Package 'pgTools'

March 24, 2023

Type Package

Title Functions for Generating PostgreSQL Statements/Scripts

Version 1.0.2

Author Timothy Conwell

Maintainer Timothy Conwell <timconwell@gmail.com>

Description Create PostgreSQL statements/scripts from R, optionally executing the SQL statements.
Common SQL operations are included, although not every configurable option is available at this time.
SQL output is intended to be compliant with PostgreSQL syntax specifications. PostgreSQL documentation is available here
<https://www.postgresql.org/docs/current/index.html>.

License GPL (>= 3)

Encoding UTF-8

Depends R (>= 3.5.0), data.table, toolbox

Imports DBI, odbc, parallel, stringi

RoxygenNote 7.2.0

URL https://github.com/tconwell/pgTools

BugReports https://github.com/tconwell/pgTools/issues

NeedsCompilation no

Repository CRAN

Date/Publication 2023-03-24 11:40:02 UTC

R topics documented:

`alterDATABASE` *Generate a PostgreSQL ALTER DATABASE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL ALTER DATABASE statement, optionally execute the statement if con is not NULL.

Usage

```
alterDATABASE(
  name,
  allow_connections = NULL,
  connection_limit = NULL,
  is_template = NULL,
  rename_to = NULL,
  owner_to = NULL,
  set_tablespace = NULL,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL ALTER DATABASE.
`allow_connections`	A string, the "allowconn" parameter for PostgreSQL ALTER DATABASE.
`connection_limit`	A string, the "connlimit" parameter for PostgreSQL ALTER DATABASE.
`is_template`	A string, the "istemplate" parameter for PostgreSQL ALTER DATABASE.
`rename_to`	A string, the "new_name" parameter for PostgreSQL ALTER DATABASE.
`owner_to`	A string, the "new_owner" parameter for PostgreSQL ALTER DATABASE.
`set_tablespace`	A string, the "new_tablespace" parameter for PostgreSQL ALTER DATABASE.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL ALTER DATABASE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
alterDATABASE("dbTest01", rename_to = "dbProd01")
```

`alterSCHEMA` *Generate a PostgreSQL ALTER SCHEMA statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL ALTER SCHEMA statement, optionally execute the statement if con is not NULL.

Usage

```
alterSCHEMA(name, rename_to = NULL, owner_to = NULL, con = NULL)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL ALTER SCHEMA.
`rename_to`	A string, the "new_name" parameter for PostgreSQL ALTER SCHEMA.
`owner_to`	A string, the "new_owner" parameter for PostgreSQL ALTER SCHEMA.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE SCHEMA statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
alterSCHEMA("dev", rename_to = "prod")
```

`alterTABLE` *Generate a PostgreSQL ALTER TABLE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL ALTER TABLE statement, optionally execute the statement if con is not NULL.

Usage

```
alterTABLE(
  name,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  action,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL ALTER TABLE statement.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL ALTER TABLE statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL ALTER TABLE statement.
`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL ALTER TABLE statement.
`action`	A string or vector of strings, the "action" parameter for PostgreSQL ALTER TABLE statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL ALTER TABLE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
alterTABLE("sample", action = c("ADD COLUMN IF NOT EXISTS col4 BOOLEAN"))
```

`arrayStrToVec` *Write a PostgreSQL array as a string from a vector.*

Description

Write a PostgreSQL array as a string from a vector.

Usage

```
arrayStrToVec(x)
```

Arguments

`x`	A vector.

Value

A string.

Examples

```
arrayStrToVec(vecToArrayStr(c("a", "b")))
```

`callFUNCTION` *Generate a PostgreSQL statement to execute a function, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL statement to execute a function, optionally execute the statement if con is not NULL.

Usage

```
callFUNCTION(
  x = list(),
  schema = NULL,
  func,
  quote_text = TRUE,
  cast = TRUE,
  types,
  con = NULL
)
```

Arguments

`x`	A named list, names must match the parameter names of the SQL function, values are the values to set the parameters to when executing the SQL function.
`schema`	A string, the schema name of the SQL function.
`func`	A string, the name of the SQL function.
`quote_text`	TRUE/FALSE, if TRUE, calls quoteText() to add single quotes around character strings.
`cast`	TRUE/FALSE, if TRUE, will add SQL to cast the parameters to the specified type.
`types`	A vector of character strings specifying the SQL data types of the function parameters, the position of the type should match the position of the parameter for that type in x.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL statement to execute a function; or the results retrieved by DBI::dbGetQuery after executing the statement.

Examples

```
callFUNCTION(
x = list(a = 1, b = 2),
schema = NULL,
func = "sample_add",
quote_text = TRUE,
cast = FALSE,
types = c("INT", "INT")
)
```

`callPROCEDURE` *Generate a PostgreSQL statement to execute a procedure, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL statement to execute a procedure, optionally execute the statement if con is not NULL.

Usage

```
callPROCEDURE(
  x = list(),
  schema = NULL,
  proc,
  quote_text = TRUE,
  cast = TRUE,
  types,
  con = NULL
)
```

Arguments

`x`	A named list, names must match the parameter names of the SQL procedure, values are the values to set the parameters to when executing the SQL procedure.
`schema`	A string, the schema name of the SQL procedure.
`proc`	A string, the name of the SQL procedure.
`quote_text`	TRUE/FALSE, if TRUE, calls quoteText() to add single quotes around character strings.
`cast`	TRUE/FALSE, if TRUE, will add SQL to cast the parameters to the specified type.
`types`	A vector of character strings specifying the SQL data types of the procedure parameters, the position of the type should match the position of the parameter for that type in x.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL statement to execute a procedure; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
callPROCEDURE(
x = list(a = 1, b = 2),
schema = NULL,
proc = "sample_add",
quote_text = TRUE,
cast = FALSE,
types = c("INT", "INT")
)
```

---

connect *Connect to a local database with local credentials using DBI/odbc.*

Description

Connect to a local database with local credentials using DBI/odbc.

Usage

```
connect(db)
```

Arguments

db	A string, a database you can connect to locally.

Value

A database connection.

Examples

```
connect(NULL)
```

COPY *Generate a PostgreSQL COPY command, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL COPY command, optionally execute the statement if con is not NULL.

Usage

```
COPY(
  schema = NULL,
  table,
  columns = NULL,
  file,
  type = "FROM",
  delimiter = ",",
  format = "csv",
  query = NULL,
  header = TRUE,
  con = NULL
)
```

Arguments

`schema`	A string, the schema of the table to copy from/to.
`table`	A string, the table to copy from/to.
`columns`	A vector, columns to read/write.
`file`	A string, the file path and name to read/write.
`type`	A string, "FROM" or "TO".
`delimiter`	A string, the delimiter.
`format`	A string, "CSV", "TEXT", or "BINARY".
`query`	A string, the query used to select data for output.
`header`	TRUE/FALSE, if TRUE, adds HEADER to statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL COPY command; or the results retrieved by DBI::dbExecute after executing the statement.

Examples

```
COPY(
table = "table1",
file = "/home/test/test.csv"
)
```

createDATABASE — *Generate a PostgreSQL CREATE DATABASE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE DATABASE statement, optionally execute the statement if con is not NULL.

Usage

```
createDATABASE(
  name,
  owner = NULL,
  template = NULL,
  encoding = NULL,
  locale = NULL,
  lc_collate = NULL,
  lc_ctype = NULL,
  tablespace = NULL,
  allow_connections = NULL,
  connection_limit = NULL,
  is_template = NULL,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL CREATE DATABASE.
`owner`	A string, the "user_name" parameter for PostgreSQL CREATE DATABASE.
`template`	A string, the "template" parameter for PostgreSQL CREATE DATABASE.
`encoding`	A string, the "encoding" parameter for PostgreSQL CREATE DATABASE.
`locale`	A string, the "locale" parameter for PostgreSQL CREATE DATABASE
`lc_collate`	A string, the "lc_collate" parameter for PostgreSQL CREATE DATABASE.
`lc_ctype`	A string, the "lc_ctype" parameter for PostgreSQL CREATE DATABASE.
`tablespace`	A string, the "tablespace_name" parameter for PostgreSQL CREATE DATABASE.
`allow_connections`	A string, the "allowconn" parameter for PostgreSQL CREATE DATABASE.
`connection_limit`	A string, the "connlimit" parameter for PostgreSQL CREATE DATABASE.
`is_template`	A string, the "istemplate" parameter for PostgreSQL CREATE DATABASE.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE DATABASE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createDATABASE("dbTest01")
```

`createEXTENSION` *Generate a PostgreSQL CREATE EXTENSION statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE EXTENSION statement, optionally execute the statement if con is not NULL.

Usage

```
createEXTENSION(
  name,
  if_not_exists = FALSE,
  schema = NULL,
  version = NULL,
  cascade = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "extension_name" parameter for PostgreSQL CREATE EXTENSION.
`if_not_exists`	TRUE/FALSE, if TRUE, adds "IF NOT EXISTS" to PostgreSQL CREATE EXTENSION statement.
`schema`	A string, the "schema_name" parameter for PostgreSQL CREATE EXTENSION.
`version`	A string, the "version" parameter for PostgreSQL CREATE EXTENSION.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL CREATE EXTENSION statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE EXTENSION statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createEXTENSION("pgcrypto")
```

---

createFUNCTION *Generate a PostgreSQL CREATE FUNCTION statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE FUNCTION statement, optionally execute the statement if con is not NULL.

Usage

```
createFUNCTION(
  name,
  args = NULL,
  or_replace = FALSE,
  returns = NULL,
  returns_table = NULL,
  language = "SQL",
  definition,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL CREATE FUNCTION.
`args`	A named list, names are the argument names, values are strings with the argument data types.
`or_replace`	TRUE/FALSE, if TRUE, adds "OR REPLACE" to PostgreSQL CREATE FUNCTION statement.
`returns`	A string, the "returns" parameter for PostgreSQL CREATE FUNCTION.
`returns_table`	A named list, names are the return table column names, values are strings with the return table data types.
`language`	A string, the "language" parameter for PostgreSQL CREATE FUNCTION.
`definition`	A string, the "definition" parameter for PostgreSQL CREATE FUNCTION.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE PROCEDURE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createFUNCTION(
name = "sample_add",
args = list(a = "INTEGER", b = "INTEGER"),
returns = "INT",
language = "plpgsql",
definition = "BEGIN RETURN sample_add.a + sample_add.b; END;"
)
```

`createPROCEDURE` *Generate a PostgreSQL CREATE PROCEDURE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE PROCEDURE statement, optionally execute the statement if con is not NULL.

Usage

```
createPROCEDURE(
  name,
  args = NULL,
  or_replace = FALSE,
  language = "SQL",
  definition,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL CREATE PROCEDURE.
`args`	A named list, names are the argument names, values are strings with the argument data types.
`or_replace`	TRUE/FALSE, if TRUE, adds "OR REPLACE" to PostgreSQL CREATE PROCEDURE statement.
`language`	A string, the "language" parameter for PostgreSQL CREATE PROCEDURE.
`definition`	A string, the "definition" parameter for PostgreSQL CREATE PROCEDURE.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE PROCEDURE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createPROCEDURE(
name = "sample",
args = list(a = "INTEGER", b = "TEXT"),
definition = "INSERT INTO tbl(col1, col2) VALUES (a, b);"
)
```

`createSCHEMA` *Generate a PostgreSQL CREATE SCHEMA statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE SCHEMA statement, optionally execute the statement if con is not NULL.

Usage

```
createSCHEMA(name, authorization = NULL, if_not_exists = FALSE, con = NULL)
```

Arguments

`name`	A string, the "schema_name" parameter for PostgreSQL CREATE SCHEMA.
`authorization`	A string, the "role_specification" parameter for PostgreSQL CREATE SCHEMA.
`if_not_exists`	TRUE/FALSE, if TRUE, adds "IF NOT EXISTS" to PostgreSQL CREATE SCHEMA statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE SCHEMA statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createSCHEMA("dev")
```

`createTABLE`	*Generate a PostgreSQL CREATE TABLE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE TABLE statement, optionally execute the statement if con is not NULL.

Usage

```
createTABLE(
  name,
  columns,
  select = NULL,
  constraints = NULL,
  temporary = FALSE,
  if_not_exists = FALSE,
  unlogged = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "table_name" parameter for PostgreSQL CREATE TABLE.
`columns`	A named list, names are the SQL column names, values are strings with the SQL column data types, constraints, etc.
`select`	A string, the select statement to use to create the table.
`constraints`	A named list, names are the SQL constraint names, values are strings with the SQL constraint.
`temporary`	TRUE/FALSE, if TRUE, adds "TEMPORARY" to PostgreSQL CREATE TABLE statement.
`if_not_exists`	TRUE/FALSE, if TRUE, adds "IF NOT EXISTS" to PostgreSQL CREATE TABLE statement.
`unlogged`	TRUE/FALSE, if TRUE, adds "UNLOGGED" to PostgreSQL CREATE TABLE statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE TABLE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createTABLE(
name = "sample",
columns = list(col1 = "SERIAL NOT NULL", col2 = "INTEGER", col3 = "TEXT"),
constraints = list(sample_constraint = "UNIQUE(col3)")
)
```

---

createTRIGGER *Generate a PostgreSQL CREATE TRIGGER statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL CREATE TRIGGER statement, optionally execute the statement if con is not NULL.

Usage

```
createTRIGGER(name, when, event, on, for_each_row = FALSE, func, con = NULL)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL CREATE TRIGGER.
`when`	A string, the "when" parameter (BEFORE, AFTER, INSTEAD OF) for PostgreSQL CREATE TRIGGER.
`event`	A string, the "event" parameter (INSERT/UPDATE/DELETE/TRUNCATE) for PostgreSQL CREATE TRIGGER.
`on`	A string, the "table_name" parameter for PostgreSQL CREATE TRIGGER.
`for_each_row`	TRUE/FALSE, if TRUE, adds "FOR EACH ROW" to PostgreSQL CREATE TRIGGER statement.
`func`	A string, the function call to be executed by the PostgreSQL CREATE TRIGGER.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL CREATE TRIGGER statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
createTRIGGER(
name = "sample_trigger",
when = "AFTER",
event = "INSERT",
on = "sample_table",
for_each_row = TRUE,
func = "function_sample()"
)
```

`create_sql_script` *Create a SQL script, optionally execute the statement if con is not NULL.*

Description

Create a SQL script, optionally execute the statement if con is not NULL.

Usage

```
create_sql_script(..., path = NULL, con = NULL)
```

Arguments

`...`	A string, SQL command to be combined into one document or statement.
`path`	A string, the file path (include the file name) to save the script.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, SQL commands combined into one document or statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
create_sql_script(
createSCHEMA("dev"),
sql_80_char_comment(),
createTABLE(name = "sample",
columns = list(col1 = "SERIAL NOT NULL", col2 = "INTEGER", col3 = "TEXT"),
constraints = list(sample_constraint = "UNIQUE(col3)")
))
```

DELETE — *Generate a PostgreSQL DELETE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DELETE statement, optionally execute the statement if con is not NULL.

Usage

```
DELETE(schema = NULL, table, where = NULL, con = NULL)
```

Arguments

`schema`	A string, the schema name of the SQL table to DELETE from.
`table`	A string, the table name of the SQL table to DELETE from.
`where`	A named list, names are the columns for comparison, values are lists with a comparison operator and a value the comparison operator will check against. ex: list(col1 = list(comparison = "=", value = quoteText("b")))
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DELETE statement; or the results retrieved by DBI::dbGetQuery after executing the statement.

Examples

```
DELETE(
schema = "test",
table = "table1",
where = list(
  col1 = list(comparison = "=", value = quoteText("b")),
  col2 = list(comparison = "IS", value = "NULL")
)
)
```

`dropDATABASE` *Generate a PostgreSQL DROP DATABASE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP DATABASE statement, optionally execute the statement if con is not NULL.

Usage

```
dropDATABASE(name, if_exists = FALSE, force = FALSE, con = NULL)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP DATABASE.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP DATABASE statement.
`force`	TRUE/FALSE, if TRUE, adds "FORCE" to PostgreSQL DROP DATABASE statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP DATABASE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropDATABASE("dbTest01")
```

`dropEXTENSION` *Generate a PostgreSQL DROP EXTENSION statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP EXTENSION statement, optionally execute the statement if con is not NULL.

Usage

```
dropEXTENSION(
  name,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP EXTENSION.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP EXTENSION statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL DROP EXTENSION statement.
`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL DROP EXTENSION statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP EXTENSION statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropEXTENSION("pgcrypto")
```

`dropFUNCTION` *Generate a PostgreSQL DROP FUNCTION statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP FUNCTION statement, optionally execute the statement if con is not NULL.

Usage

```
dropFUNCTION(
  name,
  args = NULL,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP FUNCTION.
`args`	A named list, names are the argument names, values are strings with the argument data types.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP FUNCTION statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL DROP FUNCTION statement.
`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL DROP FUNCTION statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP FUNCTION statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropFUNCTION(
name = "sample",
args = list(a = "INTEGER", b = "TEXT")
)
```

`dropPROCEDURE` *Generate a PostgreSQL DROP PROCEDURE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP PROCEDURE statement, optionally execute the statement if con is not NULL.

Usage

```
dropPROCEDURE(
  name,
  args = NULL,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP PROCEDURE.
`args`	A named list, names are the argument names, values are strings with the argument data types.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP PROCEDURE statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL DROP PROCEDURE statement.
`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL DROP PROCEDURE statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP PROCEDURE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropPROCEDURE(
name = "sample",
args = list(a = "INTEGER", b = "TEXT")
)
```

`dropSCHEMA` *Generate a PostgreSQL DROP SCHEMA statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP SCHEMA statement, optionally execute the statement if con is not NULL.

Usage

```
dropSCHEMA(
  name,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP SCHEMA.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP SCHEMA statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL DROP SCHEMA statement.
`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL DROP SCHEMA statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP SCHEMA statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropSCHEMA("dev")
```

`dropTABLE` *Generate a PostgreSQL DROP TABLE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP TABLE statement, optionally execute the statement if con is not NULL.

Usage

```
dropTABLE(
  name,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP TABLE.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP TABLE statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL DROP TABLE statement.

`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL DROP TABLE statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP TABLE statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropTABLE("sample")
```

dropTRIGGER — *Generate a PostgreSQL DROP TRIGGER statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL DROP TRIGGER statement, optionally execute the statement if con is not NULL.

Usage

```
dropTRIGGER(
  name,
  on,
  if_exists = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`name`	A string, the "name" parameter for PostgreSQL DROP TRIGGER.
`on`	A string, the "table_name" parameter for PostgreSQL DROP TRIGGER.
`if_exists`	TRUE/FALSE, if TRUE, adds "IF EXISTS" to PostgreSQL DROP TRIGGER statement.
`cascade`	TRUE/FALSE, if TRUE, adds "CASCADE" to PostgreSQL DROP TRIGGER statement.
`restrict`	TRUE/FALSE, if TRUE, adds "RESTRICT" to PostgreSQL DROP TRIGGER statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DROP TRIGGER statement; or the results retrieved by DBI::dbSendQuery after executing the statement.

Examples

```
dropTRIGGER(
name = "sample_trigger",
on = "sample_table"
)
```

`INSERT` *Generate a PostgreSQL INSERT statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL INSERT statement, optionally execute the statement if con is not NULL.

Usage

```
INSERT(
  x = NULL,
  schema = NULL,
  table,
  types = NULL,
  returning = NULL,
  quote_text = TRUE,
  cast = TRUE,
  prepare = TRUE,
  batch_size = 50000,
  double_quote_names = FALSE,
  select = NULL,
  select_cols = NULL,
  con = NULL,
  n_cores = 1,
  table_is_temporary = FALSE,
  retain_insert_order = FALSE,
  connect_db_name = NULL
)
```

Arguments

`x`	A list, data.frame or data.table, names must match the column names of the destination SQL table.
`schema`	A string, the schema name of the destination SQL table.
`table`	A string, the table name of the destination SQL table.

`types`	A vector of character strings specifying the SQL data types of the destination columns, the position of the type should match the position of the column for that type in x. Required if prepare or cast is TRUE.
`returning`	A vector of character strings specifying the SQL column names to be returned by the INSERT statement.
`quote_text`	TRUE/FALSE, if TRUE, calls quoteText() to add single quotes around character strings.
`cast`	TRUE/FALSE, if TRUE, will add SQL to cast the data to be inserted to the specified type.
`prepare`	TRUE/FALSE, if TRUE, creates a PostgreSQL prepared statement for inserting the data.
`batch_size`	Integer, the maximum number of records to submit in one statement.
`double_quote_names`	TRUE/FALSE, if TRUE, adds double quotes to column names.
`select`	A string, a SELECT statement.
`select_cols`	A character vector of the columns to insert the results of the select statement. Only used if select is not NULL.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.
`n_cores`	A integer, the number of cores to use for parallel forking (passed to parallel::mclapply as mc.cores).
`table_is_temporary`	TRUE/FALSE, if TRUE, prevents parallel processing.
`retain_insert_order`	TRUE/FALSE, if TRUE, prevents parallel processing.
`connect_db_name`	The name of the database to pass to connect() when inserting in parallel.

Value

A string, PostgreSQL INSERT statement; or a string, PostgreSQL prepared statement; or the results retrieved by DBI::dbGetQuery after executing the statement.

Examples

```
INSERT(
x = list(col1 = c("a", "b", "c"), col2 = c(1, 2, 3)),
schema = "test",
table = "table1",
prepare = TRUE,
types = c("TEXT", "INTEGER"),
returning = NULL,
quote_text = TRUE,
cast = TRUE
)
```

`insert_batch_chunker` *Helper function for INSERT*

Description

Helper function for INSERT

Usage

```
insert_batch_chunker(x, n_batches, batch_size)
```

Arguments

`x`	A vector of data to insert.
`n_batches`	Integer, the number of batches needed to insert the data.
`batch_size`	Integer, the size of each batch.

Value

A list.

Examples

```
insert_batch_chunker(c(1, 2, 3), 1, 100)
```

`insert_table_chunker` *Helper function for INSERT*

Description

Helper function for INSERT

Usage

```
insert_table_chunker(x, n_batches, batch_size)
```

Arguments

`x`	A data table
`n_batches`	Integer, the number of batches needed to insert the data.
`batch_size`	Integer, the size of each batch.

Value

A list.

Examples

```
insert_table_chunker(as.data.table(list(c1 = c(1, 2, 3))), 1, 100)
```

`pg_addColumn` *Helper command to add a column via ALTER TABLE.*

Description

Helper command to add a column via ALTER TABLE.

Usage

```
pg_addColumn(
  column_name,
  data_type,
  default = NULL,
  constraint = NULL,
  if_not_exists = FALSE
)
```

Arguments

`column_name`	A string, the name of the column to add.
`data_type`	A string, the data type of the column to add.
`default`	A string, a default value for the column to add.
`constraint`	A string, a constraint for the column to add.
`if_not_exists`	Boolean, if TRUE, adds IF NOT EXISTS to the ADD COLUMN statement.

Value

A string, PostgreSQL helper statement to add a column using ALTER TABLE.

Examples

```
pg_addColumn(
column_name = "newCol",
data_type = "text"
)
```

`pg_alterColumnType` *Helper command to alter a column's data type via ALTER TABLE.*

Description

Helper command to alter a column's data type via ALTER TABLE.

Usage

```
pg_alterColumnType(column_name, data_type, using = NULL)
```

Arguments

`column_name`	A string, the name of the column to add.
`data_type`	A string, the data type of the column to add.
`using`	A string, a command to cast the column into the appropriate type.

Value

A string, PostgreSQL helper statement to alter a column type using ALTER TABLE.

Examples

```
pg_alterColumnType(
column_name = "newCol",
data_type = "text"
)
```

`pg_data_types` *PostgreSQL data types*

Description

A vector of PostgreSQL data types

Usage

```
pg_data_types
```

Format

A vector

pg_dropColumn *Helper command to drop a column via ALTER TABLE.*

Description

Helper command to drop a column via ALTER TABLE.

Usage

```
pg_dropColumn(
  column_name,
  cascade = FALSE,
  restrict = FALSE,
  if_exists = FALSE
)
```

Arguments

`column_name`	A string, the name of the column to drop.
`cascade`	Boolean, if TRUE, adds CASCADE to the DROP COLUMN statement.
`restrict`	Boolean, if TRUE, adds RESTRICT to the DROP COLUMN statement.
`if_exists`	Boolean, if TRUE, adds IF EXISTS to the DROP COLUMN statement.

Value

A string, PostgreSQL helper statement to drop a column using ALTER TABLE.

Examples

```
pg_dropColumn(
column_name = "newCol"
)
```

pg_renameColumn *Helper command to rename a column via ALTER TABLE.*

Description

Helper command to rename a column via ALTER TABLE.

Usage

```
pg_renameColumn(column_name, new_column_name)
```

Arguments

`column_name`	A string, the name of the column to change.
`new_column_name`	A string, the new name for the column.

Value

A string, PostgreSQL helper statement to rename a column using ALTER TABLE.

Examples

```
pg_renameColumn(
column_name = "newCol",
new_column_name = "col1"
)
```

`pg_renameTable` *Helper command to rename a table via ALTER TABLE.*

Description

Helper command to rename a table via ALTER TABLE.

Usage

```
pg_renameTable(new_table_name)
```

Arguments

`new_table_name`	A string, the new name for the table.

Value

A string, PostgreSQL helper statement to rename a table using ALTER TABLE.

Examples

```
pg_renameTable(
new_table_name = "table1"
)
```

querySELECT *Generate a PostgreSQL select statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL select statement, optionally execute the statement if con is not NULL.

Usage

```
querySELECT(
  select,
  from = list(),
  where = NULL,
  group_by = NULL,
  having = NULL,
  order_by = NULL,
  con = NULL
)
```

Arguments

`select`	A vector of columns/items to select.
`from`	A string, the table(s) to select from.
`where`	A string, text to include in the where clause.
`group_by`	A vector of columns/items to group by.
`having`	A vector of conditions to be met by aggregations.
`order_by`	A vector of columns/items to order by.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL select statement; or the results retrieved by DBI::dbGetQuery after executing the statement.

Examples

```
querySELECT(
select = c("col1", "col2", "col1 + col2 AS col3"),
from = "schema1.table1"
)
```

quoteText2 — *Add single quotes to strings using stringi::stri_join, useful for converting R strings into SQL formatted strings.*

Description

Add single quotes to strings using stringi::stri_join, useful for converting R strings into SQL formatted strings.

Usage

```
quoteText2(x, char_only = TRUE, excluded_chars = c("NULL"))
```

Arguments

`x`	A string.
`char_only`	TRUE/FALSE, if TRUE, adds quotes only if is.character(x) is TRUE.
`excluded_chars`	A character vector, will not add quotes if a value is in excluded_chars.

Value

A string, with single quotes added to match PostgreSQL string formatting.

Examples

```
quoteText2("Sample quotes.")
```

sqlNameWalk — *Convert a column name into a PostgreSQL compatible name.*

Description

Convert a column name into a PostgreSQL compatible name.

Usage

```
sqlNameWalk(x, double_quote = FALSE)
```

Arguments

`x`	A string, a column name.
`double_quote`	TRUE/FALSE, if true, will add double quotes rather than replace non-compatible characters with underscores.

Value

A string, a PostgreSQL compatible column name.

Examples

```
sqlNameWalk("column 100 - sample b")
```

---

sqlTypeWalk *Get the PostgreSQL data type for a given R data type.*

Description

Get the PostgreSQL data type for a given R data type.

Usage

```
sqlTypeWalk(x)
```

Arguments

x	A string, a R data type.

Value

A string, the PostgreSQL data type for x.

Examples

```
sqlTypeWalk(100.1209)
```

---

sql_80_char_comment *Add a 80 char SQL comment, intended to be used for visual breaks in documents.*

Description

Add a 80 char SQL comment, intended to be used for visual breaks in documents.

Usage

```
sql_80_char_comment()
```

Value

A string, 80 chars of "-".

Examples

```
sql_80_char_comment()
```

`sql_comment` *Add a single line SQL comment.*

Description

Add a single line SQL comment.

Usage

```
sql_comment(x)
```

Arguments

`x`	A string.

Value

A string prefixed with "–".

Examples

```
sql_comment("Sample single line comment.")
```

`TRUNCATE` *Generate a PostgreSQL TRUNCATE statement, optionally execute the statement if con is not NULL.*

Description

Generate a PostgreSQL TRUNCATE statement, optionally execute the statement if con is not NULL.

Usage

```
TRUNCATE(
  schema = NULL,
  table,
  restart_identity = FALSE,
  continue_identity = FALSE,
  cascade = FALSE,
  restrict = FALSE,
  con = NULL
)
```

Arguments

`schema`	A string, the schema name of the SQL table to TRUNCATE.
`table`	A string, the table name of the SQL table to TRUNCATE.
`restart_identity`	TRUE/FALSE, if TRUE, will add RESTART IDENTITY to the statement.
`continue_identity`	TRUE/FALSE, if TRUE, will add CONTINUE IDENTITY to the statement.
`cascade`	TRUE/FALSE, if TRUE, will add CASCADE to the statement.
`restrict`	TRUE/FALSE, if TRUE, will add RESTRICT to the statement.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL DELETE statement; or the results retrieved by DBI::dbGetQuery after executing the statement.

Examples

```
TRUNCATE(
schema = "test",
table = "table1"
)
```

UPDATE

Generate a PostgreSQL UPDATE statement, optionally execute the statement if con is not NULL.

Description

Generate a PostgreSQL UPDATE statement, optionally execute the statement if con is not NULL.

Usage

```
UPDATE(
  x,
  schema = NULL,
  table,
  where = list(),
  prepare = TRUE,
  types = NULL,
  returning = NULL,
  quote_text = TRUE,
  cast = TRUE,
  con = NULL
)
```

Arguments

`x`	A named list, names must match the column names of the destination SQL table, values are the values to set the SQL records to.
`schema`	A string, the schema name of the destination SQL table.
`table`	A string, the table name of the destination SQL table.
`where`	A named list, names are the columns for comparison, values are lists with a comparison operator and a value the comparison operator will check against. ex: list(col1 = list(comparison = "=", value = quoteText("b")))
`prepare`	TRUE/FALSE, if TRUE, creates a PostgreSQL prepared statement for inserting the data.
`types`	A vector of character strings specifying the SQL data types of the destination columns, the position of the type should match the position of the column for that type in x. Required if prepare or cast is TRUE.
`returning`	A vector of character strings specifying the SQL column names to be returned by the INSERT statement.
`quote_text`	TRUE/FALSE, if TRUE, calls quoteText() to add single quotes around character strings.
`cast`	TRUE/FALSE, if TRUE, will add SQL to cast the data to be inserted to the specified type.
`con`	A database connection that can be passed to DBI::dbSendQuery/DBI::dbGetQuery.

Value

A string, PostgreSQL UPDATE statement; or a string, PostgreSQL prepared statement; or the results retrieved by DBI::dbGetQuery after executing the statement.

Examples

```
UPDATE(
x = list(col1 = "a", col2 = 1),
schema = "test",
table = "table1",
where = list(
  col1 = list(comparison = "=", value = quoteText("b")),
  col2 = list(comparison = "IS", value = "NULL")
),
prepare = FALSE,
types = c("TEXT", "INTEGER"),
returning = c("col3"),
quote_text = TRUE,
cast = TRUE
)
```

vecToArrayStr

Write a PostgreSQL array as a string using ARRAY[] format from a vector.

Description

Write a PostgreSQL array as a string using ARRAY[] format from a vector.

Usage

```
vecToArrayStr(x, quote = TRUE)
```

Arguments

`x`	A vector.
`quote`	TRUE/FALSE, if TRUE, the elements of x will be quoted.

Value

A string.

Examples

```
vecToArrayStr(c("a", "b"))
```

vecToArrayStr2

Write a PostgreSQL array as a string using format from a vector.

Description

Write a PostgreSQL array as a string using format from a vector.

Usage

```
vecToArrayStr2(x, double_quote = TRUE)
```

Arguments

`x`	A vector.
`double_quote`	TRUE/FALSE, if TRUE, the elements of x will be double quoted.

Value

A string.

Examples

```
vecToArrayStr2(c("a", "b"))
```

Index