Package 'rpg'

February 22, 2015

Type Package

Title Easy Interface to Advanced PostgreSQL Features

Version 1.4

Date 2015-02-21

Author Timothy H. Keitt

Maintainer Timothy H. Keitt <tkeitt@gmail.com>

Description Allows ad hoc queries and reading and writing data frames to and from a database.

License GPL (>= 2)

URL http://www.postgresql.org/, http://github.com/thk686/rpg, http://r-forge.r-project.org/projects/rpg/, http://www.keittlab.org/

Imports Rcpp (>= 0.11.1), uuid (>= 0.1-1), RApiSerialize (>= 0.1.0), tcltk (>= 3.0.0)

Suggests foreach (>= 1.4.2), doParallel (>= 1.0.0), hflights (>= 0.1)

LinkingTo Rcpp, RApiSerialize

SystemRequirements libpq (now included with rpg)

Depends R (>= 3.0.0)

NeedsCompilation yes

Note The files configure.in, cleanup, Makevars.in and Makevars.win are based on the same files found in RPostgreSQL. I thank Dirk Eddelbuettel for giving the go ahead to reuse their build setup. Copyright of the unmodified protions remains with the original authors. The files in src/libpq/ are part of PostgreSQL. Please refer to http://www.postgresql.org/ for their copyright and license.

Repository CRAN

Repository/R-Forge/Project rpg

Repository/R-Forge/Revision 28

Repository/R-Forge/DateTimeStamp 2015-02-21 17:25:43

Date/Publication 2015-02-22 08:24:06

R topics documented:

`rpg-package` *Easy Access to Advanced PostgreSQL Features*

Description

Provides functions for connecting to, reading from and writing to a PostgreSQL database. Facilities for tracing the communication between R and PostgreSQL are provided, as are function to retieve detailed session metadata.

Details

Package:	rpg
Type:	Package
Version:	1.4
Date:	2015-2-21
License:	GPL

The main functions are `connect`, which establishes a connection, `query`, which issues queries and `fetch`, which retieves results. Intelligent defaults are used throughout. Functions that require a connection will automatically attempt to establish a valid connection based on a previous connection or from defaults. The defaults can be overriden in a variety of ways.

Author(s)

Timothy H. Keitt
http://www.keittlab.org/

Maintainer: Timothy H. Keitt <tkeitt@gmail.com>

References

http://github.com/thk686/rpg, http://www.postgresql.org/

`async_query` *Asynchronous query processing*

Description

Manage an asynchronous query

Usage

```
async_query(sql = "", pars = NULL)

async_status()

is_busy()

cancel()

finish_async()
```

Arguments

`sql`	a query string
`pars`	a vector of parameters

Details

These functions expose the asynchronous query interface from `libpq`. The function `async_query` issues a query. Its call is identical to `query` except that it will return immediately. When the issued command is ready, the function `async_status` will return a query status object exactly as `query`. Otherwise it will return `"BUSY"` to indicate the server has not finished or `"DONE"` to indicate there is nothing more to fetch.

If `async_status` does not return `"DONE"`, then you should call `finish_async` to free pending results. Note that a call to `finish_async` may block until the server is finished processing the command. It calls `cancel` internally but there is no guarantee the command will abort.

Any pending results will be lost if you call query, execute or fetch with a sql string prior to async_query returning DONE. If you need to issue queries while waiting on an async call, then use push_conn to save the query state, connect to make a new connetion, and then pop_conn followed by async_status.

is_busy is a slightly faster shortcut to check whether the server has completed the query. You must still call async_status to fetch the results.

Value

async_query: true if query was successfully sent (an invalid query will still return true)

async_status: a results status object, possibly indicating an invalid query

is_busy: a boolean

Note

In practice, you will be much better off using cursor as that will usually return very quickly even for large queries, and has the advantage of retrieving the results in chunks. You can call cancel while a cursor is active. The cursor will return PGRES_FATAL_ERROR if the cancel is effective. Alternately, issuing any query that sets the result status will have the same effect as finish_async.

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
# create a database
system("createdb rpgtesting")
connect("rpgtesting")
begin()

# write data frame contents
data(mtcars)
write_table(mtcars)

# async processing on smallish result
# this wont be interesting if your machine is very fast
async_query("SELECT a.* FROM mtcars a, mtcars b")
repeat
{
  status = async_status()
  if ( status != "BUSY" ) break
  cat("busy...\n")
  Sys.sleep(1)
}
print(status)
head(fetch())
finish_async()
Sys.sleep(1)
```

```
# async processing on larger result
async_query("SELECT a.* FROM mtcars a, mtcars b, mtcars c")
count = 0
repeat
{
  status = async_status()
  if ( status == "BUSY" )
  {
    if ( count > 2 )
    {
      cat("calling cancel...\n")
      cancel()
    }
  }
  else break
  cat("busy... \n")
  Sys.sleep(1)
  count = count + 1
}
print(status)
finish_async()

# you can run multiple queries with async_query
rollback(); begin()
write_table(mtcars)
sql1 = "SELECT mpg FROM mtcars LIMIT 3"
sql2 = "SELECT cyl FROM mtcars LIMIT 4"
async_query(paste(sql1, sql2, sep = "; "))
while ( async_status() == "BUSY" ) NULL
fetch()
while ( is_busy() ) NULL
async_status()
fetch()
finish_async()

# issue an async query and come back later
async_query(sql1)
push_conn()
connect("rpgtesting")

# fails because of transaction isolation
fetch(sql2)
pop_conn()
async_status()

# results from sql1
fetch()

# this is automatic if you issue new queries
finish_async()

# cleanup
```

```
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

---

begin *Transaction support*

Description

Start, commit or rollback transactions or savepoints

Usage

```
begin()

commit(savepoint = NULL)

rollback(savepoint = NULL)

savepoint()
```

Arguments

savepoint	an object produced by savepoint

Details

These functions allow manipulation of database transaction states. If no savepoint object is supplied, then an attempt is made to commit or rollback the current transaction.

The savepoint function will initiate a transaction if one is not currently active. In that case, no actual PostgreSQL savepoint will be used. Rolling back the savepoint will rollback the initiated transaction. If a tranaction is active, then a named savepoint will be generated. You can rollback to the database state when savepoint was called or commit all changes.

Value

savepoint: a savepoint object

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
system("createdb rpgtesting")
connect("rpgtesting")
begin()
sp1 = savepoint()

# nest savepoints
sp2 = savepoint()
data(mtcars)
write_table(mtcars, "testtab", overwrite = TRUE)
list_tables()
rollback(sp2)

list_tables()
# nest savepoints
sp3 = savepoint()
sp4 = savepoint()
write_table(mtcars, "testtab", overwrite = TRUE)
commit(sp4)
list_tables()
rollback(sp3)
list_tables()

rollback(sp1)
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

`copy_from` *Bulk read and write*

Description

Read from and write to a database using COPY

Usage

```
copy_from(what, psql_opts = "")

copy_to(x, tablename, schemaname = NULL, append = FALSE, psql_opts = "")
```

Arguments

`what`	a table name or sql query string
`psql_opts`	passed directly to the psql command line
`x`	a data frame

tablename	name of table to create
schemaname	create table in this schema
append	if false, drop and receate table

Details

These functions use the SQL COPY command and therefore are much faster than write_table and possibly read_table. These functions also call PostgreSQL's psql command from the command line and will fail if it is not found on the search path.

Because these functions shell out to psql you do not need an active connection. By specifying psql_opts you can connect to any database without affecting the active connection. If you do not specify psql_opts an attempt will be made to use the active connection information. If that fails, psql will use default connection settings.

Note

These functions call read.csv and write.csv and so will suffer the same bandwidth limitations as those functions. I argue that is good enough. There is little point in reading and writing datasets too large for those functions in R. Better to bulk load using psql on the command line and then use cursor to read the data in small bits.

Author(s)

Timothy H. Keitt

See Also

set_default_password

Examples

```
## Not run:
# example requires hflights
if ( ! require(hflights, quietly = TRUE) )
 stop("This example requires the \'hflights\' package")

# big dataset
data(hflights)
dim(hflights)

system(paste("createdb rpgtesting"))

opts = paste("-d rpgtesting")
system.time(copy_to(hflights, psql_opts = opts))
system.time(invisible(copy_from("hflights", psql_opts = opts)))

connect("rpgtesting")
begin()

## Sloooowwwwwww
```

```
## system.time(write_table(hflights))
system.time(invisible(read_table("hflights")))

rollback()
disconnect()
system(paste("dropdb rpgtesting"))
## End(Not run)
```

`cursor` *Iterator support*

Description

Construct a row iterator

Usage

```
cursor(sql, by = 1, pars = NULL)
```

Arguments

`sql`	any valid query returning rows
`by`	how many rows to return each iteration
`pars`	optional query parameters

Details

This function generates an interator object that can be used with the `foreach-package`.

It is possible to use the `%dopar%` operator as shown in the example below. You must establish a connection to the database on each node and in your current session because the call to `cursor` requires it. Note that the cursor's lifetime is the current transaction block, so if anything happens to the transaction or you call `END` or `ROLLBACK`, then the cursor will no longer function. Apparently a named SQL cursor is visible to any database session, as evidenced by the example below, even though it is declared within a transaction. This is not stated explicitely in the PostgreSQL documentation.

Note

There are some reports of issues using multicore (forking) with RStudio.

Author(s)

Timothy H. Keitt

See Also

`foreach`, `rollback`, `query`

Examples

```
## Not run:
# example requires foreach
if ( ! require(foreach, quietly = TRUE) )
 stop("This example requires the \'foreach\' package")

# connect using defaults
system("createdb rpgtesting")
connect("rpgtesting")
begin()

# write data frame contents
data(mtcars)
write_table(mtcars, row_names = "id", pkey = "id", overwrite = TRUE)

# expand rows to columns 8 rows at a time
x = foreach(i = cursor("SELECT * FROM mtcars", by = 8),
            .combine = rbind) %do% { i$mpg }
print(x, digits = 2)

# parallel example
if ( require(doParallel, quietly = TRUE) )
{
 # make the cluster
 cl = makeCluster(2)

 # must connect to database on each node
 clusterEvalQ(cl, library(rpg))
 clusterEvalQ(cl, connect("rpgtesting"))
 clusterEvalQ(cl, begin())

 # setup the dopar call
 registerDoParallel(cl)

 # take column averages 4 rows at a time
 curs1 = cursor("SELECT * FROM mtcars", by = 4)
 x = foreach(i = curs1, .combine = rbind, .inorder = FALSE) %dopar%
 {
   rr = paste0(range(abbreviate(i$id)), collapse = "-")
   pid = get_conn_info("server.pid")
   j = names(i) != "id"
   mn = signif(apply(i[, j], 2, mean), 2)
   c(rows = rr, backend = pid, mn)
 }
 x = as.data.frame(x)
 row.names(x) = x$rows
 x$rows = NULL
 print(noquote(x))

 clusterEvalQ(cl, rollback())
 stopCluster(cl)
}
```

```
#cleanup
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

`disconnect` *PostgreSQL connection*

Description

Manage database connection

Usage

```
disconnect()

connect(dbname, ...)
```

Arguments

`dbname`	name of the database or a valid `libpq` connection string
`...`	named optional connection parameters

Details

`disconnect` will free any query results as well as clean up the connection data. It is called in the pakcage `.Last.lib` function when exiting `R`.

If no connection parameters are supplied, the connection will fallback to default parameters. Usually this establishes a connection on the localhost to a database, if it exists, with the same name as the user.

Valid keywords and their defaults can be obtained by calling `get_conn_defaults(all = TRUE)`. A valid `libpq` connection string is composed of `keyword = value` pairs separated by whitespace. You can either pass the entire string or use named arguments. The names of the arguments will be used as keywords and their values as values.

If a password was required but not provided, `connect` will will open a dialog and prompt for a password. The connection is then re-tried and the status returned.

Value

`connect` returns one of:

`CONNECTION_OK`	Succesful connection
`CONNECTION_BAD`	Connection failed

Note

Do not open a connection and then fork the R process. The behavior will be unpredictable. It is perfectly acceptable however to call `connect` within each forked instance.

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
fetch("SHOW search_path") # default connection
connect("test")
connect(dbname = "test")
connect(dbname = "test", host = "localhost")
connect("dbname = test host = localhost")
disconnect()
## End(Not run)
```

---

format_for_send *Convert R objects to strings*

Description

Prepare R objects for sending to postgresql

Usage

```
format_for_send(obj)
```

Arguments

`obj`	any object

Details

R objects that will be written to postgresql must be converted to characters as all data is transferred to the server as text. The S3 method `foramt_for_send` accomplishes this. It accepts any object and returns a character representation.

You can define new conversions by supplying your own S3 override of `format_for_send`.

libpq_version *Miscelaneous functions*

Description

Various utility functions

Usage

```
libpq_version()

encrypt_password(passwd, user)

get_encoding()

set_encoding(encoding)

set_error_verbosity(verbosity)

enable_postgis(schemaname = "postgis")
```

Arguments

passwd	the password
user	the user name
encoding	the character encoding
verbosity	one of "terse", "default", "verbose"
schemaname	install in this schema

Details

enable_postgis will attempt to install the postgis extension in the named schema. The default search path is altered to include the new schema.

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
# try connecting to default database
system("createdb rpgtesting")
connect("rpgtesting")
begin()

libpq_version()
```

```
encrypt_password("test", "tester")
get_encoding()
set_encoding("UTF8")
set_error_verbosity("terse")
set_error_verbosity("verbose")
set_error_verbosity("default")
enable_postgis()

# cleanup
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

`list_tables` *PostgreSQL database information*

Description

Get information about tables in a database

Usage

```
list_tables(only.names = TRUE)

describe_table(tablename, schemaname = NULL)
```

Arguments

`only.names`	if true, just list the table names
`tablename`	the name of a PostgreSQL table
`schemaname`	if not null, look only in this schema

Value

`list_tables`: a vector of table names or a data frame

`describe_table`: a data frame with column information

Author(s)

Timothy H. Keitt

See Also

`psql`

Examples

```
## Not run:
system("createdb rpgtesting")
connect("rpgtesting")
begin()

# write data frame contents
data(mtcars)
write_table(mtcars)

# get some information
list_tables()
describe_table("mtcars")

#cleanup
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

---

ping *Database connection utilities*

Description

Conection reporting and defaults

Usage

```
ping(opts = "")

get_conn_error()

get_conn_defaults(all = FALSE)

get_conn_info(what = NULL)

set_conn_defaults(...)

set_default_password(password = NULL)

reset_conn_defaults()
```

Arguments

opts	a libpq connection string
all	if false return only defaults with settings
what	the fields to return or all if NULL

...	a named list of arguments giving new defaults
password	the password

Details

ping will ignore any keywords not directly related to the database host (e.g., username, dbname) as it does not connect; it only detect the server port is responding.

get_conn_defaults returns a data frame containing all of the possible connection string keywords, the names of environment variables used to override the defaults, the compiled in default value and the current value of the keyword.

get_conn_info returns a list containing information about the current connection. For readability, it will print as though it is a matrix. If you want to see it as a list, try unclass(get_conn_info()).

If length(what) == 1 then get_conn_info returns a scalar

set_conn_defaults sets the connection defaults by calling Sys.setenv and setting the environment variable associated with the connection keywords returned by get_conn_defaults(all = TRUE). These settings will only last as long as the current shell session and will reset after a new login.

set_default_password will query for a password (if not supplied) and set the PGPASSWORD environment variable accordingly. This can be used with psql and copy_to.

reset_conn_defaults unsets all environment variables returned by get_conn_defaults(all = TRUE).

Value

ping returns one of the following:

PQPING_OK	Server reachable
PQPING_REJECT	Server reachable but not accepting connections
PQPING_NO_RESPONSE	Server unreachable
PQPING_NO_ATTEMPT	Connection string is nonsense

get_conn_error: an error string

get_conn_defaults: a data frame with defaults listed

get_conn_info: a list of values

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
ping("connect_timeout = 3, host = www.keittlab.org")
connect()
get_conn_defaults()
set_conn_defaults(dbname = "test")
get_conn_defaults()
reset_conn_defaults()
```

```
get_conn_defaults()
get_conn_defaults(all = TRUE)
get_conn_info()
get_conn_error()
disconnect()
## End(Not run)
```

---

prepare *Prepared queries*

Description

Prepare and execute queries

Usage

```
prepare(sql)
```

Arguments

sql	a valid query string

Details

`prepare` prepares a statement for later execution. It returns a function that when called executes the prepared statement. Values passed to the returned function will substituted for parameters in the prepared statement. If the number of parameters supplied is a multiple of the number of open parameters in query prepared using `prepare`, then the prepared query will be executed repeatedly for each successive set of parameters. This repeated execution loop is evaluted in C++ and so is quite fast. The supplied parameter values will be coerced to a matrix of the appropriate dimensions. Values passed to the function will be recycled to match the number of query parameters. The passed parameters will be coerced to character strings.

Value

A function.

The function can take one argument. The values will be used to fill in parameters of the prepared statement. If no argument is passed, the statement will be executed without any parameters.

Note

One can use pure SQL to achieve the same result.

It is generally a good idea to wrap `prepare` in a transaction. If not in a transaction, you cannot rollback any updates and it will be much slower as PostgreSQL initiates a transaction-per-query by default.

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
# try connecting to default database
system("createdb rpgtesting")
connect("rpgtesting")
begin()

# write data frame contents
data(mtcars)
write_table(mtcars)

# delete the rows
query("truncate mtcars")
read_table(mtcars)

# use prepare-execute to write rows
pars = paste0("$", 1:11, collapse = ", ")
sql = paste0("INSERT INTO mtcars VALUES (", pars, ")", collapse = " ")
f = prepare(sql)
f(mtcars)
read_table(mtcars, limit = 5)

# cleanup
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

---

psql *PostgreSQL shell*

Description

Run PostgreSQL's psql shell interactively

Usage

```
psql(psql_opts = "")
```

Arguments

psql_opts	a character string passed to the psql command

Details

The `psql` function repeatedly queries for input and pipes it to PostgreSQL's psql command. It will terminate on `\q` or empty input.

If `psql_opts` is an empty string, then an attempt will be made to supply suitable options based on the current connection. If there is no active connection, psql will fallback to complied in defaults. If `psql_opts` is not an empty string, then it will be passed as-is to psql.

You can type psql's escape commands as usual. Try `\?`. You cannot use `\e` or `\ef` to evoke an editor. Doing strange things with `\!` will likely hang the R session.

There is no way to direclty enter a database password. If one is required, you can use a password file or `set_conn_defaults`.

Unfortunately it is probably impossible to enable GNU readline support so for example up-arrow will recall your R commands, not the psql commands entered. You can always call psql from a terminal.

Author(s)

Timothy H. Keitt

See Also

`set_default_password`

---

push_conn *Multiple PostgreSQL connections*

Description

Manage multiple connections on a stack

Usage

```
push_conn()

pop_conn()

swap_conn()

rotate_stack(n = 1L)

show_conn_stack()
```

Arguments

n	number of shifts

Details

These functions allow you to store multiple connections on a stack. They are only used for their side-effects. rpg stores an active connection pointer internally. This pointer can be moved onto the stack and manipulated. Once on the stack, the pointer is no longer active. You must use swap_conn or pop_conn to reactive a pushed connection, or call connect to create a new active connection.

push_conn pushes the current connection onto the connection stack leaving the active connection null.

pop_conn pops a connection off the stack and makes it active. Whatever connection was active when pop_conn is called will be disconnected and cleared. Use swap_conn to preserve the active connection.

swap_conn swaps the active connection with the connection on the top of the stack. If the stack is empty, the connection is swapped with a null connection.

rotate_stack moves the bottom of the stack to the top.

show_conn_stack returns a data frame with information about the connections on the stack.

Examples

```
## Not run:
# make some databases
dbs = paste0("rpgdb", 1:3)
lapply(paste("createdb", dbs), system)

# connect
connect(dbname = dbs[1]); push_conn()
connect(dbname = dbs[2]); push_conn()
connect(dbname = dbs[3]); push_conn()

show_conn_stack()
rotate_stack()
show_conn_stack()
rotate_stack(2)
show_conn_stack()
pop_conn()
show_conn_stack()
get_conn_info("dbname")
swap_conn()
show_conn_stack()
get_conn_info("dbname")
pop_conn()
show_conn_stack()
pop_conn()
show_conn_stack()
disconnect()
connect()
lapply(paste("dropdb", dbs), system)
## End(Not run)
```

`result_dim` *PostgreSQL query*

Description

Issue a query to the current database connection

Usage

```
result_dim()

query(sql = "", pars = NULL)

query_error()

fetch(sql = "", pars = NULL)

execute(...)
```

Arguments

`sql`	a query string
`pars`	a character vector of substitution values
`...`	list of commands to be `paste`d together

Details

`fetch` returns the result of a query as a data frame. If `sql` is `NULL` or empty, then an attempt will be made to retrieve any pending resutls from previous queries. Note that query results are not cleared until the next query is issued so `fetch` will continue to return results until a new query is issued.

`execute` is a wrapper around `query`. It will raise an exception if the command does not complete. Exceptions can be caught with `tryCatch`. You cannot use a parameterized query with `execute`. Unlike `query` it will `paste` its arguments into a single string.

Value

`result_dim` returns the number of tuples and fields

`query` returns:

PGRES_EMPTY_QUERY	The string sent to the server was empty
PGRES_COMMAND_OK	Successful completion of a command returning no data
PGRES_TUPLES_OK	Successful completion of a command returning data (such as a SELECT or SHOW)
PGRES_COPY_OUT	Copy Out (from server) data transfer started
PGRES_COPY_IN	Copy In (to server) data transfer started
PGRES_BAD_RESPONSE	The server's response was not understood.
PGRES_NONFATAL_ERROR	A nonfatal error (a notice or warning) occurred

PGRES_FATAL_ERROR	A fatal error occurred
PGRES_COPY_BOTH	Copy In/Out (to and from server) data transfer started. This is currently used only for stream

`query_error` returns an error string

`fetch` returns a data frame or a query status object on failure.

`execute` the result status string

Author(s)

Timothy H. Keitt

See Also

`psql`

Examples

```
## Not run:
system("createdb rpgtesting")
connect("rpgtesting")
begin()
execute("DROP SCHEMA IF EXISTS rpgtesting CASCADE")
execute("CREATE SCHEMA rpgtesting")
execute("SET search_path TO rpgtesting")
execute("DROP TABLE IF EXISTS test")
execute("CREATE TABLE test (id integer, field text)")
query("INSERT INTO test VALUES ($1, $2)", c(1, "test"))
fetch("SELECT * FROM test")
result_dim()
fetch("SELECT * FROM testing")
query_error()
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

---

stow *Object storage*

Description

Serialize and write R objects

Usage

```
stow(..., tablename = "rpgstow", schemaname = "rpgstow")

list_stowed(tablename = "rpgstow", schemaname = "rpgstow")

retrieve(objnames, tablename = "rpgstow", schemaname = "rpgstow")

delete_stowed(objnames, tablename = "rpgstow", schemaname = "rpgstow")

stow_image(imagename = "rpgimage", schemaname = "rpgstow")

retrieve_image(imagename = "rpgimage", schemaname = "rpgstow")
```

Arguments

`...`	a list of objects or object names
`tablename`	the table for storing objects
`schemaname`	the schema to use
`objnames`	a character vector with object names or regular expressions
`imagename`	a table name for stowing the session image

Details

These functions allow one to write out any R object to a PostgreSQL database and later retrieve them into any R session. The pair `stow` and `retrieve` are modeled roughly as `save` and `load`.

The contents of `...` are handled specially. If a named argument is passed, then the object will be stowed and retrieved with that name. A raw string will not be stowed if it matches the name of any R object; the matching R object will be stowed instead. A vector of strings will however be stowed as is. Object names will be coerced to valid identifiers using `make.names`. Names must be unique and not collide with any existing object name in the same table.

`retrieve` assigns the stowed values to the stowed names in the global environment. It will overwrite any variable that has the same name as a stowed object.

The functions `retrieve` and `delete_stowed` use regular expression matching as implemented by the PostgreSQL ~ operator.

`stow_image` and `retrieve_image` will stow all objects in the current session and retrieve them later. Note that `stow_image` will overwrite all existing objects stowed within `imagename`.

Author(s)

Timothy H. Keitt

See Also

`RApiSerialize`

Examples

```
## Not run:
system("createdb rpgtesting")
connect("rpgtesting")
begin()

stow("test")
list_stowed()
stow("test")
list_stowed()
stow(x = "test")
list_stowed()
x = 1
stow(x)
list_stowed()
stow(y = x)
list_stowed()
rm(x)
retrieve(".*")
print(test)
print(x)
print(y)
delete_stowed(".*")
data(mtcars)
stow(mtcars)
list_stowed()
rm(mtcars)
retrieve("mtcars")
head(mtcars)

rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

`trace_conn` *PostgeSQL connection tracing*

Description

Functions to manage connection tracing

Usage

```
trace_conn(filename = "", append = FALSE)

untrace_conn(remove = FALSE)

trace_filename()
```

```
dump_conn_trace(warn = FALSE, ...)
```

Arguments

filename	where to send the tracing
append	if true, append to existing file
remove	if true, unlink the tracing file
warn	if true, readLines will issue warnings
...	passed to readLines

Details

PostgeSQL tracing lets you observe all information passing between rpg and the database server.

trace_conn begins tracing and untrace_conn stops tracing.

dump_conn_trace invokes readLines on the trace file.

Value

trace_filename: the name of the file containing trace information.

Author(s)

Timothy H. Keitt

Examples

```
## Not run:
system("createdb rpgtesting")
connect("rpgtesting")
trace_conn()
list_tables()
dump_conn_trace(n = 40)
untrace_conn(remove = TRUE)
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

---

write_table *PostgreSQL data frame IO*

Description

Reads and writes table to and from database

Usage

```
write_table(x, tablename, pkey = NULL, row_names = NULL,
  schemaname = NULL, types = NULL, overwrite = FALSE)

read_table(tablename, what = "*", limit = NULL, row_names = NULL,
  schemaname = NULL, pkey_to_row_names = FALSE)
```

Arguments

x	a data frame or something convertible to a data frame
tablename	the name of the table to read from or write to
pkey	a column name to use as primary key
row_names	a column name to write row names
schemaname	the schema name
types	a list of valid PostgreSQL type names
overwrite	if true, destroy existing table with the same name
what	a vector of column names
limit	only return this many rows
pkey_to_row_names	if true and row_names not given, use primary key column

Details

A table is created using the current connection. If pkey does not match any column name, then a new column is created with the name given by pkey. Its type will be serial and it will be set as the primary key. If pkey does match an existing column name, then that column will be used as the primary key. Note that make.unique will be called on the column names before this matching is done. If row_names is a character string, the data frame row names will be stored in a column with the column name given by row_names. The row_names column can also be the primary key if pkey is the same as row_names.

If row_names is specified when calling read_table, then the resulting data frame will have row names installed from the column named in row_names. Note that the column named in row_names must match a column specified by what. The matching column will be removed from the data frame.

If types is not supplied, they will be computed from the classes and types of the columns of input.

Value

write_table the final query status

read_table a data frame

Note

The entire process is wrapped within a transcation. On failure at any point, the transaction will be rolled back and the database unaffected.

Also, `write_table` uses SQL `INSERT` statements and as such will be slow for large tables. You are much better off bulk loading data using the `COPY` command outside of `R`.

Author(s)

Timothy H. Keitt

See Also

`copy_from`

Examples

```
## Not run:
# connect using defaults
system("createdb rpgtesting")
connect("rpgtesting")
begin()

# write data frame contents
data(mtcars)
write_table(mtcars)

# make "cyl" primary key (will fail unique constraint)
write_table(mtcars, pkey = "cyl", overwrite = TRUE)

# also write row names to "id"
write_table(mtcars, row_names = "id", overwrite = TRUE)

# row names as primary key
write_table(mtcars, row_names = "id", pkey = "id", overwrite = TRUE)

# default R row names and only first 3 columns
read_table("mtcars", what = "mpg, cyl, disp", limit = 3)

# row names from column "id"
read_table("mtcars", row_names = "id", limit = 3)

# get row names from primary key
read_table("mtcars", pkey_to_row_names = TRUE, limit = 3)

#cleanup
rollback()
disconnect()
system("dropdb rpgtesting")
## End(Not run)
```

Index