Package ‘SCDB’
March 21, 2024

Type Package

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

Version 0.4.0

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 datawarehouses and provides a number
of Quality of life improvements for working on SQL databases with R.

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.1

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

Suggests callr, conflicted, duckdb, jsonlite, knitr, lintr, odbc,
markdown, roxygen2, pkgdown, RPostgres, RSQLite, spelling,
testthat (>= 3.0.0), tibble, tidyverse, withr

Language en-US


Config/testthat/edition 3

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

VignetteBuilder knitr

NeedsCompilation no

Author Rasmus Skytte Randløv [aut, cre, rev]
(https://orcid.org/0000-0002-5860-3838),
Marcus Munch Grünewald [aut] (https://orcid.org/0009-0006-8090-406X),
Lasse Engbo Christiansen [rev]
(https://orcid.org/0000-0001-5019-1931),
Statens Serum Institut [cph, fnd]
**R topics documented:**

- close_connection ........................................... 2
- create_logs_if_missing .................................. 3
- create_table .............................................. 4
- db_locks .................................................. 4
- db_timestamp ............................................. 6
- defer_db_cleanup ......................................... 6
- digest_to_checksum ....................................... 7
- filter_keys ................................................. 8
- get_catalog ............................................... 9
- get_connection .......................................... 11
- get_table ................................................. 13
- get_tables ............................................... 14
- id .......................................................... 15
- interlace ................................................... 16
- is.historical ............................................. 17
- joins ....................................................... 18
- Logger ...................................................... 20
- LoggerNull ............................................... 24
- nrow ........................................................ 25
- schema_exists ............................................ 26
- slice_time ................................................ 26
- table_exists ............................................. 27
- unique_table_name ...................................... 28
- unite.tbl_db ............................................. 29
- update_snapshot ......................................... 30

**Index**

| Close connection | Close connection to the database |

**Description**

Close connection to the database

**Usage**

```r
close_connection(conn)
```
create_logs_if_missing

Arguments

conn (DBIConnection(1)) Connection object.

Value
dbDisconnect() returns TRUE, invisibly.

Examples

```r
conn <- get_connection()
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

```r
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**

```r
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 if 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

```R
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**

```r
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

```r
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)
```

digest_to_checksum  Computes an checksum from columns

Description

Computes an checksum from columns

Usage

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

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(1))
  Columns to exclude from the checksum generation.

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

```r
digest_to_checksum(mtcars)
```

<table>
<thead>
<tr>
<th>filter_keys</th>
<th>Filters .data according to all records in the filter</th>
</tr>
</thead>
</table>

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

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

Arguments

<table>
<thead>
<tr>
<th>.data</th>
<th>(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>filters</td>
<td>(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1))</td>
</tr>
<tr>
<td>by</td>
<td>A join specification created with <code>join_by()</code>, or a character vector of variables to join by.</td>
</tr>
</tbody>
</table>

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()`.
get_catalog

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

```r
conn <- get_connection()

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

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

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

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 'PgDriver'
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`. 
get_table

See Also

RSQLite::SQLite
RPostgres::Postgres
odbc::odbc
duckdb::duckdb

Examples

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

DBI::dbIsInvalid(conn) # TRUE

close_connection(conn)

DBI::dbIsInvalid(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

```r
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

```r
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.

interlace_sql() is deprecated in favor of interlace()

Usage

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

```r
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)) |>
  dplyr::copy_to(conn, df = _, 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))) |>
  dplyr::copy_to(conn, df = _, 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

```r
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. dplyr *_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

```r
## 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.

- **...**
  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.

### Examples

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

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$new()
• Logger$finalize()
• 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 finalize():** Remove generated log_name from database if not writing to a file.

**Usage:**
Logger$finalize()

**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:
depth 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!"))

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")
table_exists

Description

This function 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

```r
table_exists(conn, db_table)

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

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

```r
print(unique_table_name()) # SCDB_001
print(unique_table_name()) # SCDB_002

print(unique_table_name("test")) # test_001
print(unique_table_name("test")) # test_002
```

---

**unite.tbl_dbi**  
**tidyr::unite for tbl_dbi**

---

**Description**

Convenience function to paste together multiple columns into one.

**Usage**

```r
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

```r
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)
```

Description

`update_snapshots` 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

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

Arguments

- `.data` *(data.frame(1), tibble(1), data.table(1), or tbl_dbi(1))*
  Data object.
- `conn` *(DBIConnection(1))*
  Connection object.
The `update_snapshot` function updates a table with a specified subset of data and timestamp.

### Parameters

- **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?

### Value

No return value, called for side effects.

### See Also

- filter_keys

### Examples

```r
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()
)
```
update_snapshot

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

close_connection(conn)
Index

?join.by, 8, 19
anti_join.tbl_sql (joins), 18

close_connection, 2
collect(), 19
create_logs_if_missing, 3
create_table, 4
cross_join(), 8, 19
db_locks, 4
db_timestamp, 6
DBI::dbCreateTable(), 4
DBI::Id, 16
dplyr::join.tbl_sql, 19
defer_db_cleanup, 6
digest_to_checksum, 7
dplyr::mutate-joins, 19
duckdb::duckdb, 13

dfilter_keys, 8
dfull_join.tbl_sql (joins), 18
dget_catalog, 9
dget_connection, 11
dget_schema (get_catalog), 9
dget_schema(), 14
dget_table, 13
dget_tables, 14

id, 15
inner_join.tbl_sql (joins), 18
interlace, 16
interlace_sql (interlace), 16
is.historical, 17

join.by(), 8, 19
joins, 18

left_join.tbl_sql (joins), 18
lock_table (db_locks), 4

Logger, 20
LoggerNull, 24

nrow, 25
odbc::odbc, 13
OlsonNames(), 12

quasiquotation, 29

R6, 20, 24
right_join.tbl_sql (joins), 18
rlang::ensym(), 29
RPostgres::Postgres, 13
RSQLite::SQLite, 13

SCDB::Logger, 24

schema_exists, 26
semi_join.tbl_sql (joins), 18

separate(), 29
show_query(), 19

slice_time, 26

strftime(), 22

Sys.time(), 22


table_exists, 27

unique_table_name, 28

unite.tbl_dbi, 29
unlock_table (db_locks), 4
update_snapshot, 30