Package 'SCDB'

January 13, 2024

Type Package

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

Version 0.3

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 7.2.3

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

Suggests conflicted, jsonlite, knitr, odbc, rmarkdown, RPostgres, roxygen2, spelling, testthat (>= 3.0.0), tibble

Language en-US

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

Config/testthat/edition 3

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

VignetteBuilder knitr

NeedsCompilation no

Author Rasmus Skytte Randløv [aut, cre] (<https://orcid.org/0000-0002-5860-3838>), Marcus Munch Grünewald [aut] (<https://orcid.org/0009-0006-8090-406X>), Statens Serum Institut [cph, fnd]

Maintainer Rasmus Skytte Randløv <rske@ssi.dk>

Repository CRAN

Date/Publication 2024-01-13 01:10:02 UTC

R topics documented:

`close_connection` *Close connection to the DB*

Description

Close connection to the DB

Usage

```
close_connection(conn)
```

Arguments

`conn`	An object that inherits from DBIConnection (as generated by get_connection())'

Value

`dbDisconnect()` returns `TRUE`, invisibly.

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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(log_table, conn)
```

Arguments

log_table	A specification of where the logs should exist ("schema.table")
conn	An object that inherits from DBIConnection (as generated by get_connection())'

Value

A tbl_dbi with the generated (or existing) log table

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())
log_table_id <- id("test.logs", conn = conn, allow_table_only = TRUE)

create_logs_if_missing(log_table_id, conn)

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_id, temporary = TRUE, ...)
```

Arguments

.data	A data frame, a tibble, a data.table or a tbl.
conn	An object that inherits from DBIConnection (as generated by get_connection())'
db_table_id	A DBI::Id() object or a character string readable by id.
temporary	Should the table be created as a temporary table?
...	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(drv = RSQLite::SQLite())

create_table(mtcars, conn = conn, db_table_id = "mtcars_tmp")

close_connection(conn)
```

---

db_timestamp *Determine the type of timestamps the DB supports*

Description

Determine the type of timestamps the DB supports

Usage

```
db_timestamp(timestamp, conn = NULL)

## Default S3 method:
db_timestamp(timestamp, conn)

## S3 method for class 'SQLiteConnection'
db_timestamp(timestamp, conn)
```

Arguments

timestamp	The timestamp to be transformed to the DB type. Can be character.
conn	A DBIConnection to the DB where the timestamp should be stored

Value

The given timestamp converted to a SQL-backend dependent timestamp

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

db_timestamp(Sys.time(), conn)

close_connection(conn)
```

---

digest_to_checksum *Computes an MD5 checksum from columns*

Description

Computes an MD5 checksum from columns

Usage

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

Arguments

.data	A data frame, a tibble, a data.table or a tbl.
col	Name of the column to put the checksums in
exclude	Columns to exclude from the checksum generation
warn	Flag to warn if target column already exists in data

Value

.data with an checksum column added

Examples

```
digest_to_checksum(mtcars)
```

---

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

Description

If filter = NULL, not filtering is done If filter is different from NULL, the .data is filtered by a 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	A data frame, a tibble, a data.table or a tbl.
filters	A filter to subset data by. if filters == NULL, no filtering occurs. Else, an inner_join is performed using all columns of the filter
by	passed to inner_join if different from NULL
na_by	passed to inner_join if different from NULL

Value

An object similar to .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_connection *Opens connection to the database*

Description

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

Usage

```
get_connection(
  drv = RPostgres::Postgres(),
  host = NULL,
  port = NULL,
  dbname = NULL,
  user = NULL,
  password = NULL,
```

```
  timezone = NULL,
  timezone_out = NULL,
  ...,
  bigint = "integer",
  check_interrupts = TRUE
)
```

Arguments

drv	An object that inherits from DBIDriver or an existing DBIConnection (default: RPostgres::Postgres())
host	Character string giving the ip of the host to connect to
port	Host port to connect to (numeric)
dbname	Name of the database located at the host
user	Username to login with
password	Password to login with
timezone	Sets the timezone of DBI::dbConnect()
timezone_out	Sets the timezone_out of DBI::dbConnect()
...	Additional parameters sent to DBI::dbConnect()
bigint	The R type that 64-bit integer types should be mapped to, default is bit64::integer64, which allows the full range of 64 bit integers.
check_interrupts	Should user interrupts be checked during the query execution (before first row of data is available)? Setting to TRUE allows interruption of queries running too long.

Value

An object that inherits from DBIConnection driver specified in drv

See Also

RPostgres::Postgres

Examples

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

DBI::dbIsValid(conn) # TRUE

close_connection(conn)

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

`get_schema` *Get the current schema of a database-related objects*

Description

Get the current schema of a database-related objects

Usage

```
get_schema(.x)
```

Arguments

`.x`	The object from which to retrieve a schema

Value

For `DBIConnection` objects, the current schema of the connection. See "default schema" for more.

For `tbl_dbi` objects, the schema as retrieved from the lazy_query. If the lazy_query does not specify a schema, `NA` is returned. Note that lazy queries are sensitive to server-side changes and may therefore return entirely different tables if changes are made server-side.

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 one of `main` or `temp` (if `main` contains no tables). A return value of `NA` means that no tables exist within the connection and no default schema therefore exists.

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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

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

close_connection(conn)
```

get_table *Gets a named table from a given schema*

Description

Gets a named table from a given schema

Usage

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

Arguments

conn	An object that inherits from DBIConnection (as generated by get_connection())'
db_table_id	A DBI::Id() object or a character string readable by id. If missing, a list of available tables is printed.
slice_ts	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	Default FALSE. If set TRUE, the history columns "checksum", "from_ts", "until_ts" are returned also

Value

A "lazy" dataframe (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(drv = RSQLite::SQLite())

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

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

close_connection(conn)
```

get_tables *Gets the available tables*

Description

Gets the available tables

Usage

```
get_tables(conn, pattern = NULL, show_temp = "never")
```

Arguments

conn	An object that inherits from DBIConnection (as generated by get_connection())'
pattern	A regex pattern with which to subset the returned tables
show_temp	A string specifying how to handle temporary tables.

- `"never"`, the default, will never return any temporary tables.
- `"always"`, lists temporary tables as well as ordinary tables. Note that this may return duplicate tables.
- `"fallback"` lists temporary tables, but only if no ordinary table with the same name exists

Value

A data.frame containing table names in the DB

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

dplyr::copy_to(conn, datasets::mtcars, name = DBI::Id(table = "my_test_table"))

get_tables(conn, pattern = "my_[th]est")
get_tables(conn, pattern = "my_[th]est", show_temp = "always")

close_connection(conn)
```

`id` *Convenience function for DBI::Id*

Description

Convenience function for DBI::Id

Usage

```
id(db_table_id, conn = NULL, allow_table_only = TRUE)
```

Arguments

`db_table_id`	A `DBI::Id()` object or a character string readable by id.
`conn`	An object that inherits from DBIConnection (as generated by get_connection())'
`allow_table_only`	logical. If `TRUE`, allows for returning an `DBI::Id` with `table = myschema.table` if schema `myschema` is not found in `conn`.

Details

The given db_table_id is parsed using an assumption of "schema.table" syntax into a DBI::Id object with corresponding schema (if the conn supports it) and table values.

Value

A DBI::Id object parsed from db_table_id

See Also

DBI::Id which this function wraps.

Examples

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

interlace_sql *Combine any number of SQL queries, where each has their own time axis of validity (valid_from and valid_until)*

Description

The function "interlaces" the queries and combines their validity time axes onto a single time axis

Usage

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

Arguments

tables	A list(!) of tables you want to combine is supplied here as lazy_queries.
by	The (group) variable to merge by
colnames	If the time axes of validity is not called "valid_to" and "valid_until" inside each lazy_query, 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
```

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())


if (schema_exists(conn, "test")) {
  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, t1, id("test.SCDB_tmp1", conn), overwrite = TRUE, temporary = FALSE)

  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, t2, id("test.SCDB_tmp2", conn), overwrite = TRUE, temporary = FALSE)

  interlace_sql(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	A data frame, a tibble, a data.table or a tbl.

Value

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

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

dplyr::copy_to(conn, mtcars, name = id("mtcars", conn))
create_table(mtcars, conn, db_table_id = 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 has the option "na_matches = na" to match on NA / NULL but this is very inefficient This function does the matching more efficiently. If a column contains NA / NULL, give the argument to na_by to match during the join If no na_by is given, the function defaults to using `dplyr::*_join`

Usage

```
inner_join.tbl_sql(x, y, by = NULL, ...)

left_join.tbl_sql(x, y, by = NULL, ...)

right_join.tbl_sql(x, y, by = NULL, ...)

full_join.tbl_sql(x, y, by = NULL, ...)

semi_join.tbl_sql(x, y, by = NULL, ...)

anti_join.tbl_sql(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.

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

# Can join with local data frames by setting copy = TRUE
band_db %>%
  left_join(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"))
db %>% left_join(label, by = "x")
# But you can activate R's usual behaviour with the na_matches argument
db %>% left_join(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])
db1 %>% left_join(db2) %>% show_query()
db1 %>% left_join(db2, sql_on = "LHS.x < RHS.x") %>% show_query()
```

`Logger` *Logger*

Description

Create an object for logging database operations

Value

A new instance of the `Logger` R6 class.

Public fields

`log_path (character(1))`
: A directory where log file is written (if this is not NULL). Defaults to `getOption("SCDB.log_path")`.

`log_tbl` The DB table used for logging. Class is connection-specific, but inherits from `tbl_dbi`.

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

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

Active bindings

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

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

Methods

Public methods:

- `Logger$new()`
- `Logger$finalize()`
- `Logger$log_info()`
- `Logger$log_warn()`
- `Logger$log_error()`
- `Logger$log_to_db()`
- `Logger$clone()`

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

Usage:

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

Arguments:

`db_tablestring` A string specifying the table being updated

`log_table_id` A `DBI::Id()` object or a character string readable by id, specifying the location of the log table.

`log_conn` A database connection inheriting from `DBIConnection`

`log_path` The path where logs are stored. If NULL, no file logs are created.

`output_to_console` Should the Logger output to console (TRUE/FALSE)?

`warn` Show a warning if neither log_table_id or log_path could be determined

`ts` A timestamp describing the data being processed (not the current time)

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

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 file

Usage:

```
Logger$log_info(
  ...,
  tic = Sys.time(),
  output_to_console = self$output_to_console,
  log_type = "INFO"
)
```

Arguments:

`...` One or more character strings to be concatenated

`tic` The timestamp used by the log entry (default Sys.time())

`output_to_console` Should the line be written to console?

`log_type` A character string which describes the severity of the log message

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

Usage:

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

Arguments:

`...` One or more character strings to be concatenated

`log_type` A character string which describes 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:

`...` One or more character strings to be concatenated

`log_type` A character string which describes the severity of the log message

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

Usage:

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

Arguments:

`...` Name-value pairs with which to update the log table

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_tablestring = "test.table",
                     ts = "2020-01-01 09:00:00")
```

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

Description

nrow() but also works on remote tables

Usage

```
nrow(.data)
```

Arguments

.data	lazy_query to parse

Value

The number of records in the object

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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	An object that inherits from DBIConnection (as generated by get_connection())'
schema	A character string giving the schema name

Value

TRUE if the given schema is found on conn

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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`	A data frame, a tibble, a data.table or a tbl.
`slice_ts`	The time / date to slice by
`from_ts`	The name of the column in .data specifying valid from time (note: must be unquoted)
`until_ts`	The name of the column in .data specifying valid until time (note: must be unquoted)

Value

An object similar to .data

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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")

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

Test if a table exists in database

Usage

```
table_exists(conn, db_table_id)

## S3 method for class 'SQLiteConnection'
table_exists(conn, db_table_id)

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

Arguments

conn	An object that inherits from DBIConnection (as generated by get_connection())'
db_table_id	A `DBI::Id()` object or a character string readable by id.

Value

TRUE if db_table_id can be parsed to a table found in conn

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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

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

close_connection(conn)
```

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

Description

tidyr::unite for tbl_dbi

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"

Examples

```
library(tidyr, warn.conflicts = FALSE)
df <- expand_grid(x = c("a", NA), y = c("b", NA))
df
df %>% unite("z", x:y, remove = FALSE)
# To remove missing values:

# Separate is almost the complement of unite
df %>%
  unite("xy", x:y) %>%
  separate(xy, c("x", "y"))
```

`update_snapshot` *Update a historical table*

Description

Update a historical table

Usage

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

Arguments

`.data`	A tbl object. Typically a lazy query to the DB (from get_table())
`conn`	An object that inherits from DBIConnection (as generated by get_connection())'
`db_table`	An object that inherits from `tbl_dbi`, a `DBI::Id()` object or a character string readable by id.
`timestamp`	A timestamp (POSIXct) with which to update from_ts/until_ts columns
`filters`	A filter to subset data by. if filters == NULL, no filtering occurs. Else, an inner_join is performed using all columns of the filter
`message`	A message to add to the log-file (useful for supplying metadata to the log)
`tic`	A timestamp when computation began. If not supplied, it will be created at call-time. (Used to more accurately convey how long runtime of the update process has been)
`logger`	A Logger instance. If none is given, one is initialized with default arguments.
`enforce_chronological_order`	A logical that controls whether or not to check if timestamp of update is prior to timestamps in the DB

Value

No return value, called for side effects

See Also

filter_keys

Examples

```
conn <- get_connection(drv = RSQLite::SQLite())

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

update_snapshot(data,
                conn = conn,
                db_table = "test.mtcars",
                timestamp = Sys.time())

close_connection(conn)
```

`%notin%` *not-in operator*

Description

not-in operator

Usage

```
x %notin% table
```

Arguments

x	vector or NULL: the values to be matched. Long vectors are supported.
table	vector or NULL: the values to be matched against. Long vectors are not supported.

Value

A logical vector, indicating if a match was NOT located for each element of x

Examples

```
2 %notin% c(1,3) # TRUE
```

Index