Package 'SCDB'

January 23, 2026

Type Package

Title Easily Access and Maintain Time-Based Versioned Data (Slowly-Changing-Dimension)

Version 0.6.0

Description
A collection of functions that enable easy access and updating of a database of data over time. More specifically, the package facilitates type-2 history for data-warehouses and provides a number of Quality of life improvements for working on SQL databases with R. For reference see Ralph Kimball and Margy Ross (2013, ISBN 9781118530801).

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 3.6.0)

Imports checkmate, DBI, dbplyr (>= 2.4.0), dplyr, glue, lubridate, methods, openssl, parallelly, purrr, rlang, R6, stringr, tidyr, tidyselect, utils, magrittr

Suggests callr, conflicted, devtools, duckdb (>= 0.10.1), ggplot2, here, jsonlite, knitr, lintr, microbenchmark, odbc, pak, rmarkdown, roxygen2, pkgdown, RPostgres, RSQLite, spelling, testthat (>= 3.0.0), tibble, tidyverse, withr

Language en-US

URL https://github.com/ssi-dk/SCDB, https://ssi-dk.github.io/SCDB/

Config/testthat/edition 3

BugReports https://github.com/ssi-dk/SCDB/issues

VignetteBuilder knitr

NeedsCompilation no

Author Rasmus Skytte Randløv [aut, cre, rev] (ORCID: <https://orcid.org/0000-0002-5860-3838>), Marcus Munch Grünewald [aut] (ORCID:

<https://orcid.org/0009-0006-8090-406X>),
Lasse Engbo Christiansen [rev, ctb] (ORCID:
<https://orcid.org/0000-0001-5019-1931>),
Sofia Myrup Otero [rev],
Kim Daniel Jacobsen [ctb],
Statens Serum Institut [cph, fnd]

Maintainer Rasmus Skytte Randløv <rske@ssi.dk>

Repository CRAN

Date/Publication 2026-01-23 16:00:23 UTC

Contents

close_connection *Close connection to the database*

Description

Close connection to the database

Usage

```
close_connection(conn)
```

Arguments

conn	(DBIConnection(1)) Connection object.

Value

dbDisconnect() returns TRUE, invisibly.

Examples

```
conn <- get_connection()

close_connection(conn)
```

create_index *Create the indexes on table*

Description

Create the indexes on table

Usage

```
create_index(conn, db_table, columns)
```

Arguments

conn	(DBIConnection) A connection to a database.
db_table	(id-like object(1)) A table specification (coercible by id()).
columns	(character()) The columns that should be unique.

Value

NULL (called for side effects)

Examples

```
conn <- get_connection()

mt <- dplyr::copy_to(conn, dplyr::distinct(mtcars, .data$mpg, .data$cyl), name = "mtcars")
create_index(conn, mt, c("mpg", "cyl"))

close_connection(conn)
```

`create_logs_if_missing` *Create a table with the SCDB log structure if it does not exists*

Description

Create a table with the SCDB log structure if it does not exists

Usage

```
create_logs_if_missing(conn, log_table)
```

Arguments

`conn`	(`DBIConnection(1)`) Connection object.
`log_table`	(`id-like object`) A table specification where the logs should exist (coercible by `id()`).

Value

Invisibly returns the generated (or existing) log table.

Examples

```
conn <- get_connection()
log_table <- id("test.logs", conn = conn, allow_table_only = TRUE)

create_logs_if_missing(conn, log_table)

close_connection(conn)
```

create_table *Create a historical table from input data*

Description

Create a historical table from input data

Usage

```
create_table(.data, conn = NULL, db_table, ...)
```

Arguments

.data	(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1)) Data object.
conn	(DBIConnection(1)) Connection object.
db_table	(id-like object(1)) A table specification (coercible by id()).
...	Other arguments passed to DBI::dbCreateTable().

Value

Invisibly returns the table as it looks on the destination (or locally if conn is NULL).

Examples

```
conn <- get_connection()

create_table(mtcars, conn = conn, db_table = "mtcars")

close_connection(conn)
```

db_locks *Sets, queries and removes locks for database tables*

Description

This set of function adds a simple locking system to database tables.

- lock_table() adds a record in the schema.locks table with the current time and R-session process id.
- unlock_table() removes records in the schema.locks table with the target table and the R-session process id.

When locking a table, the function will check for existing locks on the table and produce an error a lock is held by a process which no longer exists. In this case, the lock needs to be removed manually by removing the record from the lock table. In addition, the error implies that a table may have partial updates that needs to be manually rolled back.

Usage

```
lock_table(conn, db_table, schema = NULL)

unlock_table(conn, db_table, schema = NULL, pid = Sys.getpid())
```

Arguments

conn	(DBIConnection(1)) Connection object.
db_table	(character(1)) A specification of "schema.table" to modify lock for.
schema	(character(1)) The schema where the "locks" table should be created.
pid	(numeric(1)) The process id to remove the lock for.

Value

- lock_table() returns the TRUE (FALSE) if the lock was (un)successfully added. If a lock exists for a non-active process, an error is thrown.
- unlock_table() returns NULL (called for side effects).

Examples

```
conn <- DBI::dbConnect(RSQLite::SQLite())

lock_table(conn, "test_table") # TRUE

unlock_table(conn, "test_table")

DBI::dbDisconnect(conn)
```

---

db_timestamp *Determine the type of timestamps the database supports*

Description

Determine the type of timestamps the database supports

Usage

```
db_timestamp(timestamp, conn = NULL)
```

Arguments

timestamp	(POSIXct(1) or character(1)) The timestamp to be transformed to the database type.
conn	(DBIConnection(1)) Connection object.

Value

The given timestamp converted to a SQL-backend dependent timestamp.

Examples

```
conn <- get_connection()

db_timestamp(Sys.time(), conn)

close_connection(conn)
```

---

defer_db_cleanup *Delete table at function exit*

Description

This function marks a table for deletion once the current function exits.

Usage

```
defer_db_cleanup(db_table)
```

Arguments

db_table	(tbl_sql) A unmanipulated reference to a sql table.

Value

NULL (called for side effects)

Examples

```
conn <- get_connection()

mt <- dplyr::copy_to(conn, mtcars)
id_mt <- id(mt)

defer_db_cleanup(mt)

DBI::dbExistsTable(conn, id_mt) # TRUE

withr::deferred_run()

DBI::dbExistsTable(conn, id_mt) # FALSE

close_connection(conn)
```

`delta_loading` *Import and export a data-chunk with history from historical data*

Description

`delta_export()` exports data from tables created with `update_snapshot()` in chunks to allow for faster migration of data between sources.

`delta_load()` import deltas created by `delta_export()` to rebuild a historical table.

Usage

```
delta_export(conn, db_table, timestamp_from, timestamp_until = NULL)

delta_load(conn, db_table, delta, logger = NULL)
```

Arguments

`conn`	(`DBIConnection(1)`) Connection object.
`db_table`	(`id-like object(1)`) A table specification (coercible by `id()`).
`timestamp_from`	(`POSIXct(1)`, `Date(1)`, or `character(1)`) The timestamp describing the start of the export (including).
`timestamp_until`	(`POSIXct(1)`, `Date(1)`, or `character(1)`) The timestamp describing the end of the export (not-including). If `NULL` (default), all history after `timestamp_from` is exported.
`delta`	.data (`data.frame(1)`, `tibble(1)`, `data.table(1)`, or `tbl_dbi(1)`) A "delta" exported from `delta_export()` to load. A list of "deltas" can also be supplied.

`logger` (`Logger(1)`)
A configured logging object. If none is given, one is initialized with default arguments.

Details

This pair of functions is designed to facilitate easy migration or incremental backups of a historical table (created by `update_snapshot()`).

To construct the basis of incremental backups, `delta_export()` can be called with only `timestamp_from` at the desired frequency (weekly etc.)

To migrate a historical table in chunks, `delta_export()` can be called with `timestamp_until` to constrain the size of the delta.

In either case, the table can then be re-constructed by "replaying" the deltas with `delta_load()`. The order the deltas are replayed does not matter, but all have to be replayed to achieve the same state as the source table.

Value

`delta_export()` returns a lazy-query containing the data (and history) in the source to be used in conjunction with `delta_load()`.

This table is a temporary table that may need cleaning up.

`delta_load()` returns NULL (called for side effects).

See Also

update_snapshot

Examples

```
conn <- get_connection()

data <- dplyr::copy_to(conn, mtcars)

# Copy the first 3 records
update_snapshot(
  head(data, 3),
  conn = conn,
  db_table = "test.mtcars",
  timestamp = "2020-01-01"
)

# Create a delta with the current state
delta <- delta_export(
  conn,
  db_table = "test.mtcars",
  timestamp_from = "2020-01-01"
)

# Update with the first 5 records
```

```
update_snapshot(
  head(data, 5),
  conn = conn,
  db_table = "test.mtcars",
  timestamp = "2021-01-01"
)

dplyr::tbl(conn, "test.mtcars")

# Create a backup using the delta
delta_load(
  conn = conn,
  db_table = "test.mtcars_backup",
  delta = delta
)

dplyr::tbl(conn, "test.mtcars_backup")

close_connection(conn)
```

`digest_to_checksum` *Computes an checksum from columns*

Description

Computes an checksum from columns

Usage

```
digest_to_checksum(.data, col = "checksum", exclude = NULL, warn = TRUE)
```

Arguments

`.data`	(`data.frame(1)`, `tibble(1)`, `data.table(1)`, or `tbl_dbi(1)`) Data object.
`col`	(`character(1)`) Name of the column to put the checksums in. Will be generated if missing.
`exclude`	(`character()`) Columns to exclude from the checksum generation.
`warn`	(`logical(1)`) Warn if `col` exists in the input data?

Details

In most cases, the md5 algorithm is used to compute the checksums. For Microsoft SQL Server, the SHA-256 algorithm is used.

Value

.data with a checksum column added.

Examples

```
digest_to_checksum(mtcars)
```

`filter_keys` *Filters .data according to all records in the filter*

Description

If `filters` is `NULL`, no filtering is done. Otherwise, the `.data` object is filtered via an `inner_join()` using all columns of the filter: `inner_join(.data, filter, by = colnames(filter))`

`by` and `na_by` can overwrite the `inner_join()` columns used in the filtering.

Usage

```
filter_keys(.data, filters, by = NULL, na_by = NULL, ...)
```

Arguments

`.data`	(`data.frame(1)`, `tibble(1)`, `data.table(1)`, or `tbl_dbi(1)`) Data object.
`filters`	(`data.frame(1)`, `tibble(1)`, `data.table(1)`, or `tbl_dbi(1)`) A object subset data by. If filters is `NULL`, no filtering occurs. Otherwise, an `inner_join()` is performed using all columns of the filter object.
`by`	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

na_by	(character()) Columns where NA should match with NA.
...	Further arguments passed to dplyr::inner_join().

Value

An object of same class as .data

Examples

```
# Filtering with null means no filtering is done
filter <- NULL
identical(filter_keys(mtcars, filter), mtcars) # TRUE

# Filtering by vs = 0
filter <- data.frame(vs = 0)
identical(filter_keys(mtcars, filter), dplyr::filter(mtcars, vs == 0)) # TRUE

# Filtering by the specific combinations of vs = 0 and am = 1
filter <- dplyr::distinct(mtcars, vs, am)
filter_keys(mtcars, filter)
```

---

get_catalog *Get the current schema/catalog of a database-related objects*

Description

Get the current schema/catalog of a database-related objects

Usage

```
get_catalog(obj, ...)

## S3 method for class '`Microsoft SQL Server`'
get_catalog(obj, temporary = FALSE, ...)

get_schema(obj, ...)

## S3 method for class 'PqConnection'
get_schema(obj, temporary = FALSE, ...)

## S3 method for class 'SQLiteConnection'
get_schema(obj, temporary = FALSE, ...)
```

Arguments

obj	(DBIConnection(1), tbl_dbi(1), Id(1)) The object from which to retrieve a schema/catalog.
...	Further arguments passed to methods.
temporary	(logical(1)) Should the reference be to the temporary schema/catalog?

Value

The catalog is extracted from obj depending on the type of input:

- For get_catalog.Microsoft SQL Server, the current database context of the connection or "tempdb" if temporary = TRUE.
- For get_schema.tbl_dbi the catalog is determined via id().
- For get_catalog.*, NULL is returned.

The schema is extracted from obj depending on the type of input:

- For get_schema.DBIConnection(), the current schema of the connection if temporary = FALSE. See "Default schema" for more. If temporary = TRUE, the temporary schema of the connection is returned.
- For get_schema.tbl_dbi() the schema is determined via id().
- For get_schema.Id(), the schema is extracted from the Id specification.

Default schema

In some backends, it is possible to modify settings so that when a schema is not explicitly stated in a query, the backend searches for the table in this schema by default. For Postgres databases, this can be shown with SELECT CURRENT_SCHEMA() (defaults to public) and modified with SET search_path TO { schema }.

For SQLite databases, a temp schema for temporary tables always exists as well as a main schema for permanent tables. Additional databases may be attached to the connection with a named schema, but as the attachment must be made after the connection is established, get_schema will never return any of these, as the default schema will always be main.

Examples

```
conn <- get_connection()

dplyr::copy_to(conn, mtcars, name = "mtcars", temporary = FALSE)

get_schema(conn)
get_schema(get_table(conn, id("mtcars", conn = conn)))

get_catalog(conn)
get_catalog(get_table(conn, id("mtcars", conn = conn)))

close_connection(conn)
```

`get_connection` *Opens connection to the database*

Description

This is a convenience wrapper for DBI::dbConnect() for different database backends.

Connects to the specified dbname of host:port using user and password from given arguments (if applicable). Certain drivers may use credentials stored in a file, such as ~/.pgpass (PostgreSQL).

Usage

```
get_connection(drv, ...)

## S3 method for class 'SQLiteDriver'
get_connection(
  drv,
  dbname = ":memory:",
  ...,
  bigint = c("integer", "bigint64", "numeric", "character")
)

## S3 method for class 'PqDriver'
get_connection(
  drv,
  dbname = NULL,
  host = NULL,
  port = NULL,
  password = NULL,
  user = NULL,
  ...,
  bigint = c("integer", "bigint64", "numeric", "character"),
  check_interrupts = TRUE,
  timezone = Sys.timezone(),
  timezone_out = Sys.timezone()
)

## S3 method for class 'OdbcDriver'
get_connection(
  drv,
  dsn = NULL,
  ...,
  bigint = c("integer", "bigint64", "numeric", "character"),
  timezone = Sys.timezone(),
  timezone_out = Sys.timezone()
)
```

```
## S3 method for class 'duckdb_driver'
get_connection(
  drv,
  dbdir = ":memory:",
  ...,
  bigint = c("numeric", "character"),
  timezone_out = Sys.timezone()
)

## Default S3 method:
get_connection(drv, ...)
```

Arguments

drv	(DBIDriver(1) or DBIConnection(1)) The driver for the connection (defaults to SQLiteDriver).
...	Additional parameters sent to DBI::dbConnect().
dbname	(character(1)) Name of the database located at the host.
bigint	(character(1)) The datatype to convert integers to. Support depends on the database backend.
host	(character(1)) The ip of the host to connect to.
port	(numeric(1) or character(1)) Host port to connect to.
password	(character(1)) Password to login with.
user	(character(1)) Username to login with.
check_interrupts	(logical(1)) Should user interrupts be checked during the query execution?
timezone	(character(1)) Sets the timezone of DBI::dbConnect(). Must be in OlsonNames().
timezone_out	(character(1)) Sets the timezone_out of DBI::dbConnect(). Must be in OlsonNames().
dsn	(character(1)) The data source name to connect to.
dbdir	(character(1)) The directory where the database is located.

Value

An object that inherits from DBIConnection driver specified in drv.

See Also

RSQLite::SQLite

RPostgres::Postgres

odbc::odbc

duckdb::duckdb

Examples

```
conn <- get_connection(drv = RSQLite::SQLite(), dbname = ":memory:")

DBI::dbIsValid(conn) # TRUE

close_connection(conn)

DBI::dbIsValid(conn) # FALSE
```

`get_table` *Retrieves a named table from a given schema on the connection*

Description

Retrieves a named table from a given schema on the connection

Usage

```
get_table(conn, db_table = NULL, slice_ts = NA, include_slice_info = FALSE)
```

Arguments

`conn`	(`DBIConnection(1)`) Connection object.
`db_table`	(`id-like object(1)`) A table specification (coercible by `id()`). If missing, a list of available tables is printed.
`slice_ts`	(`POSIXct(1)`, `Date(1)`, or `character(1)`) If set different from `NA` (default), the returned data looks as on the given date. If set as `NULL`, all data is returned.
`include_slice_info`	(`logical(1)`) Should the history columns "checksum", "from_ts", "until_ts" are also be returned?

Value

A "lazy" data.frame (tbl_lazy) generated using dbplyr.

Note that a temporary table will be preferred over ordinary tables in the default schema (see `get_schema()`) with an identical name.

Examples

```
conn <- get_connection()

dplyr::copy_to(conn, mtcars, name = "mtcars", temporary = FALSE)

get_table(conn)
if (table_exists(conn, "mtcars")) {
  get_table(conn, "mtcars")
}

close_connection(conn)
```

`get_tables` *List the available tables on the connection*

Description

List the available tables on the connection

Usage

```
get_tables(conn, pattern = NULL, show_temporary = TRUE)
```

Arguments

`conn`	(`DBIConnection(1)`) Connection object.
`pattern`	(`character(1)`) Regex pattern with which to subset the returned tables.
`show_temporary`	(`logical(1)`) Should temporary tables be listed?

Value

A data.frame containing table names including schema (and catalog when available) in the database.

Examples

```
conn <- get_connection()

dplyr::copy_to(conn, mtcars, name = "my_test_table_1", temporary = FALSE)
dplyr::copy_to(conn, mtcars, name = "my_test_table_2")

get_tables(conn, pattern = "my_[th]est")
get_tables(conn, pattern = "my_[th]est", show_temporary = FALSE)

close_connection(conn)
```

---

id *Convenience function for DBI::Id*

Description

Convenience function for DBI::Id

Usage

```
id(db_table, ...)

## S3 method for class 'Id'
id(db_table, conn = NULL, ...)

## S3 method for class 'character'
id(db_table, conn = NULL, allow_table_only = TRUE, ...)

## S3 method for class 'data.frame'
id(db_table, ...)
```

Arguments

db_table	(id-like object(1)) A table specification (coercible by id()).
...	Further arguments passed to methods.
conn	(DBIConnection(1)) Connection object.
allow_table_only	(logical(1)) If TRUE, allows for returning an DBI::Id with table = "myschema.table" if schema "myschema" is not found in conn. If FALSE, the function will raise an error if the implied schema cannot be found in conn.

Details

The given db_table is parsed to a DBI::Id depending on the type of input:

- character: db_table is parsed to a DBI::Id object using an assumption of "schema.table" syntax with corresponding schema (if found in conn) and table values. If no schema is implied, the default schema of conn will be used.
- DBI::Id: if schema is not specified in Id, the schema is set to the default schema for conn (if given).
- tbl_sql: the remote name is used to resolve the table identification.
- data.frame: A Id is built from the data.frame (columns catalog, schema, and table). Can be used in conjunction with get_tables(conn, pattern).

Value

A DBI::Id object parsed from db_table (see details).

See Also

DBI::Id which this function wraps.

Examples

```
id("schema.table")
```

---

interlace — *Combine any number of tables, where each has their own time axis of validity*

Description

The function "interlaces" the queries and combines their validity time axes (valid_from and valid_until) onto a single time axis.

Usage

```
interlace(tables, by = NULL, colnames = NULL)
```

Arguments

tables	(list(tbl_dbi(1))) The historical tables to combine.
by	(character()) The variable to merge by.
colnames	(named list()) If the time axes of validity is not called "valid_to" and "valid_until" inside each tbl_dbi, you can specify their names by supplying the arguments as a list: e.g. c(t1.from = "\<colname\>", t2.until = "\<colname\>"). colnames must be named in same order as as given in tables (i.e. t1, t2, t3, ...).

Value

The combination of input queries with a single, interlaced valid_from / valid_until time axis.

```
The combination of input queries with a single, interlaced
        valid_from / valid_until time axis
```

Examples

```
conn <- get_connection()

t1 <- data.frame(key = c("A", "A", "B"),
                 obs_1   = c(1, 2, 2),
                 valid_from  = as.Date(c("2021-01-01", "2021-02-01", "2021-01-01")),
                 valid_until = as.Date(c("2021-02-01", "2021-03-01", NA)))
t1 <- dplyr::copy_to(conn, df = t1, name = "t1")

t2 <- data.frame(key = c("A", "B"),
                 obs_2 = c("a", "b"),
                 valid_from  = as.Date(c("2021-01-01", "2021-01-01")),
                 valid_until = as.Date(c("2021-04-01", NA)))
t2 <- dplyr::copy_to(conn, df = t2, name = "t2")

interlace(list(t1, t2), by = "key")

close_connection(conn)
```

---

is.historical *Checks if table contains historical data*

Description

Checks if table contains historical data

Usage

```
is.historical(.data)
```

Arguments

.data	(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1)) Data object.

Value

TRUE if .data contains the columns: "checksum", "from_ts", and "until_ts". FALSE otherwise.

Examples

```
conn <- get_connection()

dplyr::copy_to(conn, mtcars, name = "mtcars", temporary = FALSE)
create_table(mtcars, conn, db_table = id("mtcars_historical", conn))

is.historical(get_table(conn, "mtcars")) # FALSE
is.historical(get_table(conn, "mtcars_historical")) # TRUE

close_connection(conn)
```

---

joins *SQL Joins*

Description

Overloads the dplyr `*_join` to accept an `na_by` argument. By default, joining using SQL does not match on `NA` / `NULL`. dbplyr `*_joins` has the option "na_matches = na" to match on `NA` / `NULL` but this is very inefficient in some cases. This function does the matching more efficiently: If a column contains `NA` / `NULL`, the names of these columns can be passed via the `na_by` argument and efficiently match as if "na_matches = na". If no `na_by` argument is given is given, the function defaults to using `dplyr::*_join`.

Usage

```
## S3 method for class 'tbl_sql'
inner_join(x, y, by = NULL, ...)

## S3 method for class 'tbl_sql'
left_join(x, y, by = NULL, ...)

## S3 method for class 'tbl_sql'
right_join(x, y, by = NULL, ...)

## S3 method for class 'tbl_sql'
full_join(x, y, by = NULL, ...)

## S3 method for class 'tbl_sql'
semi_join(x, y, by = NULL, ...)

## S3 method for class 'tbl_sql'
anti_join(x, y, by = NULL, ...)
```

Arguments

`x, y`	A pair of lazy data frames backed by database queries.

by — A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

... — Other parameters passed onto methods.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

See Also

dplyr::mutate-joins which this function wraps.

dbplyr::join.tbl_sql which this function wraps.

dplyr::show_query

Examples

```
library(dplyr, warn.conflicts = FALSE)
library(dbplyr, warn.conflicts = FALSE)

band_db <- tbl_memdb(dplyr::band_members)
instrument_db <- tbl_memdb(dplyr::band_instruments)

left_join(band_db, instrument_db) %>%
  show_query()

# Can join with local data frames by setting copy = TRUE
left_join(band_db, dplyr::band_instruments, copy = TRUE)

# Unlike R, joins in SQL don't usually match NAs (NULLs)
db <- memdb_frame(x = c(1, 2, NA))
label <- memdb_frame(x = c(1, NA), label = c("one", "missing"))
left_join(db, label, by = "x")
```

```
# But you can activate R's usual behaviour with the na_matches argument
left_join(db, label, by = "x", na_matches = "na")

# By default, joins are equijoins, but you can use `sql_on` to
# express richer relationships
db1 <- memdb_frame(x = 1:5)
db2 <- memdb_frame(x = 1:3, y = letters[1:3])

left_join(db1, db2) %>% show_query()
left_join(db1, db2, sql_on = "LHS.x < RHS.x") %>% show_query()
```

`Logger` *Logger: Complete logging to console, file and database*

Description

The `Logger` class facilitates logging to a database and/or file and to console.

A `Logger` is associated with a specific table and timestamp which must be supplied at initialization. This information is used to create the log file (if a `log_path` is given) and the log entry in the database (if a `log_table_id` and `log_conn` is given).

Logging to the database must match the fields in the log table.

Value

A new instance of the `Logger` R6 class.

Active bindings

`output_to_console (logical(1))`
: Should the Logger output to console? Read only. This can always be overridden by Logger$log_info(..., output_to_console = FALSE).

`log_path (character(1))`
: The location log files are written (if this is not NULL). Defaults to `getOption("SCDB.log_path")`. Read only.

`log_tbl (tbl_dbi(1))`
: The database table used for logging. Class is connection-specific, but inherits from `tbl_dbi`. Read only.

`start_time (POSIXct(1))`
: The time at which data processing was started. Read only.

`log_filename (character(1))`
: The filename (basename) of the file that the `Logger` instance will output to. Read only.

`log_realpath (character(1))`
: The full path to the logger's log file. Read only.

Methods

Public methods:

- Logger$new()
- Logger$set_timestamp()
- Logger$log_info()
- Logger$log_warn()
- Logger$log_error()
- Logger$log_to_db()
- Logger$finalize_db_entry()
- Logger$clone()

Method `new()`**:** Create a new `Logger` object

Usage:

```
Logger$new(
  db_table = NULL,
  timestamp = NULL,
  output_to_console = TRUE,
  log_table_id = getOption("SCDB.log_table_id"),
  log_conn = NULL,
  log_path = getOption("SCDB.log_path"),
  start_time = Sys.time(),
  warn = TRUE
)
```

Arguments:

`db_table (id-like object(1))`
: A table specification (coercible by `id()`) specifying the table being updated.

`timestamp (POSIXct(1), Date(1), or character(1))`
: A timestamp describing the data being processed (not the current time).

`output_to_console (logical(1))`
: Should the Logger output to console?

`log_table_id (id-like object(1))`
: A table specification (coercible by `id()`) specifying the location of the log table.

`log_conn (DBIConnection(1))`
: A database connection where log table should exist.

`log_path (character(1))`
: The path where logs are stored. If `NULL`, no file logs are created.

`start_time (POSIXct(1))`
: The time at which data processing was started (defaults to `Sys.time()`).

`warn (logical(1))`
: Should a warning be produced if no logging will be done?

Method `set_timestamp()`**:** Update the timestamp being logged

Usage:

```
Logger$set_timestamp(timestamp)
```

Arguments:

`timestamp (POSIXct(1), Date(1), or character(1))`
A timestamp describing the data being processed (not the current time).

Method `log_info()`**:** Write a line to log (console / file).

Usage:

```
Logger$log_info(
  ...,
  tic = Sys.time(),
  output_to_console = self$output_to_console,
  log_type = "INFO",
  timestamp_format = getOption("SCDB.log_timestamp_format", "%F %R:%OS3")
)
```

Arguments:

`... (character())`
Character strings to be concatenated as log message.

`tic (POSIXct(1))`
The timestamp used by the log entry.

`output_to_console (logical(1))`
Should the line be written to console?

`log_type (character(1))`
The severity of the log message.

`timestamp_format (character(1))`
The format of the timestamp used in the log message (parsable by `strftime()`).

Returns: Returns the log message invisibly

Method `log_warn()`**:** Write a warning to log file and generate warning.

Usage:

```
Logger$log_warn(..., log_type = "WARNING")
```

Arguments:

`... (character())`
Character strings to be concatenated as log message.

`log_type (character(1))`
The severity of the log message.

Method `log_error()`**:** Write an error to log file and stop execution.

Usage:

```
Logger$log_error(..., log_type = "ERROR")
```

Arguments:

`... (character())`
Character strings to be concatenated as log message.

`log_type (character(1))`
The severity of the log message.

Method `log_to_db()`**:** Write or update log table.

Usage:

```
Logger$log_to_db(...)
```

Arguments:

`...` (`Name-value pairs`)
: Structured data written to database log table. Name indicates column and value indicates value to be written.

Method `finalize_db_entry()`**:** Auto-fills "end_time" and "duration" for the log entry and clears the "log_file" field if no file is being written.

Usage:

```
Logger$finalize_db_entry(end_time = Sys.time())
```

Arguments:

`end_time` (`POSIXct(1)`, `Date(1)`, or `character(1)`)
: The end time for the log entry.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Logger$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
logger <- Logger$new(
  db_table = "test.table",
  timestamp = "2020-01-01 09:00:00"
)

logger$log_info("This is an info message")
logger$log_to_db(message = "This is a message")

try(logger$log_warn("This is a warning!"))
try(logger$log_error("This is an error!"))
```

`LoggerNull` *LoggerNull: The no-logging Logger*

Description

The `LoggerNull` class overwrites the functions of the `Logger` so no logging is produced. Errors and warnings are still produced.

Value

A new instance of the `LoggerNull` R6 class.

Super class

SCDB::Logger -> LoggerNull

Methods

Public methods:

- LoggerNull$new()
- LoggerNull$log_to_db()
- LoggerNull$finalize_db_entry()
- LoggerNull$clone()

Method new(): Create a new LoggerNull object

Usage:

```
LoggerNull$new(...)
```

Arguments:

... Captures arguments given, but does nothing

Method log_to_db(): Matches the signature of Logger$log_to_db(), but does nothing.

Usage:

```
LoggerNull$log_to_db(...)
```

Arguments:

... Captures arguments given, but does nothing

Method finalize_db_entry(): Matches the signature of Logger$finalize_db_entry(), but does nothing.

Usage:

```
LoggerNull$finalize_db_entry(...)
```

Arguments:

... Captures arguments given, but does nothing

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
LoggerNull$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
logger <- LoggerNull$new()

logger$log_info("This message will not print!")
logger$log_to_db(message = "This message will no be written in database!")
try(logger$log_warn("This is a warning!"))
try(logger$log_error("This is an error!"))
```

nrow *nrow() but also works on remote tables*

Description

nrow() but also works on remote tables

Usage

```
nrow(.data)
```

Arguments

.data	(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1)) Data object.

Value

The number of records in the object.

Examples

```
conn <- get_connection()

m <- dplyr::copy_to(conn, mtcars)
nrow(m) == nrow(mtcars) # TRUE

close_connection(conn)
```

schema_exists *Test if a schema exists in given connection*

Description

Test if a schema exists in given connection

Usage

```
schema_exists(conn, schema)
```

Arguments

conn	(DBIConnection(1)) Connection object.
schema	(character(1)) The schema name to test existence for.

Value

TRUE if the given schema is found on conn.

Examples

```
conn <- get_connection()

schema_exists(conn, "test")

close_connection(conn)
```

`slice_time` *Slices a data object based on time / date*

Description

Slices a data object based on time / date

Usage

```
slice_time(.data, slice_ts, from_ts = "from_ts", until_ts = "until_ts")
```

Arguments

`.data`	(`data.frame(1)`, `tibble(1)`, `data.table(1)`, or `tbl_dbi(1)`) Data object.
`slice_ts`	(`POSIXct(1)`, `Date(1)`, or `character(1)`) The time / date to slice by.
`from_ts`, `until_ts`	(`character(1)`) The name of the columns in .data specifying valid from and valid until time.

Value

An object of same class as .data

Examples

```
conn <- get_connection()

m <- mtcars %>%
  dplyr::mutate(
    "from_ts" = dplyr::if_else(dplyr::row_number() > 10,
                               as.Date("2020-01-01"),
                               as.Date("2021-01-01")),
    "until_ts" = as.Date(NA))
```

```
dplyr::copy_to(conn, m, name = "mtcars", temporary = FALSE)

q <- dplyr::tbl(conn, id("mtcars", conn))

nrow(slice_time(q, "2020-01-01")) # 10
nrow(slice_time(q, "2021-01-01")) # nrow(mtcars)

close_connection(conn)
```

`table_exists` *Test if a table exists in database*

Description

This functions attempts to determine the existence of a given table. If a character input is given, matching is done heuristically assuming a "schema.table" notation. If no schema is implied in this case, the default schema is assumed.

Usage

```
table_exists(conn, db_table)

## S3 method for class 'DBIConnection'
table_exists(conn, db_table)
```

Arguments

`conn`	(`DBIConnection(1)`) Connection object.
`db_table`	(`id-like object(1)`) A table specification (coercible by `id()`).

Value

`TRUE` if db_table can be parsed to a table found in `conn`.

Examples

```
conn <- get_connection()

dplyr::copy_to(conn, mtcars, name = "mtcars", temporary = FALSE)
dplyr::copy_to(conn, iris, name = "iris")

table_exists(conn, "mtcars")    # TRUE
table_exists(conn, "iris")      # FALSE
table_exists(conn, "temp.iris") # TRUE

close_connection(conn)
```

unique_table_name *Create a name for a temporary table*

Description

This function is heavily inspired by the unexported dbplyr function unique_table_name

Usage

```
unique_table_name(scope = "SCDB")
```

Arguments

scope	(character(1)) A naming scope to generate the table name within.

Value

A character string for a table name based on the given scope parameter

Examples

```
print(unique_table_name()) # SCDB_<10 alphanumerical letters>
print(unique_table_name()) # SCDB_<10 alphanumerical letters>

print(unique_table_name("test")) # test_<10 alphanumerical letters>
print(unique_table_name("test")) # test_<10 alphanumerical letters>
```

unite.tbl_dbi *tidyr::unite for tbl_dbi*

Description

Convenience function to paste together multiple columns into one.

Usage

```
unite.tbl_dbi(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)
```

Arguments

data	A data frame.
col	The name of the new column, as a string or symbol. This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with rlang::ensym() (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).
...	<tidy-select> Columns to unite
sep	Separator to use between values.
remove	If TRUE, remove input columns from output data frame.
na.rm	If TRUE, missing values will be removed prior to uniting each value.

Value

A tbl_dbi with the specified columns united into a new column named according to "col".

See Also

separate(), the complement.

Examples

```
library(tidyr, warn.conflicts = FALSE)

df <- expand_grid(x = c("a", NA), y = c("b", NA))

unite(df, "z", x:y, remove = FALSE)

# To remove missing values:
unite(df, "z", x:y, na.rm = TRUE, remove = FALSE)

# Separate is almost the complement of unite
unite(df, "xy", x:y) %>%
  separate(xy, c("x", "y"))
# (but note `x` and `y` contain now "NA" not NA)
```

---

update_snapshot *Update a historical table*

Description

update_snapshot() makes it easy to create and update a historical data table on a remote (SQL) server. The function takes the data (.data) as it looks on a given point in time (timestamp) and then updates (or creates) an remote table identified by db_table. This update only stores the changes between the new data (.data) and the data currently stored on the remote. This way, the data can be reconstructed as it looked at any point in time while taking as little space as possible.

See vignette("basic-principles") for further introduction to the function.

Usage

```
update_snapshot(
  .data,
  conn,
  db_table,
  timestamp,
  filters = NULL,
  message = NULL,
  tic = Sys.time(),
  logger = NULL,
  enforce_chronological_order = TRUE,
  collapse_continuous_records = FALSE
)
```

Arguments

.data	(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1)) Data object.
conn	(DBIConnection(1)) Connection object.
db_table	(id-like object(1)) A table specification (coercible by id()).
timestamp	(POSIXct(1), Date(1), or character(1)) The timestamp describing the data being processed (not the current time).
filters	(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1)) A object subset data by. If filters is NULL, no filtering occurs. Otherwise, an inner_join() is performed using all columns of the filter object.
message	(character(1)) A message to add to the log-file (useful for supplying metadata to the log).
tic	(POSIXct(1)) A timestamp when computation began. If not supplied, it will be created at call-time (used to more accurately convey the runtime of the update process).
logger	(Logger(1)) A configured logging object. If none is given, one is initialized with default arguments.

`enforce_chronological_order`
: (`logical(1)`)
Are updates allowed if they are chronologically earlier than latest update?

`collapse_continuous_records`
: (`logical(1)`)
Check for records where from/until time stamps are equal and delete? Forced `TRUE` when `enforce_chronological_order` is `FALSE`.

Details

The most common use case is having consecutive snapshots of a dataset and wanting to store the changes between them. If you have a special case where you want to insert data that is not consecutive, you can set the `enforce_chronological_order` to `FALSE`. This will allow you to insert data that is earlier than the latest time stamp.

If you have more updates in a single day and use `Date()` rather than `POSIXct()`, as your time stamp, you may end up with records where `from_ts` and `until_ts` are equal. These records not normally accessible with `get_table()` and you may want to prevent these records using `collapse_continuous_records = TRUE`.

Value

No return value, called for side effects.

See Also

filter_keys

Examples

```
conn <- get_connection()

data <- dplyr::copy_to(conn, mtcars)

# Copy the first 3 records
update_snapshot(
  head(data, 3),
  conn = conn,
  db_table = "test.mtcars",
  timestamp = Sys.time()
)

# Update with the first 5 records
update_snapshot(
  head(data, 5),
  conn = conn,
  db_table = "test.mtcars",
  timestamp = Sys.time()
)

dplyr::tbl(conn, "test.mtcars")
```

```
close_connection(conn)
```

Index