the ggplot2 package (listed in Suggests). File statistics come from `get_table_info()`. Tables whose rows are still inlined in the catalog have no data files yet and show an empty bar; run `flush_inlined_data()` to write them out. Many small files can be compacted with `merge_adjacent_files()`, and a large delete-file share is a sign to run `rewrite_data_files()`.

Value

A ggplot object, which can be further customized with ggplot2 functions

See Also

Other maintenance: `backup_ducklake()`, `checkpoint_ducklake()`, `cleanup_old_files()`, `delete_orphaned_files()`, `expire_snapshots()`, `flush_inlined_data()`, `get_table_info()`, `list_ducklake_files()`, `merge_adjacent_files()`, `rewrite_data_files()`

Examples

```
lake_dir <- tempfile("plotfiles_lake_")
dir.create(lake_dir)
attach_ducklake("plotfiles_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

create_table(iris, "flowers")

# File counts and sizes for every table in the lake
plot_table_files()
```

```
# Customize the result like any ggplot
plot_table_files() +
  ggplot2::theme_classic()

detach_ducklake("plotfiles_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

quack_query *Run a one-off query against a remote Quack server*

Description

Sends a single SQL query to a Quack server and returns the result as a data.frame. Unlike `attach_quack()`, this does not attach the remote catalog, so it is a quick way to pull a result without changing the session state.

Usage

```
quack_query(uri, query, token = NULL, disable_ssl = FALSE)
```

Arguments

`uri`	Address of the Quack server, for example `"quack:localhost"`.
`query`	A SQL query string to run on the server.
`token`	Authentication token expected by the server. If `NULL`, a Quack secret is used if one exists.
`disable_ssl`	Connect over plain HTTP instead of HTTPS (default `FALSE`).

Value

A data.frame with the query result.

See Also

`attach_quack()`

Other quack: `attach_quack()`, `detach_quack()`, `install_quack()`, `quack_serve()`, `quack_stop()`

Examples

```
## Not run:
quack_query(
  "quack:data.example.org",
  "SELECT count(*) FROM adsl",
  token = "super_secret"
)

## End(Not run)
```

quack_serve *Serve the current session over Quack*

Description

Starts a Quack server in the current DuckDB instance. Everything attached to the session, including a DuckLake attached with attach_ducklake(), becomes reachable by other DuckDB clients over the quack: protocol. The server runs in the background of the DuckDB instance, so the R session stays usable.

Usage

```
quack_serve(
  uri = "quack:localhost",
  token = NULL,
  allow_other_hostname = FALSE,
  disable_ssl = FALSE
)
```

Arguments

uri	Address to listen on (default "quack:localhost"). The default port is 9494.
token	Authentication token that clients must supply. If NULL, the server accepts any client that can reach it, and a warning is issued.
allow_other_hostname	Accept connections addressed to a hostname other than the one in uri (default FALSE).
disable_ssl	Serve over plain HTTP instead of HTTPS (default FALSE). Only appropriate on a trusted network.

Value

The server uri, invisibly.

See Also

quack_stop(), attach_quack()

Other quack: attach_quack(), detach_quack(), install_quack(), quack_query(), quack_stop()

Examples

```
## Not run:
attach_ducklake("trial", lake_path = "path/to/lake")
quack_serve(token = "super_secret")
# ... colleagues connect with attach_quack() ...
quack_stop()

## End(Not run)
```

quack_stop *Stop a Quack server*

Description

Stops a Quack server started with quack_serve().

Usage

```
quack_stop(uri = "quack:localhost")
```

Arguments

uri	Address the server is listening on (default "quack:localhost").

Value

TRUE, invisibly.

See Also

quack_serve()

Other quack: attach_quack(), detach_quack(), install_quack(), quack_query(), quack_serve()

Examples

```
## Not run:
quack_stop()

## End(Not run)
```

rename_ducklake_table *Rename a DuckLake table*

Description

Renames a table in place with ALTER TABLE ... RENAME TO, a metadata-only change.

Usage

```
rename_ducklake_table(from, to)
```

Arguments

from	Current table name.
to	New table name.

Details

History survives the rename, but snapshots from before it stay associated with the old name: `get_ducklake_table_version()` on the new name reaches back only as far as the rename, while the old name still serves the earlier snapshots.

Value

Invisibly returns `NULL`.

See Also

`rename_table_column()`

Other schema evolution: `add_table_column()`, `drop_table_column()`, `rename_table_column()`, `set_column_type()`

Examples

```
lake_dir <- tempfile("renametbl_lake_")
dir.create(lake_dir)
attach_ducklake("renametbl_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

rename_ducklake_table("cars", "cars_daily")

detach_ducklake("renametbl_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`rename_table_column` *Rename a column in a DuckLake table*

Description

Renames a column in place with `ALTER TABLE ... RENAME COLUMN`, a metadata-only change.

Usage

```
rename_table_column(table_name, from, to)
```

Arguments

`table_name`	The table to change.
`from`	Current column name.
`to`	New column name.

Value

Invisibly returns `NULL`.

See Also

add_table_column(), drop_table_column(), rename_ducklake_table()

Other schema evolution: add_table_column(), drop_table_column(), rename_ducklake_table(), set_column_type()

Examples

```
lake_dir <- tempfile("renamecol_lake_")
dir.create(lake_dir)
attach_ducklake("renamecol_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

rename_table_column("cars", from = "mpg", to = "miles_per_gallon")

detach_ducklake("renamecol_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

replace_table *Replace a table with modified data and create a new snapshot*

Description

Replace a table with modified data and create a new snapshot

Usage

```
replace_table(.data, table_name, .quiet = TRUE)
```

Arguments

.data	A dplyr query object (tbl_lazy) with transformations
table_name	Table name to replace
.quiet	Logical, whether to suppress messages (default TRUE)

Details

This function is designed for schema changes or bulk transformations that should create a new versioned snapshot. It:

1. Collects the transformed data
2. Drops the existing table
3. Creates a new table with the updated schema/data

The drop and create run atomically: when no transaction is open, `replace_table()` wraps them in one of its own, so a failed create never leaves the table dropped. Wrap the call in `with_transaction()` (or `begin_transaction()`/`commit_transaction()`) when you want to record an author and commit message on the snapshot, or to group the replacement with other changes.

When to use replace_table():

- **Bulk transformations** - a dplyr pipeline that recomputes, reshapes, or filters most of the table

When to reach elsewhere:

- **Schema-only changes** - `add_table_column()`, `drop_table_column()`, `rename_table_column()`, and `set_column_type()` alter the table in place; nothing is collected or rewritten
- **Derived columns** - `add_table_column()` followed by a `mutate()` pipeline through `ducklake_exec()` fills the new column with an in-database UPDATE
- **Targeted row changes** - `rows_update()`, `rows_upsert()`, or `ducklake_exec()` modify only the affected rows

Both paths create a snapshot: replace_table() via DROP + CREATE, and ducklake_exec() via the in-place UPDATE/DELETE it runs, so either way the change is available for time travel.

Value

Invisibly returns NULL

See Also

Other table operations: `add_data_files()`, `create_table()`, `create_view()`, `drop_view()`, `ducklake_exec()`, `get_ducklake_table()`, `get_metadata_table()`, `list_ducklake_tables()`, `show_ducklake_query()`

Examples

```
lake_dir <- tempfile("replace_lake_")
dir.create(lake_dir)
attach_ducklake("replace_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Add new derived columns (atomic on its own; creates a new snapshot)
get_ducklake_table("cars") |>
  dplyr::mutate(
    thirsty = dplyr::if_else(mpg < 20, "Y", "N"),
    mpg_band = dplyr::case_when(
      mpg < 15 ~ "<15",
      mpg < 25 ~ "15-24",
      TRUE ~ ">=25"
    )
  ) |>
  replace_table("cars")

# Wrap in with_transaction() to record audit metadata on the snapshot
with_transaction(
```

```
  get_ducklake_table("cars") |>
    dplyr::select(-thirsty, -mpg_band) |>
    replace_table("cars"),
  author = "Data Engineer",
  commit_message = "Drop derived columns"
)

detach_ducklake("replace_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`reset_table_partitioning` *Remove partitioning keys from a table*

Description

Clears a table's partitioning keys so newly written data files are no longer split along them. Existing files are unaffected.

Usage

```
reset_table_partitioning(table_name)
```

Arguments

`table_name`	The name of the table.

Value

Invisibly returns `NULL`.

See Also

`set_table_partitioning()`, `get_table_partitions()`

Other partitioning: `get_table_partitions()`, `set_table_partitioning()`

Examples

```
lake_dir <- tempfile("unpart_lake_")
dir.create(lake_dir)
attach_ducklake("unpart_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

set_table_partitioning("cars", "cyl")
reset_table_partitioning("cars")

detach_ducklake("unpart_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`reset_table_sorting` *Remove the sort order from a table*

Description

Clears a table's declared sort order so newly written data files are no longer sorted. Existing files are unaffected.

Usage

```
reset_table_sorting(table_name)
```

Arguments

`table_name`	The name of the table.

Value

Invisibly returns `NULL`.

See Also

`set_table_sorting()`

Other sorting: `set_table_sorting()`

Examples

```
lake_dir <- tempfile("unsort_lake_")
dir.create(lake_dir)
attach_ducklake("unsort_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

set_table_sorting("cars", "mpg")
reset_table_sorting("cars")

detach_ducklake("unsort_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`restore_table_version` *Restore a table to a previous version*

Description

Rolls a table back to the state it had at an earlier snapshot or point in time, by recreating it from a time-travel read of itself. History is preserved: the restore is recorded as a **new** snapshot (with a commit message noting the restore), so nothing is rewritten or lost and you can still time-travel to any snapshot, including those after the restore point.

Usage

```
restore_table_version(
  table_name,
  version = NULL,
  timestamp = NULL,
  author = NULL,
  commit_message = NULL,
  conn = NULL
)
```

Arguments

`table_name`	The name of the table to restore
`version`	Optional snapshot id to restore to (see `list_table_snapshots()`)
`timestamp`	Optional timestamp to restore to (POSIXct, converted to UTC, or character already in UTC)
`author`	Optional author to record on the restore snapshot, for the audit trail
`commit_message`	Optional commit message for the restore snapshot. Defaults to a message noting the restore point (e.g. `"Restored my_table to snapshot 5"`).
`conn`	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

You must specify either `version` or `timestamp`, but not both.

Under the hood this runs `CREATE OR REPLACE TABLE t AS SELECT * FROM t AT (VERSION => n)` inside a transaction. Because the restore creates a new snapshot, it is itself reversible with another `restore_table_version()` call.

Value

Invisibly returns TRUE on success

See Also

get_ducklake_table_version(), get_ducklake_table_asof(), list_table_snapshots()

Other time travel: get_ducklake_table_asof(), get_ducklake_table_version(), get_table_changes(), list_table_snapshots(), plot_snapshots(), plot_table_changes()

Examples

```
lake_dir <- tempfile("restore_lake_")
dir.create(lake_dir)
attach_ducklake("restore_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, amount = c(10, 20, 30)), "orders")

rows_delete(
  get_ducklake_table("orders"),
  data.frame(id = 1L),
  by = "id"
)
snapshots <- list_table_snapshots("orders")
first_version <- snapshots$snapshot_id[1]

# Roll the table back to its first snapshot
restore_table_version("orders", version = first_version)

# Record who performed the restore in the audit trail
restore_table_version(
  "orders",
  version = first_version,
  author = "Data Steward",
  commit_message = "Roll back erroneous bulk update"
)

detach_ducklake("restore_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

rewrite_data_files *Rewrite data files with many deletes*

Description

Rewrites Parquet files whose rows have mostly been deleted. Deletes in DuckLake are recorded in separate delete files; heavily-deleted data files slow reads down until they are rewritten without the dead rows.

Usage

```
rewrite_data_files(
  ducklake_name = NULL,
```

```
  table_name = NULL,
  delete_threshold = NULL
)
```

Arguments

ducklake_name	Name of the attached DuckLake catalog. If NULL, the current database is used.
table_name	Optional table name. When provided, only that table's files are rewritten.
delete_threshold	Optional fraction of deleted rows (between 0 and 1). above which a file is rewritten. DuckLake's default is 0.95.

Details

The rewritten originals are scheduled for deletion once no snapshot references them; run cleanup_old_files() to remove them.

Value

A data frame with one row per output file (columns schema_name, table_name, files_processed, files_created).

See Also

cleanup_old_files(), checkpoint_ducklake()

Other maintenance: backup_ducklake(), checkpoint_ducklake(), cleanup_old_files(), delete_orphaned_files(), expire_snapshots(), flush_inlined_data(), get_table_info(), list_ducklake_files(), merge_adjacent_files(), plot_table_files()

Examples

```
lake_dir <- tempfile("rewrite_lake_")
dir.create(lake_dir)
attach_ducklake("rewrite_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

rows_delete(
  get_ducklake_table("cars"),
  data.frame(gear = 3),
  by = "gear"
)

# Rewrite any file that is at least half deleted
rewrite_data_files("rewrite_lake", delete_threshold = 0.5)

# Just one table, with DuckLake's default threshold
rewrite_data_files(table_name = "cars")

detach_ducklake("rewrite_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`rollback_transaction` *Rollback a transaction*

Description

Rolls back the current transaction, discarding all changes made since the transaction began.

Usage

```
rollback_transaction(conn = NULL)
```

Arguments

conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

This function discards all changes made since `begin_transaction()` was called, reverting the database to its state before the transaction began.

Value

Invisibly returns TRUE on success

See Also

Other transactions: `begin_transaction()`, `commit_transaction()`, `set_snapshot_metadata()`, `with_transaction()`

Examples

```
lake_dir <- tempfile("rollback_lake_")
dir.create(lake_dir)
attach_ducklake("rollback_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

begin_transaction()
rows_delete(
  get_ducklake_table("cars"),
  data.frame(gear = 3),
  by = "gear"
)

# Something went wrong, rollback
rollback_transaction()

detach_ducklake("rollback_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

rows_delete *Delete rows from a DuckLake table*

Description

A wrapper around dplyr::rows_delete() with in_place = TRUE as the default, since DuckLake is designed for in-place modifications.

Usage

```
rows_delete(
  x,
  y,
  by = NULL,
  copy = TRUE,
  in_place = TRUE,
  unmatched = "ignore",
  ...
)
```

Arguments

x	Target table (from get_ducklake_table())
y	Data frame with rows to delete (matched by 'by' columns)
by	Column(s) to match on
copy	Whether to copy y to the same source as x (default TRUE)
in_place	Whether to modify the table in place (default TRUE for DuckLake)
unmatched	How to handle unmatched rows (default "ignore")
...	Additional arguments passed to dplyr::rows_delete()

Details

Choosing how to change a table:

- To look up or combine data for analysis, use dplyr joins (`left_join()` and friends). Joins read from the lake and build a new result; they never modify a lake table.
- To append, correct, or remove specific rows, use `rows_insert()`, `rows_update()`, or `rows_delete()`. Each call is a single SQL statement against the existing table – no data leaves the database, and with data inlining enabled (DuckLake's default) small changes land in the catalog without creating tiny Parquet files.
- To update rows that exist and insert the ones that don't in one atomic statement, use `rows_upsert()`.
- For conditional merge logic or deletes driven by a staging table, use `merge_into()`.
- To change a table's shape without touching its data – add, drop, or rename columns, widen a type – use the schema evolution family (`add_table_column()` and friends): metadata-only changes that rewrite nothing.

- For bulk transformations that touch most rows, use `replace_table()`. It collects the transformed data into R and rewrites the whole table – heavier than the row operations, and it resets the row lineage that the in-place operations preserve in the change feed.

Value

The updated table

See Also

Other row operations: `merge_into()`, `rows_insert()`, `rows_update()`, `rows_upsert()`

Examples

```
lake_dir <- tempfile("rowsdel_lake_")
dir.create(lake_dir)
attach_ducklake("rowsdel_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, value = c("a", "b", "c")), "items")

rows_delete(
  get_ducklake_table("items"),
  data.frame(id = c(1, 2)),
  by = "id"
)

detach_ducklake("rowsdel_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

rows_insert *Insert rows into a DuckLake table*

Description

A wrapper around dplyr::rows_insert() with in_place = TRUE as the default, since DuckLake is designed for in-place modifications.

Usage

```
rows_insert(
  x,
  y,
  by = NULL,
  copy = TRUE,
  in_place = TRUE,
  conflict = "ignore",
  ...
)
```

Arguments

x	Target table (from get_ducklake_table())
y	Data frame with new rows
by	Column(s) to match on (for conflict detection)
copy	Whether to copy y to the same source as x (default TRUE)
in_place	Whether to modify the table in place (default TRUE for DuckLake)
conflict	How to handle conflicts (default "ignore")
...	Additional arguments passed to dplyr::rows_insert()

Details

Choosing how to change a table:

- To look up or combine data for analysis, use dplyr joins (`left_join()` and friends). Joins read from the lake and build a new result; they never modify a lake table.
- To append, correct, or remove specific rows, use `rows_insert()`, `rows_update()`, or `rows_delete()`. Each call is a single SQL statement against the existing table – no data leaves the database, and with data inlining enabled (DuckLake's default) small changes land in the catalog without creating tiny Parquet files.
- To update rows that exist and insert the ones that don't in one atomic statement, use `rows_upsert()`.
- For conditional merge logic or deletes driven by a staging table, use `merge_into()`.
- To change a table's shape without touching its data – add, drop, or rename columns, widen a type – use the schema evolution family (`add_table_column()` and friends): metadata-only changes that rewrite nothing.
- For bulk transformations that touch most rows, use `replace_table()`. It collects the transformed data into R and rewrites the whole table – heavier than the row operations, and it resets the row lineage that the in-place operations preserve in the change feed.

Value

The updated table

See Also

Other row operations: `merge_into()`, `rows_delete()`, `rows_update()`, `rows_upsert()`

Examples

```
lake_dir <- tempfile("rowsins_lake_")
dir.create(lake_dir)
attach_ducklake("rowsins_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, value = c("a", "b", "c")), "items")

rows_insert(
  get_ducklake_table("items"),
  data.frame(id = 99, value = "new row"),
  by = "id"
)
```

```
detach_ducklake("rowsins_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`rows_update` *Update rows in a DuckLake table*

Description

A wrapper around dplyr::rows_update() with in_place = TRUE as the default, since DuckLake is designed for in-place modifications.

Usage

```
rows_update(
  x,
  y,
  by = NULL,
  copy = TRUE,
  in_place = TRUE,
  unmatched = "ignore",
  ...
)
```

Arguments

`x`	Target table (from get_ducklake_table())
`y`	Data frame with updates
`by`	Column(s) to match on
`copy`	Whether to copy y to the same source as x (default TRUE)
`in_place`	Whether to modify the table in place (default TRUE for DuckLake)
`unmatched`	How to handle unmatched rows (default "ignore")
`...`	Additional arguments passed to dplyr::rows_update()

Details

Choosing how to change a table:

- To look up or combine data for analysis, use dplyr joins (`left_join()` and friends). Joins read from the lake and build a new result; they never modify a lake table.
- To append, correct, or remove specific rows, use `rows_insert()`, `rows_update()`, or `rows_delete()`. Each call is a single SQL statement against the existing table – no data leaves the database, and with data inlining enabled (DuckLake's default) small changes land in the catalog without creating tiny Parquet files.
- To update rows that exist and insert the ones that don't in one atomic statement, use `rows_upsert()`.

- For conditional merge logic or deletes driven by a staging table, use `merge_into()`.
- To change a table's shape without touching its data – add, drop, or rename columns, widen a type – use the schema evolution family (`add_table_column()` and friends): metadata-only changes that rewrite nothing.
- For bulk transformations that touch most rows, use `replace_table()`. It collects the transformed data into R and rewrites the whole table – heavier than the row operations, and it resets the row lineage that the in-place operations preserve in the change feed.

Value

The updated table

See Also

Other row operations: `merge_into()`, `rows_delete()`, `rows_insert()`, `rows_upsert()`

Examples

```
lake_dir <- tempfile("rowsupd_lake_")
dir.create(lake_dir)
attach_ducklake("rowsupd_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, value = c("a", "b", "c")), "items")

# Update rows - in_place = TRUE by default
rows_update(
  get_ducklake_table("items"),
  data.frame(id = 1, value = "new"),
  by = "id"
)

detach_ducklake("rowsupd_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`rows_upsert` *Upsert rows into a DuckLake table*

Description

Updates rows of `x` that match a row of `y` (by the `by` columns) and inserts the rows of `y` that have no match, as one atomic `MERGE INTO` statement – a single snapshot, with row lineage preserved in the change feed. A wrapper around dplyr::rows_upsert() with in_place = TRUE as the default, since DuckLake is designed for in-place modifications.

Usage

```
rows_upsert(x, y, by = NULL, copy = TRUE, in_place = TRUE, ...)
```

Arguments

x	Target table (from get_ducklake_table())
y	Data frame with rows to update or insert
by	Column(s) to match on. Defaults to the first column of y, with a message.
copy	Whether to copy y to the same source as x (default TRUE)
in_place	Whether to modify the table in place (default TRUE for DuckLake)
...	Additional arguments passed to dplyr::rows_upsert()

Details

Choosing how to change a table:

- To look up or combine data for analysis, use dplyr joins (`left_join()` and friends). Joins read from the lake and build a new result; they never modify a lake table.
- To append, correct, or remove specific rows, use `rows_insert()`, `rows_update()`, or `rows_delete()`. Each call is a single SQL statement against the existing table – no data leaves the database, and with data inlining enabled (DuckLake's default) small changes land in the catalog without creating tiny Parquet files.
- To update rows that exist and insert the ones that don't in one atomic statement, use `rows_upsert()`.
- For conditional merge logic or deletes driven by a staging table, use `merge_into()`.
- To change a table's shape without touching its data – add, drop, or rename columns, widen a type – use the schema evolution family (`add_table_column()` and friends): metadata-only changes that rewrite nothing.
- For bulk transformations that touch most rows, use `replace_table()`. It collects the transformed data into R and rewrites the whole table – heavier than the row operations, and it resets the row lineage that the in-place operations preserve in the change feed.

DuckLake-specific behavior:

DuckLake tables have no primary keys or unique constraints, so the usual database upsert (`INSERT ... ON CONFLICT`) does not apply. This method instead generates `MERGE INTO`, matching on the `by` columns:

- When `y` covers a subset of `x`'s columns, matched rows are updated in those columns only; inserted rows receive the column's default value in the remaining columns (`NULL` when the table defines none). Note the difference from data-frame upserts, which fill with `NA`.
- When `y` has only the `by` columns, there is nothing to update and the call inserts the unmatched rows.
- Rows where a `by` column is `NULL` never match and are always inserted.
- When several rows of `y` share the same `by` key, each matched update applies in an unspecified order; the last write wins. Keep `y` keys unique.

Value

The updated table, invisibly

See Also

`merge_into()` for conditional merge clauses and source-driven deletes.

Other row operations: `merge_into()`, `rows_delete()`, `rows_insert()`, `rows_update()`

Examples

```
lake_dir <- tempfile("rowsups_lake_")
dir.create(lake_dir)
attach_ducklake("rowsups_lake", lake_path = lake_dir)
create_table(data.frame(id = 1:3, value = c("a", "b", "c")), "items")

# Update id 2, insert id 4 - one statement, one snapshot
rows_upsert(
  get_ducklake_table("items"),
  data.frame(id = c(2, 4), value = c("updated", "new")),
  by = "id"
)

detach_ducklake("rowsups_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

set_column_comments *Set column comments on a DuckLake table*

Description

Stores per-column descriptions in the DuckLake catalog with `COMMENT ON COLUMN`, one `name = "comment"` pair per column. For data with haven/labelled variable labels, `create_table()` can store the labels automatically; this function adds or revises them afterward – for example to label a derived variable.

Usage

```
set_column_comments(table_name, ...)
```

Arguments

`table_name`	The table whose columns to describe.
`...`	Named comments, e.g. `USUBJID = "Unique subject identifier"`. Use `NA` (or `NULL`) as a value to clear that column's comment. To pass a named vector built elsewhere, splice it with `do.call(set_column_comments, c(list("tbl"), as.list(my_labels)))`.

Details

All comments from one call are written in a single transaction, so they land as one snapshot. Inside `with_transaction()` they join the open transaction instead.

Value

Invisibly returns `NULL`.

See Also

set_table_comment(), get_table_comments()

Other table documentation: get_table_comments(), set_table_comment()

Examples

```
lake_dir <- tempfile("colcomment_lake_")
dir.create(lake_dir)
attach_ducklake("colcomment_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

set_column_comments(
  "cars",
  mpg = "Miles per US gallon",
  cyl = "Number of cylinders",
  disp = NA # clear this one
)

detach_ducklake("colcomment_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

set_column_type *Change the type of a DuckLake table column*

Description

Changes a column's type in place with `ALTER TABLE ... ALTER COLUMN ... SET TYPE`. DuckLake permits widening promotions only (for example `INTEGER` to `BIGINT`, or `FLOAT` to `DOUBLE`): every existing value must be representable in the new type, so no data can be lost and no data files need rewriting.

Usage

```
set_column_type(table_name, column_name, type)
```

Arguments

table_name	The table to change.
column_name	Name of the column.
type	The new (wider) SQL type.

Details

To *narrow* a type, which DuckLake refuses, take the explicit route: add_table_column() with the smaller type, copy the values over (after checking they fit), drop_table_column() the original, and rename_table_column() the new column into place.

Value

Invisibly returns NULL.

See Also

add_table_column(), drop_table_column(), rename_table_column()

Other schema evolution: add_table_column(), drop_table_column(), rename_ducklake_table(), rename_table_column()

Examples

```
lake_dir <- tempfile("coltype_lake_")
dir.create(lake_dir)
attach_ducklake("coltype_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Widening promotions only: INTEGER -> BIGINT is fine
add_table_column("cars", "order_id", "INTEGER")
set_column_type("cars", "order_id", "BIGINT")

detach_ducklake("coltype_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

set_ducklake_connection

Use your own DuckDB connection with ducklake

Description

By default, ducklake creates and manages its own DuckDB connection. Call this function to make ducklake use a connection you have created instead – for example, a connection you share with duckplyr or other DBI-based tools, or one configured with custom DuckDB settings.

Usage

```
set_ducklake_connection(conn)
```

Arguments

conn	A live DuckDB connection created with DBI::dbConnect() and duckdb::duckdb().

Details

ducklake never closes a connection you supply: detach_ducklake() with shutdown = TRUE and the end-of-session cleanup only shut down connections that ducklake created itself. Closing your connection remains your responsibility.

If ducklake was already managing its own connection, that connection is shut down before yours is registered.

Value

The connection, invisibly.

See Also

`get_ducklake_connection()`

Other connection management: `attach_ducklake()`, `create_storage_secret()`, `detach_ducklake()`, `ducklake_extension_available()`, `get_ducklake_backend()`, `get_ducklake_connection()`, `install_ducklake()`

Examples

```
db_file <- tempfile(fileext = ".duckdb")
lake_dir <- tempfile("own_conn_lake_")
dir.create(lake_dir)

conn <- DBI::dbConnect(duckdb::duckdb(), dbdir = db_file)
set_ducklake_connection(conn)
attach_ducklake("own_conn_lake", lake_path = lake_dir)

# A connection you registered is yours to close
detach_ducklake("own_conn_lake")
DBI::dbDisconnect(conn, shutdown = TRUE)
unlink(c(db_file, lake_dir), recursive = TRUE)
```

---

set_ducklake_option *Set a DuckLake option*

Description

Sets a DuckLake configuration option, either lake-wide or scoped to a schema or table. Options are persisted in the metadata catalog, so they survive detach/attach cycles and apply to every client of the lake.

Usage

```
set_ducklake_option(
  option,
  value,
  table_name = NULL,
  schema_name = NULL,
  ducklake_name = NULL
)
```

Arguments

option	Name of the option, e.g. "parquet_compression", "target_file_size", "sort_on_insert", or "data_inlining_row_limit". See https://ducklake.select/docs/stable/duckdb/usage/configuration for the full list.
value	The value to set. Logicals are rendered as true/false, numbers as numeric literals, and everything else as a quoted string.
table_name	Optional table name to scope the option to one table.
schema_name	Optional schema name to scope the option to one schema (or, together with table_name, to qualify the table).
ducklake_name	Optional name of the attached DuckLake catalog. If NULL, the current database is used.

Details

Table-scoped settings override schema-scoped ones, which override the lake-wide default. Runs CALL <lake>.set_option(...).

Commonly tuned options include parquet_compression (default "snappy"; "zstd" trades write speed for smaller files), target_file_size (default "512MB"), sort_on_insert (default TRUE; see set_table_sorting()), and require_commit_message (default FALSE).

Value

Invisibly returns NULL.

See Also

get_ducklake_options(), set_inlining_row_limit()

Other options: get_ducklake_options()

Examples

```
lake_dir <- tempfile("setopt_lake_")
dir.create(lake_dir)
attach_ducklake("setopt_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Smaller files at some write cost, lake-wide
set_ducklake_option("parquet_compression", "zstd")

# Skip one table during compaction
set_ducklake_option("auto_compact", FALSE, table_name = "cars")

# Make every snapshot carry a commit message (constrains later writes)
set_ducklake_option("require_commit_message", TRUE)

detach_ducklake("setopt_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`set_inlining_row_limit` *Set the data inlining row limit*

Description

Controls the threshold below which DuckLake stores small inserts and deletes directly in the catalog database instead of writing Parquet files. This avoids the "small files problem" common in streaming or frequent-update workloads.

Usage

```
set_inlining_row_limit(
  limit,
  table_name = NULL,
  schema_name = NULL,
  ducklake_name = NULL
)
```

Arguments

`limit`	Integer. The maximum number of rows that will be inlined. Set to `0` to disable inlining entirely.
`table_name`	Optional table name. When provided the limit is persisted for that table in the DuckLake metadata (takes priority over the global setting).
`schema_name`	Optional schema name. When provided (without `table_name`) the limit is persisted for all tables in that schema.
`ducklake_name`	Optional name of the attached DuckLake catalog. Required when setting a table- or schema-level override. If `NULL`, the current database is used.

Details

The limit can be set at three levels (highest priority first):

1. **Table-level** – persisted in the DuckLake metadata for a specific table
2. **Schema-level** – persisted for all tables in a schema
3. **Global (DuckDB setting)** – applies to all DuckLake connections

Data inlining is enabled by default in DuckLake v1.0 with a threshold of 10 rows. Any insert or delete affecting fewer rows than the limit is written to an inlined table inside the catalog instead of creating a Parquet file.

For streaming or high-frequency-insert workloads, increase the limit (e.g., 50 or 100). For workloads that always write large batches, the default is fine or you can disable inlining with `limit = 0`.

Use `flush_inlined_data()` or `checkpoint_ducklake()` to materialise inlined data to Parquet when ready.

Value

Invisibly returns NULL.

See Also

get_inlining_row_limit(), flush_inlined_data(), checkpoint_ducklake()

Other data inlining: checkpoint_ducklake(), flush_inlined_data(), get_inlining_row_limit()

Examples

```
lake_dir <- tempfile("inline_set_lake_")
dir.create(lake_dir)
attach_ducklake("inline_set_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Change the global default
set_inlining_row_limit(50)

# Override for a specific table
set_inlining_row_limit(100, table_name = "cars")

# Disable inlining globally
set_inlining_row_limit(0)

detach_ducklake("inline_set_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

set_snapshot_metadata *Set metadata for the most recent snapshot*

Description

Sets the author, commit message, and/or extra info for the most recent snapshot in a DuckLake catalog by updating the ducklake_snapshot_changes metadata table directly.

Usage

```
set_snapshot_metadata(
  ducklake_name,
  author = NULL,
  commit_message = NULL,
  commit_extra_info = NULL,
  conn = NULL
)
```

Arguments

ducklake_name	The name of the DuckLake catalog
author	Optional author name to associate with the snapshot
commit_message	Optional commit message describing the changes
commit_extra_info	Optional extra information about the commit
conn	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

This function retroactively updates metadata on the most recent snapshot. To set metadata at commit time, use the author, commit_message, and commit_extra_info arguments in commit_transaction() or with_transaction() instead.

Value

Invisibly returns TRUE on success

See Also

Other transactions: begin_transaction(), commit_transaction(), rollback_transaction(), with_transaction()

Examples

```
lake_dir <- tempfile("meta_lake_")
dir.create(lake_dir)
attach_ducklake("meta_lake", lake_path = lake_dir)

begin_transaction()
create_table(mtcars, "cars")
commit_transaction()

# Add metadata to the snapshot after the fact
set_snapshot_metadata(
  ducklake_name = "meta_lake",
  author = "Data Team",
  commit_message = "Added the cars dataset"
)

detach_ducklake("meta_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`set_table_comment` *Set the comment on a DuckLake table*

Description

Stores a description of the table in the DuckLake catalog with `COMMENT ON TABLE`. Comments live in the lake itself, so every client – R, Python, or plain SQL – sees the same documentation, and AI tools reading the catalog get the context too.

Usage

```
set_table_comment(table_name, comment)
```

Arguments

`table_name`	The table to describe.
`comment`	The comment text, or `NULL`/`NA` to clear an existing comment.

Value

Invisibly returns `NULL`.

See Also

`set_column_comments()`, `get_table_comments()`

Other table documentation: `get_table_comments()`, `set_column_comments()`

Examples

```
lake_dir <- tempfile("comment_lake_")
dir.create(lake_dir)
attach_ducklake("comment_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

set_table_comment("cars", "Motor Trend road tests, one row per model")
set_table_comment("cars", NULL) # clear

detach_ducklake("comment_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

set_table_partitioning *Set partitioning keys for a table*

Description

Declares how newly written data files for a table should be split up. Partitioning lets DuckLake prune files during query planning, which can speed up filtered reads on large tables considerably.

Usage

```
set_table_partitioning(table_name, partition_by)
```

Arguments

table_name	The name of the table to partition.
partition_by	Character vector of partition expressions. Each entry must be one of: • a column name, e.g. "region" (identity transform) • "year(col)", "month(col)", "day(col)", or "hour(col)" for timestamp columns • "bucket(n, col)" for hash bucketing into n buckets

Details

Partitioning only affects data written *after* the keys are set; previously written files keep their layout. To re-partition existing data, rewrite the table (e.g. with replace_table()) after setting the keys.

Runs ALTER TABLE ... SET PARTITIONED BY (...). The expressions are validated against the transforms DuckLake supports before any SQL is built.

Value

Invisibly returns NULL.

See Also

reset_table_partitioning(), get_table_partitions()

Other partitioning: get_table_partitions(), reset_table_partitioning()

Examples

```
lake_dir <- tempfile("part_lake_")
dir.create(lake_dir)
attach_ducklake("part_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Plain column partitioning
set_table_partitioning("cars", "cyl")
```

```
# Compound key
set_table_partitioning("cars", c("gear", "cyl"))

# Timestamp columns can be split by year(), month(), day(), or hour()

detach_ducklake("part_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`set_table_sorting` *Set the sort order of a table*

Description

Declares how a table's data files should be sorted. DuckLake sorts data on insert (unless the `sort_on_insert` option is disabled), during compaction with `merge_adjacent_files()`, and when flushing inlined data with `flush_inlined_data()`. Sorted files carry tighter min/max statistics, so filters on the sort columns prune files instead of scanning them – the complement to `set_table_partitioning()` for high-cardinality columns.

Usage

```
set_table_sorting(table_name, sort_by)
```

Arguments

`table_name`	The name of the table to sort.
`sort_by`	Character vector of sort keys. Each entry is a column name, optionally followed by `ASC` or `DESC` and by `NULLS FIRST` or `NULLS LAST`, e.g. `"event_time DESC"` or `"id ASC NULLS LAST"`.

Details

Runs `ALTER TABLE ... SET SORTED BY (...)`. Only newly written files are sorted; existing files keep their layout until compaction rewrites them.

DuckLake also accepts arbitrary SQL expressions as sort keys; this wrapper deliberately accepts only column-based keys so the input can be validated. For expression keys, run the `ALTER TABLE` statement directly with `DBI::dbExecute()`.

To keep insert speed and sort the files only at compaction time, disable sorting on insert with `set_ducklake_option("sort_on_insert", FALSE, table_name = ...)`.

Value

Invisibly returns `NULL`.

See Also

reset_table_sorting(), set_ducklake_option(), set_table_partitioning()

Other sorting: reset_table_sorting()

Examples

```
lake_dir <- tempfile("sort_lake_")
dir.create(lake_dir)
attach_ducklake("sort_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Order rows so range filters can prune files
set_table_sorting("cars", "mpg")

# Compound key with explicit directions
set_table_sorting("cars", c("cyl ASC", "mpg DESC"))

detach_ducklake("sort_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

---

show_ducklake_query *Show the SQL that would be executed by ducklake operations*

Description

This function shows the SQL that would be generated and executed by ducklake. This is useful for debugging and understanding what SQL is being sent to DuckDB.

Usage

```
show_ducklake_query(.data, table_name = NULL)
```

Arguments

.data	A dplyr query object (tbl_lazy)
table_name	The target table name for the operation. If not provided, will be extracted from the table attribute (set by get_ducklake_table())

Value

The first argument, invisibly (following show_query convention)

See Also

Other table operations: add_data_files(), create_table(), create_view(), drop_view(), ducklake_exec(), get_ducklake_table(), get_metadata_table(), list_ducklake_tables(), replace_table()

Examples

```
lake_dir <- tempfile("sql_lake_")
dir.create(lake_dir)
attach_ducklake("sql_lake", lake_path = lake_dir)
create_table(mtcars, "cars")

# Show SQL for an update operation (table name inferred)
get_ducklake_table("cars") |>
  dplyr::mutate(gear = 5) |>
  show_ducklake_query()

detach_ducklake("sql_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

`with_transaction` *Execute code within a transaction*

Description

Wraps code execution in a transaction, automatically committing on success or rolling back on error. This provides a more R-idiomatic and safer way to handle transactions compared to manually calling `begin_transaction()` and `commit_transaction()`.

Usage

```
with_transaction(
  expr,
  author = NULL,
  commit_message = NULL,
  commit_extra_info = NULL,
  conn = NULL
)
```

Arguments

`expr`	An R expression or code block to execute within the transaction. Can be a single statement or a `{...}` block containing multiple statements.
`author`	Optional author name to associate with the snapshot
`commit_message`	Optional commit message describing the changes
`commit_extra_info`	Optional extra information about the commit
`conn`	Optional DuckDB connection object. If not provided, uses the default ducklake connection.

Details

This function provides automatic error handling and cleanup for transactions:

- Begins a transaction before executing the code
- Executes the provided expression
- On success: commits the transaction and adds metadata (if provided)
- On error: automatically rolls back the transaction and re-throws the error

This pattern is similar to `withr::with_*()` functions and provides better safety guarantees than manually managing transactions.

Value

Invisibly returns the result of the expression

See Also

Other transactions: `begin_transaction()`, `commit_transaction()`, `rollback_transaction()`, `set_snapshot_metadata()`

Examples

```
lake_dir <- tempfile("with_tx_lake_")
dir.create(lake_dir)
attach_ducklake("with_tx_lake", lake_path = lake_dir)

# Single operation
with_transaction(
  create_table(mtcars, "cars"),
  author = "Data Team",
  commit_message = "Add cars dataset"
)

# Multiple operations in a block
with_transaction({
  create_table(iris, "flowers")
  create_table(airquality, "air")
}, author = "Data Team", commit_message = "Add datasets")

# With dplyr pipeline
with_transaction(
  get_ducklake_table("cars") |>
    dplyr::mutate(kpl = mpg * 0.425144) |>
    replace_table("cars"),
  author = "Data Team",
  commit_message = "Add km/L column"
)

# Automatic rollback on error
tryCatch(
  with_transaction({
```

```
    create_table(ChickWeight, "chicks")
    stop("Simulated error") # Transaction will be rolled back
  }),
  error = function(e) message("Transaction was rolled back: ", e$message)
)

# "chicks" was never committed
list_ducklake_tables()

detach_ducklake("with_tx_lake", shutdown = TRUE)
unlink(lake_dir, recursive = TRUE)
```

Index