

Package: csdb (via r-universe)

October 16, 2026

Title An Abstracted System for Easily Working with Databases with Large Datasets

Version 2026.10.18

Description Provides object-oriented database management tools for working with large datasets across multiple database systems. Features include robust connection management for 'PostgreSQL' databases, advanced table operations with bulk data loading and upsert functionality, comprehensive data validation through customizable field type and content validators, efficient index management, and cross-database compatibility. Designed for high-performance data operations in surveillance systems and large-scale data processing workflows.

Depends R (>= 4.1.0)

License MIT + file LICENSE

URL <https://niphr.github.io/csdb/>, <https://github.com/niphr/csdb>

BugReports <https://github.com/niphr/csdb/issues>

Encoding UTF-8

LazyData true

Imports csutil, data.table, DBI, dbplyr, dplyr, fs, ggplot2, glue, methods, odbc, R6, RSQLite, S7, stringr, utils, uuid (>= 1.1-0), yyjsonr

Suggests testthat, knitr, rmarkdown, digest, crayon, withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/Needs/website niphr/cstemplate

Config/roxygen2/version 8.0.0

Config/pak/sysreqs cmake make libicu-dev libuv1-dev unixodbc-dev

Repository <https://niphr.r-universe.dev>

Date/Publication 2026-10-16 19:45:21 UTC

RemoteUrl <https://github.com/niphr/csdb>

RemoteRef HEAD
RemoteSha 508776b98eb872e19562cf7ebd1bee9ecbd7b290

Contents

csdb_get_auth_hook	2
csdb_get_context	3
csdb_get_password_hook	4
csdb_set_auth_hook	4
csdb_set_context	5
csdb_set_password_hook	6
DBConnection_v9	7
DBTable_v9	11
get_table_names_and_info	21
insert_data_many	22
nor_covid19_cases_by_time_location	24
validator_field_contents_blank	25
validator_field_contents_csfmt_rts_data_v1	26
validator_field_contents_csfmt_rts_data_v2	27
validator_field_types_blank	28
validator_field_types_csfmt_rts_data_v1	29
validator_field_types_csfmt_rts_data_v2	30
Index	32

csdb_get_auth_hook	<i>Get the current authentication hook</i>
--------------------	--

Description

Returns the currently registered authentication hook function.

Usage

csdb_get_auth_hook()

Value

The current authentication hook function, or NULL when no hook is set.

See Also

[DBConnection_v9](#), whose connect() method calls this function to look up the hook. The introduction vignette, vignette("csdb", package = "csdb"), does not mention this function.
Other auth hook functions: [csdb_set_auth_hook\(\)](#)

Examples

```
# Returns NULL when no hook has been set
csdb_get_auth_hook()

# Register a hook and then read it back. Registering does not call it.
previous <- csdb_set_auth_hook(function() {
  system2("/bin/kinit", stdout = NULL)
})
hook <- csdb_get_auth_hook()
is.function(hook)

csdb_set_auth_hook(previous)
```

csdb_get_context	<i>Get the caller context</i>
------------------	-------------------------------

Description

Returns the string that [csdb_set_context\(\)](#) set.

Usage

```
csdb_get_context()
```

Value

One string, "" when no context is set.

See Also

Other caller context functions: [csdb_set_context\(\)](#)

Examples

```
csdb_get_context()
```

csdb_get_password_hook

Get the current password hook

Description

Returns the function that [csdb_set_password_hook](#) registered.

Usage

```
csdb_get_password_hook()
```

Value

The current password hook function, or NULL when no hook is set.

See Also

Other password hook functions: [csdb_set_password_hook\(\)](#)

Examples

```
# Returns NULL when no hook is set.
csdb_get_password_hook()
```

csdb_set_auth_hook

Set authentication hook for database connections

Description

Registers a function that csdb calls when a database connection fails. Use it to refresh a Kerberos ticket, or other authentication credentials, before the next connection attempt.

Usage

```
csdb_set_auth_hook(hook)
```

Arguments

hook	A function with no arguments that performs authentication, or NULL to clear the hook.
------	---

Value

Invisibly returns the previous hook (if any).

See Also

[DBConnection_v9](#), whose `connect()` method calls the registered hook once, after its first failed attempt. The introduction vignette, `vignette("csdb", package = "csdb")`, does not mention this function.

Other auth hook functions: [csdb_get_auth_hook\(\)](#)

Examples

```
# The hook is held in the csdb.auth_hook option. Setting one returns
# the previous hook, so it can be put back afterwards.
previous <- csdb_set_auth_hook(function() invisible(NULL))
is.function(csdb_get_auth_hook())
csdb_set_auth_hook(previous)
csdb_get_auth_hook()

# A real hook refreshes a credential, for example a Kerberos ticket.
# Registering the hook does not call it, so this block runs on a
# machine that has no such script.
previous <- csdb_set_auth_hook(function() {
  system2("/bin/authenticate.sh", stdout = NULL)
})
is.function(csdb_get_auth_hook())

# Put back whatever was registered before.
csdb_set_auth_hook(previous)
```

csdb_set_context	<i>Set the caller context</i>
------------------	-------------------------------

Description

Sets a short string that names the caller, for example "<task>|<plan>". PostgreSQL shows it as the `application_name` of each `csdb` connection in `pg_stat_activity`.

Usage

```
csdb_set_context(x)
```

Arguments

x	One string. "" clears the context.
---	------------------------------------

Details

With a non-empty context, csdb runs SET application_name with the context right after each PostgreSQL connection that DBConnection_v9 opens. After a failed SET, csdb runs SELECT 1. A connection that answers stays open and reports nothing. A connection that does not answer counts as a failed connect attempt. A PostgreSQL load uses that connection, so it shows the same name.

The name keeps only the letters A-Z and a-z, the digits 0-9 and the characters _ | . : -. csdb drops every other character, and then keeps the first 63 characters, the limit of PostgreSQL. A context that keeps no character sends no SET.

The context is an option, so a forked worker inherits it. A new R process, for example one that callr starts, does not.

Do not put a secret in the context. pg_stat_activity shows it.

Value

Invisibly returns the previous context.

See Also

Other caller context functions: [csdb_get_context\(\)](#)

Examples

```
previous <- csdb_set_context("my_task|plan7")
csdb_get_context()
csdb_set_context(previous)
```

csdb_set_password_hook

Set the password hook for PostgreSQL connections

Description

Registers a function that returns the password for each new PostgreSQL connection. Use it for a password that expires, such as an Entra access token.

Usage

```
csdb_set_password_hook(hook)
```

Arguments

hook	A function with no arguments that returns the password, or NULL to clear the hook.
------	--

Details

csdb calls the hook with no arguments, once for each attempt of `DBConnection_v9$connect()` with the driver "PostgreSQL Unicode". It uses the current return value as the password. A PostgreSQL load uses the connection, and calls the hook only when it opens a new one.

The hook **MUST** return a single non-empty string. Any other value stops the connection with an error that names the hook. The error does not show the value.

The hook applies to PostgreSQL only. A SQL Server, SQLite or other connection, and a bcp load, use the password in their settings. csdb therefore never sends the hook value to another kind of server.

The message of a failed connection shows *** in place of the hook value.

Value

Invisibly returns the previous hook, or NULL when no hook was set.

See Also

[csdb_set_auth_hook](#) registers a function that `DBConnection_v9$connect()` calls after its first failed attempt.

Other password hook functions: [csdb_get_password_hook\(\)](#)

Examples

```
# The hook is held in the csdb.password_hook option. Registering the hook
# does not call it.
previous <- csdb_set_password_hook(function() "example-token")
is.function(csdb_get_password_hook())

# Put back whatever was registered before.
csdb_set_password_hook(previous)
csdb_get_password_hook()
```

DBConnection_v9

R6 Class representing a database connection

Description

A database connection manager that handles connections to various database systems including Microsoft SQL Server and PostgreSQL. This class provides connection management, authentication, and automatic reconnection.

Details

The DBConnection_v9 class holds the database connection logic and provides a consistent interface to different database systems. It supports both trusted connections and user/password authentication. It handles connection failures, and it reconnects automatically.

Key features:

- Support for multiple database systems (SQL Server, PostgreSQL).
- Automatic connection management with retry logic.
- Secure credential handling.
- Connection status monitoring.
- Graceful error handling and recovery.
- A connection is never shared with another process.

Fork safety

A connection belongs to the process that opened it. After a fork, the child holds a copy of this object and a copy of the parent's connection. Both processes then use one socket. PostgreSQL can return wrong results and report no error. Measured against the NorSySS server on 2026-08-14. A child asked for select 4 and read 3. The parent asked for select 999 and read 2. DBI::dbIsValid() reports TRUE on such a handle, so nothing else detects it.

This class records the process that opens each connection. It drops any connection whose recorded process is not the current one. is_connected() then returns FALSE, connection returns NULL, and autoconnection opens a new connection for the current process. disconnect() closes nothing, because the handle belongs to the other process.

The object never closes an inherited handle, and it keeps a reference to it. Both parts are needed. A close, by DBI::dbDisconnect() or by the garbage collector, would close the other process's socket.

PostgreSQL loads

DBTable_v9\$insert_data(), \$upsert_data() and [insert_data_many](#) send rows to PostgreSQL as JSON text over this connection. csdb starts no client, so a load makes no new login. One statement sends at most 500,000 rows, and every statement of one load runs in one transaction.

csdb finds the failure class in the English error text of the server. Each new connection warns once when the lc_messages of the server is not C, POSIX or English. A failed load is csdb_load_ambiguous unless its ROLLBACK succeeded and the connection then answered SELECT 1. Only then can a known English text give csdb_load_duplicate_key or csdb_load_transient. Any other text gives csdb_load_other. The COMMIT and a broken connection also give csdb_load_ambiguous.

Three more rules apply:

- A statement that runs past load_timeout stops on the server, which stores no row. The error has class csdb_load_other.
- A statement that stores fewer rows than csdb sent stops the load before the commit. The error has class csdb_load_other, and the server stores no row.
- A connection that cannot open before the load gives csdb_load_transient, with the error of connect() in its message.

A row with more than 900 MB of JSON text stops the load before csdb sends it. PostgreSQL refuses a value of 1 GB or more. The error has class csdb_load_other.

PostgreSQL double precision

The field type DOUBLE creates a double precision column. Until 2026.10.16 it created real, which keeps about 7 significant digits.

DBTable_v9\$create_table() changes each real column that a table declares DOUBLE to double precision with one ALTER TABLE, and the table keeps its rows. PostgreSQL rewrites the table and its indexes, and blocks every other access to the table until the rewrite ends. The ALTER waits at most 60 s for its lock, and then stops with an error. The table then stays as it was.

Public fields

config Configuration details of the database.

Active bindings

connection Database connection. NULL when another process opened it.

autoconnection Database connection that automatically connects if possible. After a fork it opens a connection for the current process.

Methods

Public methods:

- [DBConnection_v9\\$new\(\)](#)
- [DBConnection_v9\\$is_connected\(\)](#)
- [DBConnection_v9\\$print\(\)](#)
- [DBConnection_v9\\$connect\(\)](#)
- [DBConnection_v9\\$disconnect\(\)](#)
- [DBConnection_v9\\$clone\(\)](#)

DBConnection_v9\$new(): Create a new DBConnection_v9 object.

Usage:

```
DBConnection_v9$new(
  driver = NULL,
  server = NULL,
  port = NULL,
  db = NULL,
  schema = NULL,
  user = NULL,
  password = NULL,
  trusted_connection = NULL,
  sslmode = NULL,
  role_create_table = NULL
)
```

Arguments:

driver Driver.
 server Server.
 port Port.
 db DB.
 schema Schema (e.g. "dbo").
 user User.
 password Password.
 trusted_connection NULL or "yes".
 sslmode NULL or "require".
 role_create_table NULL or the role to take when creating tables.

Returns: A new 'DBConnection_v9' object.

DBConnection_v9\$is_connected(): Is the DB schema connected?

A connection that another process opened does not count. The method drops that connection first, and then reports FALSE.

Usage:

DBConnection_v9\$is_connected()

Returns: TRUE/FALSE.

DBConnection_v9\$print(): Class-specific print function.

Usage:

DBConnection_v9\$print(...)

Arguments:

... Not used.

DBConnection_v9\$connect(): Connect to the database.

The method drops a connection that another process opened, and then opens a connection for the current process.

The method tries again after any error, until it has made attempts attempts. It waits 2 s before attempt 2 and 10 s before each later attempt, as a bcp or a JSON load does. Each failed attempt emits a message with its number, the wait and the error.

Usage:

DBConnection_v9\$connect(attempts = 3)

Arguments:

attempts The number of attempts to connect.

DBConnection_v9\$disconnect(): Disconnect from the database.

The method closes only a connection that this process opened. A connection that another process opened stays open.

Usage:

DBConnection_v9\$disconnect()

DBConnection_v9\$clone(): The objects of this class are cloneable with this method.

Usage:

DBConnection_v9\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

See Also

The introduction vignette, `vignette("csdb", package = "csdb")`, which creates one of these, connects, and disconnects again. `csdb_set_auth_hook` registers the function that `connect()` calls after its first failed attempt.

Other database classes: [DBTable_v9](#)

Examples

```
# Creating the object stores the settings. It opens no connection,
# so this runs without a database server.
db <- DBConnection_v9$new(
  driver = "PostgreSQL Unicode",
  server = "localhost",
  port = 5432,
  db = "mydb",
  user = "myuser",
  password = "mypass"
)
db$is_connected()
db

# The full cycle, on SQLite. SQLite needs no server, so this block runs
# anywhere. Only the driver and the db argument change for a server.
# vignette("backends", package = "csdb") puts the two configurations
# side by side.
sqlite_db <- DBConnection_v9$new(
  driver = "SQLite",
  db = tempfile(fileext = ".sqlite")
)

sqlite_db$connect()
sqlite_db$is_connected()
DBI::dbListTables(sqlite_db$connection)

sqlite_db$disconnect()
sqlite_db$is_connected()

# $autoconnection opens the file again, so a read after a disconnect
# still works.
DBI::dbListTables(sqlite_db$autoconnection)
sqlite_db$disconnect()
```

Description

A database table management class that provides operations for data manipulation, schema validation, and table administration. This class combines database connectivity with data validation and bulk operations.

Details

The DBTable_v9 class is a database table abstraction that provides:

Core functionality:

- Table creation and schema management.
- Data insertion with bulk loading capabilities.
- Upsert operations (insert or update).
- Index management (creation, deletion).
- Data validation through customizable validators.
- Integration with dplyr for data queries.

Advanced features:

- Automatic table creation based on field specifications.
- Schema validation with custom validator functions.
- Efficient bulk data loading using database-specific methods.
- Index optimization for query performance.
- Cross-database compatibility (SQL Server, PostgreSQL).

Data validation: The class supports custom validation functions for both field types and data contents, which ensure data integrity and schema compliance.

What the object creates in the database

One object creates three kinds of thing, and each carries its own name rule.

The table Named table_name, in the schema that dbconfig names.

The primary key constraint Named pk_<slug>_<16 hexadecimal characters>, at most 63 characters. The rule is the index rule below, with the prefix pk_ and the logical name pk. The table identity is the table name on SQL Server, and schema.table on PostgreSQL. Two different tables get two names. Schema a with table bc and schema ab with table c therefore no longer share a name. csdb::pk_physical_name() returns the name for one table. SQLite gives the key no name.

One index per entry in indexes The names you write in indexes are logical names. Each index reaches the database under a physical name of the form ix_<slug>_<16 hexadecimal characters>, at most 63 characters. The name carries the table identity, so two tables in one schema that both declare ind1 get two indexes. csdb::index_physical_name() returns the name for one table and one logical name.

The case of a constraint name

Every constraint name and every index name is lower case. PostgreSQL folds an unquoted identifier to lower case, and SQLite and SQL Server do not. A lower case name therefore reads the same in the source and in every catalogue. A DROP CONSTRAINT can quote the name or leave it unquoted, and it finds the constraint both ways.

A table that an earlier csdb release created keeps its old constraint name, PK_ plus the table name with every ., [and] deleted. PostgreSQL stores that name in lower case. csdb does not rename it.

Indexes on a table that exists

An existing table gets its missing declared indexes on first use. The first call that reaches the table looks for the physical name of each declared index on that table. It creates each index that is absent, and nothing else. Later calls on the same object do no index work, and a direct call to create_table() is one of them. After remove_table() or keep_rows_where(), the next create_table() call checks again. Each new R process, and each new object, checks again.

A failed create gives a warning that names the table and the index. The call then continues with its read or write.

An index under an old name stays. A table that a release before 2026.10.4 indexed as ind1 then holds both ind1 and the new physical name. csdb does not drop the old index.

The check reads pg_indexes on PostgreSQL and sqlite_master on SQLite. On SQL Server one statement reads sys.indexes and creates the index when it is absent. On any other backend csdb checks nothing.

Public fields

dbconnection Database connection.

dbconfig Configuration details of the database.

table_name Name of the table in the database.

table_name_short_for_mssql_fully_specified_for_postgres Fully specified name of the table in the database (e.g. \[db\].\[dbo\].\[table_name\]).

table_name_short_for_mssql_fully_specified_for_postgres_text Fully specified name of the table in the database (e.g. \[db\].\[dbo\].\[table_name\]).

table_name_fully_specified Fully specified name of the table in the database (e.g. \[db\].\[dbo\].\[table_name\]).

table_name_fully_specified_text Fully specified name of the table in the database (e.g. \[db\].\[dbo\].\[table_name\]) as a text string.

field_types The types of each column in the database table (INTEGER, DOUBLE, TEXT, BOOLEAN, DATE, DATETIME).

field_types_with_length The same as field_types but with (100) added to the end of all TEXT fields.

keys The combination of variables that uniquely identify each row in the database.

keys_with_length The same as keys but with (100) added to the end of all TEXT fields.

indexes A named list of vectors (generally "ind1", "ind2", etc.) that improves the speed of data retrieval operations on a database table.

validator_field_contents A function that validates the data before it is inserted into the database.

`load_folder` A temporary folder that is used to write data to before inserting into the database.

`censors` A named list of censors.

Methods

Public methods:

- `DBTable_v9$new()`
- `DBTable_v9$print()`
- `DBTable_v9$connect()`
- `DBTable_v9$disconnect()`
- `DBTable_v9$table_exists()`
- `DBTable_v9$create_table()`
- `DBTable_v9$remove_table()`
- `DBTable_v9$insert_data()`
- `DBTable_v9$upsert_data()`
- `DBTable_v9$drop_all_rows()`
- `DBTable_v9$replace_all_rows()`
- `DBTable_v9$drop_rows_where()`
- `DBTable_v9$keep_rows_where()`
- `DBTable_v9$drop_all_rows_and_then_upsert_data()`
- `DBTable_v9$drop_all_rows_and_then_insert_data()`
- `DBTable_v9$tbl()`
- `DBTable_v9$print_dplyr_select()`
- `DBTable_v9$add_indexes()`
- `DBTable_v9$drop_indexes()`
- `DBTable_v9$confirm_indexes()`
- `DBTable_v9$nrow()`
- `DBTable_v9$info()`
- `DBTable_v9$clone()`

`DBTable_v9$new()`: Create a new `DBTable_v9` object.

Usage:

```
DBTable_v9$new(
  dbconfig,
  table_name,
  field_types,
  keys,
  indexes = NULL,
  validator_field_types = validator_field_types_blank,
  validator_field_contents = validator_field_contents_blank,
  dbconnection = NULL
)
```

Arguments:

dbconfig Configuration details of the database (driver, server, port, db, schema, user, password, trusted_connection, sslmode, role_create_table).

table_name Name of the table in the database.

field_types The types of each column in the database table (INTEGER, DOUBLE, TEXT, BOOLEAN, DATE, DATETIME).

keys The combination of these variables uniquely identifies each row of data in the table.

indexes A named list of vectors (generally "ind1", "ind2", etc.) that improves the speed of data retrieval operations on a database table.

validator_field_types A function that validates the field_types before the DB schema is created.

validator_field_contents A function that validates the data before it is inserted into the database.

dbconnection An existing DBConnection_v9 to use, or NULL. The object borrows a supplied connection and does not own it. disconnect() then does nothing, so the caller decides when the connection closes. The object creates and owns a connection when this argument is NULL. It is the last argument, because a subclass can forward the earlier seven positionally.

Returns: A new 'DBTable_v9' object.

DBTable_v9\$print(): Class-specific print function.

Usage:

DBTable_v9\$print(...)

Arguments:

... Not used.

DBTable_v9\$connect(): Connect to the database.

Usage:

DBTable_v9\$connect()

DBTable_v9\$disconnect(): Disconnect from the database. This does nothing when the connection came from the dbconnection argument, because the caller owns that connection.

Usage:

DBTable_v9\$disconnect()

DBTable_v9\$table_exists(): Does the table exist?

Usage:

DBTable_v9\$table_exists()

DBTable_v9\$create_table(): Create the database table. When the table exists and its fields differ from field_types, this drops the table with all its rows and creates it again.

The fields differ when the names or their order differ, as check_fields_match() compares them. Do not call this on a table whose rows you need, unless field_types matches the table.

You rarely call this yourself. connect(), tbl() and 12 other public methods call it once per object, through the private method lazy_creation_of_table(). 14 of the 22 public methods therefore can drop the table.

When the table exists and its fields match, this creates each declared index that is missing. See the section "Indexes on a table that exists".

On PostgreSQL, a field type DOUBLE creates a double precision column. A real column declared DOUBLE in a table that exists counts as matching. It becomes double precision with one ALTER TABLE, and the table keeps its rows. See the section "PostgreSQL double precision" of [DBConnection_v9](#).

Usage:

```
DBTable_v9$create_table()
```

DBTable_v9\$remove_table(): Drop the database table.

Usage:

```
DBTable_v9$remove_table()
```

DBTable_v9\$insert_data(): Inserts data into the database table.

Usage:

```
DBTable_v9$insert_data(
  newdata,
  confirm_insert_via_nrow = FALSE,
  verbose = TRUE,
  load_timeout = 3600
)
```

Arguments:

`newdata` The data to insert.

`confirm_insert_via_nrow` If TRUE, a PostgreSQL or SQL Server load that stops on a duplicate key is upserted once instead. Otherwise the method counts the rows after the insert, and upserts when the table holds fewer rows than newdata. It stops when the table still holds fewer rows than newdata after the upsert.

`verbose` Boolean.

`load_timeout` The longest time in seconds that one load MAY run. On SQL Server, csdb then kills bcp and stops with an error of class csdb_load_ambiguous, and it does not retry the load. On PostgreSQL, see the section "PostgreSQL loads" of [DBConnection_v9](#). The default of 3600 s is the 60 minutes after which Airflow kills a nrsys task.

Details: For SQL Server and MySQL, the method writes newdata to a new load file in tempdir() for each call. csdb deletes that file when the load returns, also after a failed load. A write that fails partway leaves the file in tempdir(). PostgreSQL and SQLite write no file. Called without file, the internal load methods default to a new file from tempfile(). They delete that default file in every case, also after a write that fails partway.

DBTable_v9\$upsert_data(): Upserts data into the database table.

Usage:

```
DBTable_v9$upsert_data(
  newdata,
  drop_indexes = names(self$indexes),
  verbose = TRUE,
  load_timeout = 3600
)
```

Arguments:

`newdata` The data to insert.

`drop_indexes` A vector of the indexes to drop before the upsert (can increase performance).

`verbose` Boolean.

`load_timeout` The longest time in seconds that one load MAY run. On SQL Server, `csdb` then kills `bcp` and stops with an error of class `csdb_load_ambiguous`, and it does not retry the load. On PostgreSQL, see the section "PostgreSQL loads" of [DBConnection_v9](#). The default of 3600 s is the 60 minutes after which Airflow kills a `norsyss` task.

Details: The load file follows the same rules as in `insert_data()`: a new file in `tempdir()` for each call, deleted when the load returns.

`DBTable_v9$drop_all_rows()`: Drops all rows in the database table.

Usage:

```
DBTable_v9$drop_all_rows()
```

`DBTable_v9$replace_all_rows()`: Replaces every row of the database table with `newdata`, in one transaction. When a statement fails, the table keeps its old rows.

The rows go through `DBI::dbAppendTable()`, not through `bcp` or the JSON load. Use this method for small tables.

The method stops before any statement unless `newdata` has each name in `names(field_types)` once, and no other column.

A change of `field_types` still drops and creates the table again first, through `create_table()`. The first call after a schema change can therefore start from an empty table.

Usage:

```
DBTable_v9$replace_all_rows(newdata)
```

Arguments:

`newdata` The data that replaces the rows.

`DBTable_v9$drop_rows_where()`: Drops rows in the database table according to the SQL condition.

Usage:

```
DBTable_v9$drop_rows_where(condition)
```

Arguments:

`condition` SQL text condition.

`DBTable_v9$keep_rows_where()`: Keeps rows in the database table according to the SQL condition.

Usage:

```
DBTable_v9$keep_rows_where(condition)
```

Arguments:

`condition` SQL text condition.

`DBTable_v9$drop_all_rows_and_then_upsert_data()`: Drops all rows in the database table and then upserts data.

Usage:

```
DBTable_v9$drop_all_rows_and_then_upsert_data(
  newdata,
  drop_indexes = names(self$indexes),
  verbose = TRUE,
  load_timeout = 3600
)
```

Arguments:

newdata The data to insert.

drop_indexes A vector of the indexes to drop before the upsert (can increase performance).

verbose Boolean.

load_timeout The longest time in seconds that one load MAY run. On SQL Server, csdb then kills bcp and stops with an error of class csdb_load_ambiguous, and it does not retry the load. On PostgreSQL, see the section "PostgreSQL loads" of [DBConnection_v9](#). The default of 3600 s is the 60 minutes after which Airflow kills a norsyss task.

DBTable_v9\$drop_all_rows_and_then_insert_data(): Drops all rows in the database table and then inserts data.

Usage:

```
DBTable_v9$drop_all_rows_and_then_insert_data(
  newdata,
  confirm_insert_via_nrow = FALSE,
  verbose = TRUE,
  load_timeout = 3600
)
```

Arguments:

newdata The data to insert.

confirm_insert_via_nrow If TRUE, a PostgreSQL or SQL Server load that stops on a duplicate key is upserted once instead. Otherwise the method counts the rows after the insert, and upserts when the table holds fewer rows than newdata. It stops when the table still holds fewer rows than newdata after the upsert.

verbose Boolean.

load_timeout The longest time in seconds that one load MAY run. On SQL Server, csdb then kills bcp and stops with an error of class csdb_load_ambiguous, and it does not retry the load. On PostgreSQL, see the section "PostgreSQL loads" of [DBConnection_v9](#). The default of 3600 s is the 60 minutes after which Airflow kills a norsyss task.

DBTable_v9\$tbl(): Provides access to the database table via dplyr::tbl.

Usage:

```
DBTable_v9$tbl()
```

DBTable_v9\$print_dplyr_select(): Prints a template dplyr::select call that you can copy and paste for all your variables.

Usage:

```
DBTable_v9$print_dplyr_select()
```

DBTable_v9\$add_indexes(): Adds indexes to the database table from 'self\$indexes'. Creates each index in 'self\$indexes' exactly once, even when the table does not exist yet and this call is what creates it.

The names in 'self\$indexes' are logical names. Each index reaches the database under a physical name. That name carries the table identity. Two tables in one schema that declare the same logical name therefore ask for different index names.

After each create, the method reads the catalogue. It raises when the index is absent from this table, and when the index covers columns other than the declared ones.

That check is defined for SQLite and for PostgreSQL, and for no other backend. On any other backend the method creates each index and does NOT verify it.

Usage:

```
DBTable_v9$add_indexes()
```

DBTable_v9\$drop_indexes(): Drops all indexes from the database table.

The method drops the physical name that 'add_indexes()' created, for every logical name in 'self\$indexes'. An index that a legacy release created under the logical name is not dropped here.

Usage:

```
DBTable_v9$drop_indexes()
```

DBTable_v9\$confirm_indexes(): Confirms that the database holds every index declared in 'self\$indexes', on this table, with the declared columns in the declared order.

The method never drops an index to reconcile. It takes one of four actions per declared index:

- present with the declared columns: nothing.
- absent: add it.
- present with other columns: raise.
- any index csdb did not name: ignore it.

The method reads an index definition on SQLite and on PostgreSQL only. On any other backend it checks the name alone, so it cannot see a change of columns.

Usage:

```
DBTable_v9$confirm_indexes()
```

DBTable_v9\$nrow(): Gets the number of rows in the database table.

Usage:

```
DBTable_v9$nrow(use_count = FALSE)
```

Arguments:

use_count If TRUE, then uses the count command, which is slow but accurate. If FALSE, then uses summary statistics, which is fast but inaccurate.

DBTable_v9\$info(): Gets the information about the database table.

Usage:

```
DBTable_v9$info()
```

DBTable_v9\$clone(): The objects of this class are cloneable with this method.

Usage:

```
DBTable_v9$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

The introduction vignette, `vignette("csdb", package = "csdb")`. It builds one of these on SQLite and inserts the bundled `nor_covid19_cases_by_time_location` dataset. It also shows two tables that declare one logical index name. [DBConnection_v9](#) takes the same arguments as the `dbconfig` list, and one is created here to hold the connection.

Other database classes: [DBConnection_v9](#)

Examples

```
# Creating the object opens no connection, and the field types are
# checked while it is created. These field types do not satisfy the
# csfmt_rts_data_v1 schema, so the constructor stops.
try(DBTable_v9$new(
  dbconfig = list(driver = "PostgreSQL Unicode", server = "localhost"),
  table_name = "my_data_table",
  field_types = c("id" = "INTEGER"),
  keys = "id",
  validator_field_types = validator_field_types_csfmt_rts_data_v1
))

# A full cycle on SQLite, in a file that tempfile() names. SQLite needs
# no server, so this block runs anywhere. Name a driver of
# "ODBC Driver 17 for SQL Server" or "PostgreSQL Unicode" instead, and
# nothing else in the block changes.
db_config <- list(driver = "SQLite", db = tempfile(fileext = ".sqlite"))

# Indexes are named here, because add_indexes() takes no arguments and
# reads them from the object.
my_table <- DBTable_v9$new(
  dbconfig = db_config,
  table_name = "my_data_table",
  field_types = c(
    "id" = "INTEGER",
    "name" = "TEXT",
    "value" = "DOUBLE",
    "date_created" = "DATE"
  ),
  keys = "id",
  indexes = list("ind1" = c("name", "date_created")),
  validator_field_types = validator_field_types_blank,
  validator_field_contents = validator_field_contents_blank
)

my_table$create_table()

# insert_data() and upsert_data() need a data.table.
my_table$insert_data(data.table::data.table(
  id = 1:3,
  name = c("Alice", "Bob", "Charlie"),
  value = c(10.5, 20.3, 15.7),
```

```

    date_created = as.Date("2023-01-01")
  ))

# tbl() returns a lazy dbplyr reference.
my_table$tbl() |>
  dplyr::filter(value > 15) |>
  dplyr::collect()

# Add the indexes that were named above.
my_table$add_indexes()

my_table$upsert_data(data.table::data.table(
  id = 2:4,
  name = c("Bob_Updated", "Charlie", "David"),
  value = c(25.0, 15.7, 30.2),
  date_created = as.Date("2023-01-02")
))
my_table$nrow()

my_table$disconnect()

```

get_table_names_and_info

Get table names, number of rows, and size information

Description

Gets information about database tables: their names, their row counts, and their storage sizes. The function is a generic, with one method for each supported database system.

Usage

```
get_table_names_and_info(connection)
```

Arguments

connection A database connection object, for example from [dbConnect](#).

Value

A data.table containing table information with columns:

table_name Character. Name of the table.

nrow Numeric. The row count as the database reports it. On PostgreSQL it is reltuples from pg_class, which is an estimate. On Microsoft SQL Server it is the rows column of sp_spaceused. On SQLite it is COUNT(*), which is exact rather than an estimate.

size_total_gb Numeric. Total size of the table in gigabytes. NA_real_ on SQLite.

size_data_gb Numeric. Size of data in gigabytes. NA_real_ on SQLite.

size_index_gb Numeric. Size of indexes in gigabytes. NA_real_ on SQLite.

SQLite reports no per-table size. The dbstat virtual table is the only source of one. The SQLite build that RSQLite ships does not compile dbstat in: a query against it fails with no such table: dbstat. pragma page_count and pragma page_size exist, but they describe the whole file rather than one table, so all three size columns are NA_real_.

See Also

[DBTable_v9](#), whose info() method and whose nrow(use_count = FALSE) method call this function. The introduction vignette, vignette("csdb", package = "csdb"), does not mention this function.

Examples

```
# SQLite needs no server, so this block runs anywhere. The three size
# columns are NA_real_ here, and carry a number on the other two
# backends.
con <- DBI::dbConnect(RSQLite::SQLite(), tempfile(fileext = ".sqlite"))
DBI::dbWriteTable(con, "cases", data.frame(id = 1:3, n = c(7, 8, 9)))

get_table_names_and_info(con)

DBI::dbDisconnect(con)

## Not run:
# A server backend needs a running server, so this block cannot run
# here. Methods exist for the "PostgreSQL" and "Microsoft SQL Server"
# connection classes that odbc creates.
con <- DBI::dbConnect(odbc::odbc(),
  driver = "PostgreSQL Unicode",
  server = "localhost",
  port = 5432,
  database = "mydb",
  uid = "user",
  password = "pass"
)
get_table_names_and_info(con)
DBI::dbDisconnect(con)

## End(Not run)
```

Description

Inserts one data.table into each table of a list. On PostgreSQL, one transaction loads every table, so by default the call stores every row or no row. The one exception is a duplicate key with confirm_insert_via_nrow = TRUE: csdb then inserts table by table.

Usage

```
insert_data_many(
  tables,
  newdata,
  confirm_insert_via_nrow = FALSE,
  verbose = TRUE,
  load_timeout = 3600
)
```

Arguments

tables	A list of DBTable_v9 objects.
newdata	A list of data.tables, one for each table, in the order of tables.
confirm_insert_via_nrow	See DBTable_v9\$insert_data().
verbose	See DBTable_v9\$insert_data().
load_timeout	The longest time in seconds that the load MAY run. On PostgreSQL, the server cancels the statement that runs past this time and stores no row, and the error has class csdb_load_other.

Details

On PostgreSQL, one transaction on the connection of the first table loads every table. csdb sends the rows as JSON text, at most 500,000 rows in one statement, and starts no client. A table whose data.table is NULL or has no rows gets no statement. After each statement, csdb compares the stored rows with the rows sent. A difference raises an error before the commit, so the server stores no row in any table.

The transaction holds a lock on every table until it commits, so a TRUNCATE or an index change on one of them waits. The transaction waits at most 60 s for one lock. After a longer wait, a deadlock or a serialization failure, csdb runs the whole load again, at most 3 attempts in total. A load whose connection breaks, or whose COMMIT fails, does not run again. The error then says to count the rows first.

A duplicate key stops the load. With confirm_insert_via_nrow = TRUE, csdb then calls \$insert_data(confirm_insert_via_nrow = TRUE) on each table, which upserts a table that stops on the duplicate key.

With another driver, or with tables that have different connection settings, csdb calls \$insert_data() on each table in turn.

Value

NULL, invisibly.

Examples

```
# SQLite has no JSON load, so csdb inserts into each table in turn here.
db_config <- list(driver = "SQLite", db = tempfile(fileext = ".sqlite"))
connection <- DBConnection_v9$new(driver = "SQLite", db = db_config$db)
tables <- lapply(c("cases_a", "cases_b"), function(name) {
  DBTable_v9$new(
    dbconfig = db_config,
    table_name = name,
    field_types = c("id" = "INTEGER", "n" = "INTEGER"),
    keys = "id",
    dbconnection = connection
  )
})
insert_data_many(
  tables,
  list(
    data.table::data.table(id = 1:2, n = c(5L, 7L)),
    data.table::data.table(id = 1L, n = 3L)
  )
)
tables[[1]]$nrow()
connection$disconnect()
```

nor_covid19_cases_by_time_location

Covid-19 data for PCR-confirmed cases in Norway (nation and county)

Description

This data comes from the Norwegian Surveillance System for Communicable Diseases (MSIS). The date corresponds to when the PCR-test was taken.

Usage

```
nor_covid19_cases_by_time_location
```

Format

A data.table with 11028 rows and 18 variables:

granularity_time day/isoweek

granularity_geo nation, county

country_iso3 nor

location_code norge, 11 counties

border 2020

age total
sex total
isoyear Isoyear of event
isoweek Isoweek of event
isoyearweek Isoyearweek of event
season Season of event
seasonweek Seasonweek of event
calyear Calyear of event
calmonth Calmonth of event
calyearmonth Calyearmonth of event
date Date of event
covid19_cases_testdate_n Number of confirmed covid19 cases
covid19_cases_testdate_pr100000 Number of confirmed covid19 cases per 100.000 population

Details

The raw number of cases and cases per 100.000 population are recorded.
 This data was extracted on 2022-05-04.

Source

https://github.com/folkehelseinstituttet/surveillance_data/blob/master/covid19/_DOCUMENTATION_data_covid19_msis_by_time_location.txt

Examples

```
head(nor_covid19_cases_by_time_location)
nrow(nor_covid19_cases_by_time_location)
```

validator_field_contents_blank
Blank data contents validator

Description

A pass-through validator that accepts any data without validation. Use it as a placeholder when you need no check on the data contents.

Usage

```
validator_field_contents_blank(data)
```

Arguments

data A data.frame or data.table containing the data to validate.

Value

Always returns TRUE.

See Also

The introduction vignette, `vignette("csdb", package = "csdb")`, which passes this to `DBTable_v9$new()` as its `validator_field_contents` argument. `DBTable_v9` calls the field contents validator from its `insert_data()` and `upsert_data()` methods.

Other field contents validators: `validator_field_contents_csfmt_rts_data_v1()`, `validator_field_contents_csfmt_rts_data_v2()`

Examples

```
# This validator always returns TRUE regardless of input
test_data <- data.frame(id = 1:3, name = c("A", "B", "C"), value = c(10, 20, 30))
validator_field_contents_blank(test_data)

# Works with any data structure
empty_data <- data.frame()
validator_field_contents_blank(empty_data)
```

`validator_field_contents_csfmt_rts_data_v1`

Field contents validator for csfmt_rts_data_v1 schema

Description

Checks that data contents conform to the `csfmt_rts_data_v1` schema specification. The validator checks the `granularity_time`, `granularity_geo`, `border`, `sex`, and `date` fields against the values that schema allows.

Usage

```
validator_field_contents_csfmt_rts_data_v1(data)
```

Arguments

`data` A `data.frame` or `data.table` containing the data to validate.

Value

TRUE if data is valid for `csfmt_rts_data_v1`, FALSE otherwise (with error attribute).

See Also

`DBTable_v9`, which takes this as its `validator_field_contents` argument and calls it from its `insert_data()` and `upsert_data()` methods. The introduction vignette, `vignette("csdb", package = "csdb")`, does not use this validator.

Other field contents validators: `validator_field_contents_blank()`, `validator_field_contents_csfmt_rts_data_v1()`, `validator_field_contents_csfmt_rts_data_v2()`

Examples

```
# Valid data for csfmt_rts_data_v1 (all required columns present)
valid_data <- data.frame(
  granularity_time = c("date", "isoyearweek", "total"),
  granularity_geo = c("nation", "county", "municip"),
  border = c("2020", "2020", "2020"),
  sex = c("total", "total", "total"),
  date = as.Date(c("2020-01-01", "2020-01-08", "2020-01-01")),
  stringsAsFactors = FALSE
)
validator_field_contents_csfmt_rts_data_v1(valid_data)

# Invalid data (unrecognised granularity_geo value)
invalid_data <- data.frame(
  granularity_time = "date",
  granularity_geo = "invalid_geo",
  border = "2020",
  sex = "total",
  date = as.Date("2020-01-01"),
  stringsAsFactors = FALSE
)
validator_field_contents_csfmt_rts_data_v1(invalid_data)
```

 validator_field_contents_csfmt_rts_data_v2

Field contents validator for csfmt_rts_data_v2 schema

Description

Checks that data contents conform to the csfmt_rts_data_v2 schema specification. The validator checks the granularity_time, granularity_geo, border, sex, and date fields against the values that schema allows.

Usage

```
validator_field_contents_csfmt_rts_data_v2(data)
```

Arguments

data A data.frame or data.table containing the data to validate.

Value

TRUE if data is valid for csfmt_rts_data_v2, FALSE otherwise (with error attribute).

See Also

[DBTable_v9](#), which takes this as its `validator_field_contents` argument and calls it from its `insert_data()` and `upsert_data()` methods. The introduction vignette, `vignette("csdb", package = "csdb")`, does not use this validator.

Other field contents validators: [validator_field_contents_blank\(\)](#), [validator_field_contents_csfmt_rts_data_v](#)

Examples

```
# Valid data for csfmt_rts_data_v2 (all required columns present)
valid_data_v2 <- data.frame(
  granularity_time = c("date", "isoyearweek", "total"),
  granularity_geo = c("nation", "county", "municip"),
  border = c("2020", "2020", "2020"),
  sex = c("total", "total", "total"),
  date = as.Date(c("2020-01-01", "2020-01-08", "2020-01-01")),
  stringsAsFactors = FALSE
)
validator_field_contents_csfmt_rts_data_v2(valid_data_v2)

# Invalid data (unrecognised granularity_geo value)
invalid_data_v2 <- data.frame(
  granularity_time = "date",
  granularity_geo = "invalid_geo",
  border = "2020",
  sex = "total",
  date = as.Date("2020-01-01"),
  stringsAsFactors = FALSE
)
validator_field_contents_csfmt_rts_data_v2(invalid_data_v2)
```

validator_field_types_blank

Blank field types validator

Description

A pass-through validator that accepts any field types without validation. Use it as a placeholder when you need no check on the field types.

Usage

```
validator_field_types_blank(db_field_types)
```

Arguments

`db_field_types` A named character vector of database field types.

Value

Always returns TRUE.

See Also

The introduction vignette, `vignette("csdb", package = "csdb")`, which passes this to `DBTable_v9$new()` as its `validator_field_types` argument. `DBTable_v9` calls the field type validator once, while the object is being created.

Other field type validators: `validator_field_types_csfmt_rts_data_v1()`, `validator_field_types_csfmt_rts_data_v2()`

Examples

```
# This validator always returns TRUE regardless of input
field_types <- c("id" = "INTEGER", "name" = "TEXT", "date" = "DATE")
validator_field_types_blank(field_types)

# Works with any field types
other_types <- c("value" = "DOUBLE", "status" = "BOOLEAN")
validator_field_types_blank(other_types)
```

```
validator_field_types_csfmt_rts_data_v1
      Field types validator for csfmt_rts_data_v1 schema
```

Description

Checks that field types conform to the `csfmt_rts_data_v1` schema specification. The validator checks the first 16 entries of `db_field_types` against the expected structure of that schema.

Usage

```
validator_field_types_csfmt_rts_data_v1(db_field_types)
```

Arguments

`db_field_types` A named character vector of database field types.

Value

TRUE if field types are valid for `csfmt_rts_data_v1`, FALSE otherwise.

See Also

`DBTable_v9`, which takes this as its `validator_field_types` argument and calls it once, while the object is being created. The introduction vignette, `vignette("csdb", package = "csdb")`, does not use this validator.

Other field type validators: `validator_field_types_blank()`, `validator_field_types_csfmt_rts_data_v2()`

Examples

```
# Valid field types for csfmt_rts_data_v1
valid_fields <- c(
  "granularity_time" = "TEXT",
  "granularity_geo" = "TEXT",
  "country_iso3" = "TEXT",
  "location_code" = "TEXT",
  "border" = "INTEGER",
  "age" = "TEXT",
  "sex" = "TEXT",
  "isoyear" = "INTEGER",
  "isoweek" = "INTEGER",
  "isoyearweek" = "TEXT",
  "season" = "TEXT",
  "seasonweek" = "DOUBLE",
  "calyear" = "INTEGER",
  "calmonth" = "INTEGER",
  "calyearmonth" = "TEXT",
  "date" = "DATE",
  "cases_n" = "INTEGER"
)
validator_field_types_csfmt_rts_data_v1(valid_fields)

# Invalid field types (wrong structure)
invalid_fields <- c("id" = "INTEGER", "name" = "TEXT")
validator_field_types_csfmt_rts_data_v1(invalid_fields)
```

```
validator_field_types_csfmt_rts_data_v2
```

Field types validator for csfmt_rts_data_v2 schema

Description

Checks that field types conform to the `csfmt_rts_data_v2` schema specification. The validator checks the first 18 entries of `db_field_types` against the expected structure of that schema.

Usage

```
validator_field_types_csfmt_rts_data_v2(db_field_types)
```

Arguments

`db_field_types` A named character vector of database field types.

Value

TRUE if field types are valid for `csfmt_rts_data_v2`, FALSE otherwise.

See Also

[DBTable_v9](#), which takes this as its `validator_field_types` argument and calls it once, while the object is being created. The introduction vignette, `vignette("csdb", package = "csdb")`, does not use this validator.

Other field type validators: [validator_field_types_blank\(\)](#), [validator_field_types_csfmt_rts_data_v1\(\)](#)

Examples

```
# Valid field types for csfmt_rts_data_v2. The first 18 must match the
# schema, which unlike v1 carries isoquarter and isoyearquarter.
valid_fields_v2 <- c(
  "granularity_time" = "TEXT",
  "granularity_geo" = "TEXT",
  "country_iso3" = "TEXT",
  "location_code" = "TEXT",
  "border" = "INTEGER",
  "age" = "TEXT",
  "sex" = "TEXT",
  "isoyear" = "INTEGER",
  "isoweek" = "INTEGER",
  "isoyearweek" = "TEXT",
  "isoquarter" = "INTEGER",
  "isoyearquarter" = "TEXT",
  "season" = "TEXT",
  "seasonweek" = "DOUBLE",
  "calyear" = "INTEGER",
  "calmonth" = "INTEGER",
  "calyearmonth" = "TEXT",
  "date" = "DATE",
  "tag_outcome" = "TEXT",
  "tag_type" = "TEXT",
  "cases_n" = "INTEGER"
)
validator_field_types_csfmt_rts_data_v2(valid_fields_v2)

# The v1 layout is not valid for v2: it has no isoquarter
validator_field_types_csfmt_rts_data_v2(valid_fields_v2[-c(11, 12)])
```

Index

- * **auth hook functions**
 - csdb_get_auth_hook, [2](#)
 - csdb_set_auth_hook, [4](#)
 - * **caller context functions**
 - csdb_get_context, [3](#)
 - csdb_set_context, [5](#)
 - * **database classes**
 - DBConnection_v9, [7](#)
 - DBTable_v9, [11](#)
 - * **datasets**
 - nor_covid19_cases_by_time_location, [24](#)
 - * **field contents validators**
 - validator_field_contents_blank, [25](#)
 - validator_field_contents_csfmt_rts_data_v1, [26](#)
 - validator_field_contents_csfmt_rts_data_v2, [27](#)
 - * **field type validators**
 - validator_field_types_blank, [28](#)
 - validator_field_types_csfmt_rts_data_v1, [29](#)
 - validator_field_types_csfmt_rts_data_v2, [30](#)
 - * **password hook functions**
 - csdb_get_password_hook, [4](#)
 - csdb_set_password_hook, [6](#)
- csdb_get_auth_hook, [2](#)
csdb_get_auth_hook(), [5](#)
csdb_get_context, [3](#)
csdb_get_context(), [6](#)
csdb_get_password_hook, [4](#)
csdb_get_password_hook(), [7](#)
csdb_set_auth_hook, [4](#), [7](#), [11](#)
csdb_set_auth_hook(), [2](#)
csdb_set_context, [3](#), [5](#)
csdb_set_context(), [3](#)
csdb_set_password_hook, [4](#), [6](#)
csdb_set_password_hook(), [4](#)
- dbConnect, [21](#)
DBConnection_v9, [2](#), [5](#), [7](#), [16–18](#), [20](#)
DBTable_v9, [11](#), [11](#), [22](#), [26](#), [28](#), [29](#), [31](#)
- get_table_names_and_info, [21](#)
- insert_data_many, [8](#), [22](#)
- nor_covid19_cases_by_time_location, [24](#)
- validator_field_contents_blank, [25](#)
validator_field_contents_blank(), [26](#), [28](#)
validator_field_contents_csfmt_rts_data_v1, [26](#)
validator_field_contents_csfmt_rts_data_v1(), [26](#), [28](#)
validator_field_contents_csfmt_rts_data_v2, [27](#)
validator_field_contents_csfmt_rts_data_v2(), [26](#)
validator_field_types_blank, [28](#)
validator_field_types_blank(), [29](#), [31](#)
validator_field_types_csfmt_rts_data_v1, [29](#)
validator_field_types_csfmt_rts_data_v1(), [29](#), [31](#)
validator_field_types_csfmt_rts_data_v2, [30](#)
validator_field_types_csfmt_rts_data_v2(), [29](#)