Package 'RSQLite.toolkit'

May 7, 2026

Type Package

Title Load Data in SQLite from Tabular Files

Version 0.1.2

Description A lightweight wrapper around the 'RSQLite' package for streamlined loading of data from tabular files (i,e. text delimited files like Comma Separated Values and Tab Separated Values, Microsoft Excel, and Arrow Inter-process Communication files) in 'SQLite' databases. Includes helper functions for inspecting the structure of the input files, and some functions to simplify activities on the 'SQLite' tables.

License GPL (>= 3)

Depends R (>= 4.2.0), RSQLite (>= 2.3.0)

Imports DBI, openxlsx2, arrow

Encoding UTF-8

RoxygenNote 7.3.3

URL https://github.com/fab-algo/RSQLite.toolkit, https://fab-algo.github.io/RSQLite.toolkit/

BugReports https://github.com/fab-algo/RSQLite.toolkit/issues

Suggests knitr, rmarkdown, piggyback

VignetteBuilder knitr

NeedsCompilation no

Author Ludovico G. Beretta [aut, cre, cph]

Maintainer Ludovico G. Beretta <ludovicogiovanni.beretta@gmail.com>

Repository CRAN

Date/Publication 2026-04-04 08:40:02 UTC

Contents

`dbCopyTable` *Copy a table from one SQLite database to another*

Description

The `dbCopyTable()` function can be used to create a copy of the data in a table of a SQLite database in another database. The data can be appended to an already existing table (with the same name of the source one), or a new table can be created. It is possible to move also the indexes from source to target.

Usage

```
dbCopyTable(
  db_file_src,
  db_file_tgt,
  table_name,
  drop_table = FALSE,
  copy_indexes = FALSE
)
```

Arguments

`db_file_src`	character, the file name (including path) of the source database containing the table to be copied.
`db_file_tgt`	character, the file name (including path) of the target database where the table will be copied.
`table_name`	character, the table name.
`drop_table`	logical, if `TRUE` the table in the target database will be dropped (if exists) before copying the data. If `FALSE`, the data will be appended to an existing table in the target database. Defaults to `FALSE`.
`copy_indexes`	logical, if `TRUE` and also `drop_table` is `TRUE`, all indexes defined on the source table will be created on the target table. Defaults to `FALSE`.

Value

nothing

Examples

```
db_source <- tempfile(fileext = ".sqlite")
db_target <- tempfile(fileext = ".sqlite")

# Load some sample data
dbcon <- dbConnect(RSQLite::SQLite(), db_source)

data_path <- system.file("extdata", package = "RSQLite.toolkit")
dbTableFromDSV(
  input_file = file.path(data_path, "abalone.csv"),
  dbcon = dbcon,
  table_name = "ABALONE",
  drop_table = TRUE,
  auto_pk = TRUE,
  header = TRUE,
  sep = ",",
  dec = "."
)

dbDisconnect(dbcon)

# Copy the table to a new database, recreating it
# if it already exists and copying indexes
dbCopyTable(
  db_file_src = db_source,
  db_file_tgt = db_target,
  table_name = "ABALONE",
  drop_table = TRUE,        # Recreate table if it exists
  copy_indexes = TRUE       # Copy indexes too
)

# Check that the table was copied correctly
dbcon_tgt <- dbConnect(RSQLite::SQLite(), db_target)
print(dbListTables(dbcon_tgt))
print(dbListFields(dbcon_tgt, "ABALONE"))
print(dbGetQuery(dbcon_tgt, "SELECT COUNT(*) AS TOTAL_ROWS FROM ABALONE;"))
dbDisconnect(dbcon_tgt)

# Clean up temporary database files
unlink(c(db_source, db_target))
```

---

dbCreatePK *Creates a unique index on a table in a SQLite database*

Description

The dbCreatePK() function creates a UNIQUE INDEX named <table_name>_PK on the table specified by table_name in the database connected by dbcon. The index is created on the fields specified in the pk_fields argument.

Usage

```
dbCreatePK(dbcon, table_name, pk_fields, drop_index = FALSE)
```

Arguments

dbcon	database connection, as created by the dbConnect function.
table_name	character, the name of the table where the index will be created.
pk_fields	character vector, the list of the fields' names that define the UNIQUE INDEX.
drop_index	logical, if TRUE the index named <table_name>_PK will be dropped (if exists) before recreating it. If FALSE, it will check if an index with that name exists and eventually stops. Default to FALSE.

Value

nothing

Examples

```
# Create a database and table, then add a primary key
library(RSQLite.toolkit)

# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Load sample data
data_path <- system.file("extdata", package = "RSQLite.toolkit")

dbTableFromFeather(
  input_file = file.path(data_path, "penguins.feather"),
  dbcon = dbcon, table_name = "PENGUINS",
  drop_table = TRUE
)

dbGetQuery(dbcon, "select species, sex, body_mass_g,
                   culmen_length_mm, culmen_depth_mm
                   from PENGUINS
                   group by species, sex, body_mass_g,
                   culmen_length_mm, culmen_depth_mm
                   having count(*) > 1")

# Create a primary key on multiple fields
dbCreatePK(dbcon, "PENGUINS",
           c("species", "sex", "body_mass_g",
             "culmen_length_mm", "culmen_depth_mm"))
```

```
# Check that the index was created
dbGetQuery(dbcon,
  "SELECT name, sql FROM sqlite_master WHERE type='index' AND tbl_name='PENGUINS'")

# Clean up
dbDisconnect(dbcon)
```

---

dbExecFile *Execute SQL statements from a text file*

Description

The dbExecFile() function executes the SQL statements contained in a text file.

This function reads the text in input_file, strips all comment lines (i.e. all lines beginning with -- characters) and splits the SQL statements assuming that they are separated by the ; character. The list of SQL statements is then executed, one at a time; the results of each statement are stored in a list with length equal to the number of statements.

Usage

```
dbExecFile(input_file, dbcon, plist = NULL)
```

Arguments

input_file	the file name (including path) containing the SQL statements to be executed
dbcon	database connection, as created by the dbConnect function.
plist	a list with values to be binded to the parameters of SQL statements. It should have the same length as the number of SQL statements. If any of the statements do not require parameters, the corresponding element of the list should be set to NULL. If no statements require parameters, plist can be set to NULL.

Value

a list with the results returned by each statement executed.

Examples

```
# Create a database and execute SQL from a file
library(RSQLite.toolkit)

# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Load some sample data
data_path <- system.file("extdata", package = "RSQLite.toolkit")
dbTableFromDSV(
```

```
  input_file = file.path(data_path, "abalone.csv"),
  dbcon = dbcon,
  table_name = "ABALONE",
  drop_table = TRUE,
  auto_pk = TRUE,
  header = TRUE,
  sep = ",",
  dec = "."
)

# Create a SQL file with multiple statements
sql_content <- "
-- Create a summary table
DROP TABLE IF EXISTS ABALONE_SUMMARY;

CREATE TABLE ABALONE_SUMMARY AS
SELECT SEX,
       COUNT(*) as TOTAL_COUNT,
       ROUND(AVG(LENGTH), 3) as AVG_LENGTH,
       ROUND(AVG(WHOLE), 3) as AVG_WEIGHT
FROM ABALONE
GROUP BY SEX;

-- Query the results
SELECT * FROM ABALONE_SUMMARY ORDER BY SEX;

-- Parameterized query example
SELECT SEX, COUNT(*) as COUNT
FROM ABALONE
WHERE LENGTH > :min_length
GROUP BY SEX;
"

sql_file <- tempfile(fileext = ".sql")
writeLines(sql_content, sql_file)

# Execute SQL statements with parameters
plist <- list(
  NULL,  # DROP TABLE statement (no parameters)
  NULL,  # CREATE TABLE statement (no parameters)
  NULL,  # First SELECT (no parameters)
  list(min_length = 0.5)  # Parameterized SELECT
)

results <- dbExecFile(
  input_file = sql_file,
  dbcon = dbcon,
  plist = plist
)

# Check results
print(results[[3]])  # Summary data
print(results[[4]])  # Filtered data
```

```
# Clean up
unlink(sql_file)
dbDisconnect(dbcon)
```

`dbTableFromDataFrame` *Create a table in a SQLite database from a data frame*

Description

The `dbTableFromDataFrame()` function reads the data from a rectangula region of a sheet in an Excel file and copies it to a table in a SQLite database. If table does not exist, it will create it.

Usage

```
dbTableFromDataFrame(
  df,
  dbcon,
  table_name,
  id_quote_method = "DB_NAMES",
  col_names = NULL,
  col_types = NULL,
  drop_table = FALSE,
  auto_pk = FALSE,
  build_pk = FALSE,
  pk_fields = NULL
)
```

Arguments

`df`	the data frame to be saved in the SQLite table.
`dbcon`	database connection, as created by the dbConnect function.
`table_name`	character, the name of the table.
`id_quote_method`	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the `quote_method` parameter of the `format_column_names()` function. Defautls to `DB_NAMES`.
`col_names`	character vector, names of the columuns to be imported. Used to override the field names derived from the data frame (using the quote method selected by `id_quote_method`). Must be of the same length of the number of columns in the data frame. If `NULL` the column names coming from the input (after quoting) will be used. Defaults to `NULL`.

col_types	character vector of classes to be assumed for the columns. If not null, it will override the data types inferred from the input data frame. Must be of the same length of the number of columns in the input. If NULL the data type inferred from the input will be used. Defaults to NULL.
drop_table	logical, if TRUE the target table will be dropped (if exists) and recreated before importing the data. if FALSE, data from input data frame will be appended to an existing table. Defaults to FALSE.
auto_pk	logical, if TRUE, and pk_fields parameter is NULL, an additional column named SEQ will be added to the table and it will be defined to be INTEGER PRIMARY KEY (i.e. in effect an alias for ROWID). Defaults to FALSE.
build_pk	logical, if TRUE creates a UNIQUE INDEX named <table_name>_PK defined by the combination of fields specified in the pk_fields parameter. It will be effective only if pk_fields is not null. Defaults to FALSE.
pk_fields	character vector, the list of the fields' names that define the UNIQUE INDEX. Defaults to NULL.

Value

integer, the number of records in table_name after reading data from the data frame.

Examples

```
# Create a temporary database and load data frame
# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Create a sample data frame
sample_data <- data.frame(
  id = 1:10,
  name = paste0("Item_", 1:10),
  value = runif(10, 1, 100),
  active = c(TRUE, FALSE),
  date = Sys.Date() + 0:9,
  stringsAsFactors = FALSE,
  row.names = NULL
)

# Load data frame with automatic primary key
dbTableFromDataFrame(
  df = sample_data,
  dbcon = dbcon,
  table_name = "SAMPLE_DATA",
  drop_table = TRUE,
  auto_pk = TRUE
)

# Check the imported data
dbListFields(dbcon, "SAMPLE_DATA")
dbGetQuery(dbcon, "SELECT * FROM SAMPLE_DATA LIMIT 5")
```

```
# Load with column selection and custom naming
dbTableFromDataFrame(
  df = sample_data,
  dbcon = dbcon,
  table_name = "SAMPLE_SUBSET",
  drop_table = TRUE,
  col_names = c("ID", "ITEM_NAME", "ITEM_VALUE", "IS_ACTIVE", "DATE_CREATED")
)

dbGetQuery(dbcon, "SELECT * FROM SAMPLE_SUBSET LIMIT 5")

# Clean up
dbDisconnect(dbcon)
```

---

dbTableFromDSV *Create a table from a delimiter separated values (DSV) text file*

Description

The dbTableFromDSV() function reads the data from a DSV file and copies it to a table in a SQLite database. If table does not exist, it will create it.

The dbTableFromDSV() function reads the data from a DSV file and copies it to a table in a SQLite database. If table does not exist, it will create it.

Usage

```
dbTableFromDSV(
  input_file,
  dbcon,
  table_name,
  header = TRUE,
  sep = ",",
  dec = ".",
  grp = "",
  id_quote_method = "DB_NAMES",
  col_names = NULL,
  col_types = NULL,
  col_import = NULL,
  drop_table = FALSE,
  auto_pk = FALSE,
  build_pk = FALSE,
  pk_fields = NULL,
  constant_values = NULL,
  chunk_size = 0,
  ...
)
```

Arguments

input_file	character, the file name (including path) to be read.
dbcon	database connection, as created by the dbConnect function.
table_name	character, the name of the table.
header	logical, if TRUE the first line contains the columns' names. If FALSE, the columns' names will be formed sing a "V" followed by the column number (as specified in utils::read.table()).
sep	character, field delimiter (e.g., "," for CSV, "\t" for TSV) in the input file. Defaults to ",".
dec	character, decimal separator (e.g., "." or "," depending on locale) in the input file. Defaults to ".".
grp	character, character used for digit grouping. It defaults to "" (i.e. no grouping).
id_quote_method	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the quote_method parameter of the format_column_names() function. Defautls to DB_NAMES.
col_names	character vector, names of the columuns in the input file. Used to override the field names derived from the input file (using the quote method selected by id_quote_method). Must be of the same length of the number of columns in the input file. If NULL the column names coming from the input file (after quoting) will be used. Defaults to NULL.
col_types	character vector of classes to be assumed for the columns of the input file. Must be of the same length of the number of columns in the input file. If not null, it will override the data types guessed from the input file. If NULL the data type inferred from the input files will be used. Defaults to NULL.
col_import	can be either: • a numeric vector (coherced to integers) with the columns' positions in the input file that will be imported in the SQLite table; • a character vector with the columns' names to be imported. The names are those in the input file (after quoting with id_quote_method), if col_names is NULL, or those expressed in col_names vector. Defaults to NULL, i.e. all columns will be imported.
drop_table	logical, if TRUE the target table will be dropped (if exists) and recreated before importing the data. if FALSE, data from input file will be appended to an existing table. Defaults to FALSE.
auto_pk	logical, if TRUE, and pk_fields parameter is NULL, an additional column named SEQ will be added to the table and it will be defined to be INTEGER PRIMARY KEY (i.e. in effect an alias for ROWID). Defaults to FALSE.
build_pk	logical, if TRUE creates a UNIQUE INDEX named <table_name>_PK defined by the combination of fields specified in the pk_fields parameter. It will be effective only if pk_fields is not null. Defaults to FALSE.
pk_fields	character vector, the list of the fields' names that define the UNIQUE INDEX. Defaults to NULL.

`constant_values`
: a one row data frame whose columns will be added to the table in the database. The additional table columns will be named as the data frame columns, and the corresponding values will be associeted to each record imported from the input file. It is useful to keep track of additional information (e.g., the input file name, additional context data not available in the data set, ...) when loading the content of multiple input files in the same table.

`chunk_size`
: integer, the number of lines in each "chunk" (i.e. block of lines from the input file). Setting its value to a positive integer number, will process the input file by blocks of `chunk_size` lines, avoiding to read all the data in memory at once. It can be useful for very large size files. If set to zero, it will process the whole text file in one pass. Default to zero.

`...`
: additional arguments passed to `base::scan()` function used to read input data. Please note that if the `quote` parameter is not specified, it will be set to `""` (i.e., no quoting) by default.

Value

integer, the number of records in `table_name` after reading data from `input_file`.

Examples

```
# Create a temporary database and load CSV data
library(RSQLite.toolkit)

# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Get path to example data
data_path <- system.file("extdata", package = "RSQLite.toolkit")

# Load abalone CSV data with automatic primary key
dbTableFromDSV(
  input_file = file.path(data_path, "abalone.csv"),
  dbcon = dbcon,
  table_name = "ABALONE",
  drop_table = TRUE,
  auto_pk = TRUE,
  header = TRUE,
  sep = ",",
  dec = "."
)

# Check the imported data
dbListFields(dbcon, "ABALONE")
head(dbGetQuery(dbcon, "SELECT * FROM ABALONE"))

# Load data with specific column selection
dbTableFromDSV(
  input_file = file.path(data_path, "abalone.csv"),
  dbcon = dbcon,
```

```
  table_name = "ABALONE_SUBSET",
  drop_table = TRUE,
  header = TRUE,
  sep = ",",
  dec = ".",
  col_import = c("Sex", "Length", "Diam", "Whole")
)

head(dbGetQuery(dbcon, "SELECT * FROM ABALONE_SUBSET"))

# Check available tables
dbListTables(dbcon)

# Clean up
dbDisconnect(dbcon)
```

`dbTableFromFeather` *Create a table from a Feather (Arrow IPC) file*

Description

The `dbTableFromFeather()` function reads the data from a Feather (Arrow IPC) file and copies it to a table in a SQLite database. If table does not exist, it will create it.

The `dbTableFromFeather()` function reads the data from an Apache Arrow table serialized in a Feather (Arrow IPC) file and copies it to a table in a SQLite database. If table does not exist, it will create it.

Usage

```
dbTableFromFeather(
  input_file,
  dbcon,
  table_name,
  id_quote_method = "DB_NAMES",
  col_names = NULL,
  col_types = NULL,
  col_import = NULL,
  drop_table = FALSE,
  auto_pk = FALSE,
  build_pk = FALSE,
  pk_fields = NULL,
  constant_values = NULL
)
```

Arguments

input_file	character, the file name (including path) to be read.
dbcon	database connection, as created by the dbConnect function.
table_name	character, the name of the table.
id_quote_method	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the quote_method parameter of the format_column_names() function. Defaults to DB_NAMES.
col_names	character vector, names of the columuns in the input file. Used to override the field names derived from the input file (using the quote method selected by id_quote_method). Must be of the same length of the number of columns in the input file. If NULL the column names coming from the input file (after quoting) will be used. Defaults to NULL.
col_types	character vector of classes to be assumed for the columns of the input file. Must be of the same length of the number of columns in the input file. If not null, it will override the data types guessed from the input file. If NULL the data type inferred from the input files will be used. Defaults to NULL.
col_import	can be either: • a numeric vector (coherced to integers) with the columns' positions in the input file that will be imported in the SQLite table; • a character vector with the columns' names to be imported. The names are those in the input file (after quoting with id_quote_method), if col_names is NULL, or those expressed in col_names vector. Defaults to NULL, i.e. all columns will be imported.
drop_table	logical, if TRUE the target table will be dropped (if exists) and recreated before importing the data. If FALSE, data from the input file will be appended to an existing table. Defaults to FALSE.
auto_pk	logical, if TRUE, and pk_fields parameter is NULL, an additional column named SEQ will be added to the table and it will be defined to be INTEGER PRIMARY KEY (i.e. in effect an alias for ROWID). Defaults to FALSE.
build_pk	logical, if TRUE creates a UNIQUE INDEX named <table_name>_PK defined by the combination of fields specified in the pk_fields parameter. It will be effective only if pk_fields is not null. Defaults to FALSE.
pk_fields	character vector, the list of the fields' names that define the UNIQUE INDEX. Defaults to NULL.
constant_values	a one row data frame whose columns will be added to the table in the database. The additional table columns will be named as the data frame columns, and the corresponding values will be associated to each record imported from the input file. It is useful to keep track of additional information (e.g., the input file name, additional context data not available in the data set, ...) when loading the content of multiple input files in the same table. Defults to NULL.

Value

integer, the number of records in `table_name` after reading data from `input_file`.

Examples

```
# Create a temporary database and load Feather data
library(RSQLite.toolkit)

# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Get path to example data
data_path <- system.file("extdata", package = "RSQLite.toolkit")

# Load penguins Feather data
dbTableFromFeather(
  input_file = file.path(data_path, "penguins.feather"),
  dbcon = dbcon,
  table_name = "PENGUINS",
  drop_table = TRUE
)

# Check the imported data
dbListFields(dbcon, "PENGUINS")
head(dbGetQuery(dbcon, "SELECT * FROM PENGUINS"))

# Load with custom column selection and types
dbTableFromFeather(
  input_file = file.path(data_path, "penguins.feather"),
  dbcon = dbcon,
  table_name = "PENGUINS_SUBSET",
  drop_table = TRUE,
  col_import = c("species", "flipper_length_mm", "body_mass_g", "sex")
)

# Check the imported data
dbListFields(dbcon, "PENGUINS_SUBSET")
head(dbGetQuery(dbcon, "SELECT * FROM PENGUINS_SUBSET"))

# Check available tables
dbListTables(dbcon)

# Clean up
dbDisconnect(dbcon)
```

`dbTableFromView` *Create a table in a SQLite database from a view*

Description

The dbTableFromView() function creates a table in a SQLite database from a view already present in the same database.

Usage

```
dbTableFromView(
  view_name,
  dbcon,
  table_name,
  drop_table = FALSE,
  build_pk = FALSE,
  pk_fields = NULL
)
```

Arguments

view_name	character, name of the view.
dbcon	database connection, as created by the dbConnect function.
table_name	character, the name of the table.
drop_table	logical, if TRUE the target table will be dropped (if exists) and recreated when importing the data. if FALSE, data from input file will be appended to an existing table. Defaults to FALSE.
build_pk	logical, if TRUE creates a UNIQUE INDEX named <table_name>_PK defined by the combination of fields specified in the pk_fields parameter. It will be effective only if pk_fields is not null. Defaults to FALSE.
pk_fields	character vector, the list of the fields' names that define the UNIQUE INDEX. Defaults to NULL.

Value

integer, the number of records in table_name after writing data from the input view.

Examples

```
# Create a temporary database and demonstrate view to table conversion
library(RSQLite.toolkit)

# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Load some sample data first
data_path <- system.file("extdata", package = "RSQLite.toolkit")
dbTableFromDSV(
  input_file = file.path(data_path, "abalone.csv"),
  dbcon = dbcon,
  table_name = "ABALONE",
  drop_table = TRUE,
```

```
  header = TRUE,
  sep = ",",
  dec = "."
)

# Create a view with aggregated data
dbExecute(dbcon, "DROP VIEW IF EXISTS VW_ABALONE_SUMMARY;")

dbExecute(dbcon,
  "CREATE VIEW VW_ABALONE_SUMMARY AS
   SELECT SEX,
          COUNT(*) as COUNT,
          AVG(LENGTH) as AVG_LENGTH,
          AVG(WHOLE) as AVG_WEIGHT
   FROM ABALONE
   GROUP BY SEX"
)

# Convert the view to a permanent table
dbTableFromView(
  view_name = "VW_ABALONE_SUMMARY",
  dbcon = dbcon,
  table_name = "ABALONE_STATS",
  drop_table = TRUE
)

# Check the result
dbListTables(dbcon)
dbGetQuery(dbcon, "SELECT * FROM ABALONE_STATS")

# Clean up
dbDisconnect(dbcon)
```

`dbTableFromXlsx` *Create a table in a SQLite database from an Excel worksheet*

Description

The `dbTableFromXlsx()` function creates a table in a SQLite database from a range of an Excel worksheet.

The `dbTableFromXlsx()` function reads the data from a range of an Excel worksheet. If table does not exist, it will create it.

Usage

```
dbTableFromXlsx(
  input_file,
  dbcon,
```

```
  table_name,
  sheet_name,
  first_row,
  cols_range,
  header = TRUE,
  id_quote_method = "DB_NAMES",
  col_names = NULL,
  col_types = NULL,
  col_import = NULL,
  drop_table = FALSE,
  auto_pk = FALSE,
  build_pk = FALSE,
  pk_fields = NULL,
  constant_values = NULL,
  ...
)
```

Arguments

input_file	character, the file name (including path) to be read.
dbcon	database connection, as created by the dbConnect function.
table_name	character, the name of the table.
sheet_name	character, the name of the worksheet containing the data table.
first_row	integer, the row number where the data table starts. If present, it is the row number of the header row, otherwise it is the row number of the first row of data.
cols_range	integer, a numeric vector specifying which columns in the worksheet to be read.
header	logical, if TRUE the first row contains the fields' names. If FALSE, the column names will be the column names of the Excel worksheet (i.e. letters).
id_quote_method	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the quote_method parameter of the format_column_names() function. Defautls to DB_NAMES.
col_names	character vector, names of the columuns in the input file. Used to override the field names derived from the input file (using the quote method selected by id_quote_method). Must be of the same length of the number of columns in the input file. If NULL the column names coming from the input file (after quoting) will be used. Defaults to NULL.
col_types	character vector of classes to be assumed for the columns of the input file. Must be of the same length of the number of columns in the input file. If not null, it will override the data types guessed from the input file. If NULL the data type inferred from the input files will be used. Defaults to NULL.
col_import	can be either: • a numeric vector (coherced to integers) with the columns' positions in the input file that will be imported in the SQLite table;

- a character vector with the columns' names to be imported. The names are those in the input file (after quoting with `id_quote_method`), if `col_names` is NULL, or those expressed in `col_names` vector. Defaults to NULL, i.e. all columns will be imported.

`drop_table` logical, if `TRUE` the target table will be dropped (if exists) and recreated before importing the data. if `FALSE`, data from input file will be appended to an existing table. Defaults to `FALSE`.

`auto_pk` logical, if `TRUE`, and `pk_fields` parameter is `NULL`, an additional column named `SEQ` will be added to the table and it will be defined to be `INTEGER PRIMARY KEY` (i.e. in effect an alias for `ROWID`). Defaults to `FALSE`.

`build_pk` logical, if `TRUE` creates a `UNIQUE INDEX` named `<table_name>_PK` defined by the combination of fields specified in the `pk_fields` parameter. It will be effective only if `pk_fields` is not null. Defaults to `FALSE`.

`pk_fields` character vector, the list of the fields' names that define the `UNIQUE INDEX`. Defults to `NULL`.

`constant_values` a one row data frame whose columns will be added to the table in the database. The additional table columns will be named as the data frame columns, and the corresponding values will be associeted to each record imported from the input file. It is useful to keep track of additional information (e.g., the input file name, additional context data not available in the data set, ...) when loading the content of multiple input files in the same table. Defults to `NULL`.

`...` additional arguments passed to `openxlsx2::wb_to_df()` function used to read input data.

Value

integer, the number of records in `table_name` after reading data from `input_file`.

Examples

```
# Create a temporary database and load Excel data
library(RSQLite.toolkit)

# Set up database connection
dbcon <- dbConnect(RSQLite::SQLite(), file.path(tempdir(), "example.sqlite"))

# Get path to example data
data_path <- system.file("extdata", package = "RSQLite.toolkit")

# Check if Excel file exists (may not be available in all installations)
xlsx_file <- file.path(data_path, "stock_portfolio.xlsx")

fschema <- file_schema_xlsx(xlsx_file, sheet_name="all period",
                            first_row=2, cols_range="A:S", header=TRUE,
                            id_quote_method="DB_NAMES", max_lines=10)

fschema[, c("col_names", "src_names")]
```

```
# Load Excel data from specific sheet and range
dbTableFromXlsx(
  input_file = xlsx_file,
  dbcon = dbcon,
  table_name = "PORTFOLIO_PERF",
  sheet_name = "all period",
  first_row = 2,
  cols_range = "A:S",
  drop_table = TRUE,
  col_import = c("ID", "Large_B_P", "Large_ROE", "Large_S_P",
                 "Annual_Return_7", "Excess_Return_8", "Systematic_Risk_9")
)

# Check the imported data
dbListFields(dbcon, "PORTFOLIO_PERF")
head(dbGetQuery(dbcon, "SELECT * FROM PORTFOLIO_PERF"))

# Clean up
dbDisconnect(dbcon)
```

`error_handler` *error_handler manage error messages for package*

Description

error_handler manage error messages for package

Usage

```
error_handler(err, fun, step)
```

Arguments

`err`	character, error message
`fun`	character, function name where error happened
`step`	integer, code identifying the step in the function where error happened. For dbTableFrom... functions steps are: • 101,121: read file schema (DSV, Xlsx) • 102: handle col_names and col_types • 103: create empty table • 104: read data • 105: write data • 106: indexing

Value

nothing

`file_schema_dsv` *Preview the table structure contained in a DSV file.*

Description

The `file_schema_dsv()` function returns a data frame with the schema of a DSV file reading only the first `max_lines` of a delimiter separated values (DSV) text file to infer column names and data types (it does not read the full dataset into memory). Then it converts them to the candidate data frame columns' names and data types.

Usage

```
file_schema_dsv(
  input_file,
  header = TRUE,
  sep = ",",
  dec = ".",
  grp = "",
  id_quote_method = "DB_NAMES",
  max_lines = 2000,
  null_columns = FALSE,
  force_num_cols = TRUE,
  ...
)
```

Arguments

`input_file`	character, file name (including path) to be read.
`header`	logical, if `TRUE` the first line contains the fields' names. If `FALSE`, the column names will be formed sing a "V" followed by the column number (as specified in `utils::read.table()`).
`sep`	character, field delimiter (e.g., "," for CSV, "\t" for TSV) in the input file. Defaults to ",".
`dec`	character, decimal separator (e.g., "." or "," depending on locale) in the input file. Defaults to ".".
`grp`	character, character used for digit grouping. It defaults to `""` (i.e. no grouping).
`id_quote_method`	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the `quote_method` parameter of the `format_column_names()` function. Defautls to `DB_NAMES`.
`max_lines`	integer, number of lines (excluding the header) to be read to infer columns' data types. Defaults to 2000.
`null_columns`	logical, if `TRUE` the col_type of columuns consisting only of NAs or zero-length strings will be marked as `"NULL"`, otherwise they will be marked as `character`. Defaults to `FALSE`

force_num_cols logical, if TRUE the returned schema will have all rows with n_cols columns (i.e. the guessed number of columns determined inspecting the first max_lines lines of the input file), even if there are rows in the input file with fewer or greater columns than n_cols. If FALSE and any of the tested lines has a number of columns not equal to n_cols, the function will return a list without the schema element. It defaults to TRUE.

... Additional arguments for quoting and data interpretation as described in the base::scan() function. The parameters used by file_schema_dsv are:

- quote, character, the set of quoting characters. Defaults to "" (i.e., no quoting).
- comment.char, character, the comment character. Defaults to "" (i.e., no comments).
- skip, integer, the number of lines to skip before reading data. Defaults to 0.
- fileEncoding, character, the name of the encoding of the input file. Defaults to "".
- na.strings, character vector, the strings to be interpreted as NAs. Defaults to c("NA").

Value

a list with the following named elements:

- schema, a data frame with these columns:
  - col_names: columns' names, after applying the selected quote method;
  - col_names_unquoted: columns' names, unquoted; if id_quote_method is set to DB_NAMES they will be the same as col_names; for other quote methods they will be the unquoted versions of col_names,that is generally the same as src_names unless src_names contain the quoting characters;
  - col_types: columns' R data types;
  - sql_types: columns' SQLite data types;
  - src_names: columns' names as they appear in the input file.
  - src_types: defaults to text for all columns.
  - src_is_quoted: logical vector indicating if each column has at least one value enclosed in quotes.
- col_counts, a data frame with these columns:
  - num_col: number of columns,
  - Freq: number of rows (within max_lines) that have the number of colums shown in num_col.
- n_cols, integer, the number of columns selected for the file.
- num_col, a vector of integers of length max_lines with the number of detected columns in each row tested.
- col_fill, logical, it is set to TRUE if there are lines with less columns than n_cols.
- col_flush, logical, it is set to TRUE if there are lines with more columns than n_cols.

Examples

```
# Inspect CSV file schema without loading full dataset
data_path <- system.file("extdata", package = "RSQLite.toolkit")

# Get schema information for abalone CSV
schema_info <- file_schema_dsv(
  input_file = file.path(data_path, "abalone.csv"),
  header = TRUE,
  sep = ",",
  dec = ".",
  max_lines = 50
)

# Display schema information
print(schema_info$schema[, c("col_names", "col_types", "sql_types")])

# Check column consistency
print(schema_info$col_counts)
print(paste("Guessed columns:", schema_info$n_cols))

# Example with different parameters
schema_custom <- file_schema_dsv(
  input_file = file.path(data_path, "abalone.csv"),
  header = TRUE,
  sep = ",",
  dec = ".",
  max_lines = 50,
  id_quote_method = "SQL_SERVER"
)

print(schema_custom$schema[, c("col_names", "col_types", "src_names")])
```

---

file_schema_feather *Preview the table structure contained in a Feather file.*

Description

The `file_schema_feather()` function returns a data frame with the schema of a Feather file. This function is used to preview the table structure contained in a Feather file, by reading only the metadata of the file. It inspects the input file metadata to read the field identifiers' names and data types, then converts them to the candidate data frame columns' names and data types. The dataset contained in the input file is not read in to memory, only meta-data are accessed.

Usage

```
file_schema_feather(input_file, id_quote_method = "DB_NAMES")
```

Arguments

input_file	File name (including path) to be read
id_quote_method	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the quote_method parameter of the format_column_names() function. Defaults to DB_NAMES.

Value

a data frame with these columns:

- col_names: columns' names, after applying the selected quote method;
- col_names_unquoted: columns' names, unquoted; if id_quote_method is set to DB_NAMES they will be the same as col_names; for other quote methods they will be the unquoted versions of col_names, that is generally the same as src_names unless src_names contain the quoting characters;
- col_types: columns' R data types;
- sql_types: columns' SQLite data types;
- src_names: columns' names as they appear in the input file;
- src_types: the Arrow's data type of each column.

References

The implementation is based on this question on Stackoverflow. # nolint: line_length_linter.

Examples

```
# Inspect Feather file schema
data_path <- system.file("extdata", package = "RSQLite.toolkit")

# Get schema information for penguins Feather file
schema_info <- file_schema_feather(
  input_file = file.path(data_path, "penguins.feather")
)

# Display schema information
print(schema_info[, c("col_names", "col_types", "sql_types", "src_names")])

# Check specific columns
print(paste("Number of columns:", nrow(schema_info)))
print(paste("Column names:", paste(schema_info$col_names, collapse = ", ")))
```

file_schema_xlsx *Preview the structure of a range of an Excel worksheet.*

Description

The file_schema_xlsx() function returns a data frame with the schema of an Excel data table. It will read only a range of the specified worksheet to infer column names and data types. Then it converts them to the candidate data frame columns' names and data types.

Usage

```
file_schema_xlsx(
  input_file,
  sheet_name,
  first_row,
  cols_range,
  header = TRUE,
  id_quote_method = "DB_NAMES",
  max_lines = 100,
  null_columns = FALSE,
  ...
)
```

Arguments

input_file	character, file name (including path) to be read.
sheet_name	character, the name of the worksheet containing the data table.
first_row	integer, the row number where the data table starts. If present, it is the row number of the header row, otherwise it is the row number of the first row of data.
cols_range	integer, a numeric vector specifying which columns in the worksheet to be read.
header	logical, if TRUE the first row contains the fields' names. If FALSE, the column names will be the column names of the Excel worksheet (i.e. letters).
id_quote_method	character, used to specify how to build the SQLite columns' names using the fields' identifiers read from the input file. For details see the description of the quote_method parameter of the format_column_names() function. Defaults to DB_NAMES.
max_lines	integer, number of lines (excluding the header) to be read to infer columns' data types. Defaults to 100.
null_columns	logical, if TRUE the col_type of columuns consisting only of NAs or zero-length strings will be marked as NA, otherwise they will be marked as character. Defaults to FALSE
...	Additional parameters passed to openxlsx2::wb_to_df() function.

Value

a data frame with these columns:

- `col_names`: columns' names, after applying the selected quote method;
- `col_names_unquoted`: columns' names, unquoted; if `id_quote_method` is set to `DB_NAMES` they will be the same as `col_names`; for other quote methods they will be the unquoted versions of `col_names`,that is generally the same as `src_names` unless `src_names` contain the quoting characters;
- `col_types`: columns' R data types;
- `sql_types`: columns' SQLite data types;
- `src_names`: columns' names as they appear in the input file;
- `src_types`: data type attribute of each column, as determined by the `openxlsx2::wb_to_df()` function.

Examples

```
# Inspect xlsx file schema
data_path <- system.file("extdata", package = "RSQLite.toolkit")

# Get schema information for Excel file
schema_info <- file_schema_xlsx(
  input_file = file.path(data_path, "stock_portfolio.xlsx"),
  sheet_name = "all period",
  first_row = 2,
  cols_range = "A:S",
  header = TRUE,
  id_quote_method = "DB_NAMES",
  max_lines = 10
)

# Display schema information
head(schema_info[, c("col_names", "src_names")])

# Check specific columns
print(paste("Number of columns:", nrow(schema_info)))
```

`format_column_names` *Format column names for SQLite*

Description

The `format_column_names()` function formats a vector of strings to be used as columns' names for a table in a SQLite database.

Usage

```
format_column_names(
  x,
  quote_method = "DB_NAMES",
  unique_names = TRUE,
  encoding = ""
)
```

Arguments

x	character vector with the identifiers' names to be quoted.
quote_method	character, used to specify how to build the SQLite columns' names from the identifiers passed through the x parameter. Supported values for quote_method: • DB_NAMES tries to build a valid SQLite column name: a. substituting all characters, that are not letters or digits or the _ character, with the _ character; b. prefixing N_ to all strings starting with a digit; c. prefixing F_ to all strings equal to any SQL92 keyword. • SINGLE_QUOTES encloses each string in single quotes. • SQL_SERVER encloses each string in square brackets. • MYSQL encloses each string in back ticks. Defaults to DB_NAMES.
unique_names	logical, checks for any duplicate name after applying the selected quote methods. If duplicates exist, they will be made unique by adding a postfix _[n], where n is a progressive integer. Defaults to TRUE.
encoding	character, encoding to be assumed for input strings. It is used to re-encode the input in order to process it to build column identifiers. Defaults to '""' (for the encoding of the current locale).

Value

A data frame containing the columns' identifiers in two formats:

- quoted: the quoted names, as per the selected quote_method;
- unquoted: the cleaned names, without any quoting.

Examples

```
# Example with DB_NAMES method
col_names <- c("column 1", "column-2", "3rd_column", "SELECT")

formatted_names <- format_column_names(col_names, quote_method = "DB_NAMES")
print(formatted_names)

# Example with SINGLE_QUOTES method
formatted_names_sq <- format_column_names(col_names, quote_method = "SINGLE_QUOTES")
print(formatted_names_sq)

# Example with SQL_SERVER method
formatted_names_sqlsrv <- format_column_names(col_names, quote_method = "SQL_SERVER")
```

```
print(formatted_names_sqlsrv)
```

R2SQL_types *From R class names to SQLite data types*

Description

The R2SQL_types() function returns a character vector with the names of SQLite data types corresponding to the R classes passed through the x parameter.

If any class is not recognized, it will be replaced with TEXT data type.

Usage

```
R2SQL_types(x)
```

Arguments

x	character, a vector containing the strings with the R class names.

Value

a character vector with the names of SQLite data types.

Examples

```
# Convert R data types to SQLite types
r_types <- c("character", "integer", "numeric", "logical", "Date")
sql_types <- R2SQL_types(r_types)

# Display the mapping
data.frame(
  R_type = r_types,
  SQLite_type = sql_types,
  row.names = NULL
)

# Handle unknown types (converted to TEXT)
mixed_types <- c("character", "unknown_type", "integer")
R2SQL_types(mixed_types)
```

Index