Package 'rpostgis'

November 20, 2019

Version 1.4.3

Date 2019-11-19

Title R Interface to a 'PostGIS' Database

Description Provides an interface between R and 'PostGIS'-enabled 'PostgreSQL' databases to transparently transfer spatial data. Both vector (points, lines, polygons) and raster data are supported in read and write modes. Also provides convenience functions to execute common procedures in 'PostgreSQL/PostGIS'.

SystemRequirements 'PostgreSQL' with 'PostGIS' extension

Depends R (>= 3.3.0),
RPostgreSQL,
DBI (>= 0.5)

Imports methods,
raster,
rgeos,
sp,
stats

Suggests RPostgres,
rgdal,
wkb

License GPL (>= 3)

LazyData true

URL https://mablab.org/rpostgis/index.html

BugReports https://github.com/mablab/rpostgis/issues

RoxygenNote 7.0.0

R topics documented:

`dbAddKey` *Add key.*

Description

Add a primary or foreign key to a table column.

Usage

```
dbAddKey(
  conn,
  name,
  colname,
  type = c("primary", "foreign"),
  reference,
  colref,
  display = TRUE,
  exec = TRUE
)
```

Arguments

`conn`	A connection object.
`name`	A character string, or a character vector, specifying a PostgreSQL table name.
`colname`	A character string specifying the name of the column to which the key will be assign; alternatively, a character vector specifying the name of the columns for keys spanning more than one column.

type	The type of the key, either "primary" or "foreign"
reference	A character string specifying a foreign table name to which the foreign key will be associated (ignored if type == "primary").
colref	A character string specifying the name of the primary key in the foreign table to which the foreign key will be associated; alternatively, a character vector specifying the name of the columns for keys spanning more than one column (ignored if type == "primary").
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

TRUE if the key was successfully added.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-altertable.html

Examples

```
## Examples use a dummy connection from DBI package
conn <- DBI::ANSI()

## Primary key
dbAddKey(conn, name = c("sch1", "tbl1"), colname = "id1", exec = FALSE)

## Primary key using multiple columns
dbAddKey(conn, name = c("sch1", "tbl1"), colname = c("id1", "id2",
    "id3"), exec = FALSE)

## Foreign key
dbAddKey(conn, name = c("sch1", "tbl1"), colname = "id", type = "foreign",
    reference = c("sch2", "tbl2"), colref = "id", exec = FALSE)

## Foreign key using multiple columns
dbAddKey(conn, name = c("sch1", "tbl1"), colname = c("id1", "id2"),
    type = "foreign", reference = c("sch2", "tbl2"), colref = c("id3",
        "id4"), exec = FALSE)
```

dbAsDate *Converts to timestamp.*

Description

Convert a date field to a timestamp with or without time zone.

Usage

```
dbAsDate(conn, name, date = "date", tz = NULL, display = TRUE, exec = TRUE)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL table name.
date	A character string specifying the date field.
tz	A character string specifying the time zone, in "EST", "America/New_York", "EST5EDT", "-5".
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

If exec = TRUE, returns TRUE if the conversion was successful.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/datatype-datetime.html

Examples

```
## Example uses a dummy connection from DBI package
conn <- DBI::ANSI()
dbAsDate(conn, name = c("schema", "table"), date = "date", tz = "GMT",
    exec = FALSE)
```

dbColumn *Add or remove a column.*

Description

Add or remove a column to/from a table.

Usage

```
dbColumn(
  conn,
  name,
  colname,
  action = c("add", "drop"),
  coltype = "integer",
  cascade = FALSE,
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL table name.
colname	A character string specifying the name of the column
action	A character string specifying if the column is to be added ("add", default) or removed ("drop").
coltype	A character string indicating the type of the column, if action = "add".
cascade	Logical. Whether to drop foreign key constraints of other tables, if action = "drop".
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

TRUE if the column was successfully added or removed.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-altertable.html

Examples

```
## examples use a dummy connection from DBI package
conn<-DBI::ANSI()
## Add an integer column
dbColumn(conn, name = c("schema", "table"), colname = "field", exec = FALSE)
## Drop a column (with CASCADE)
dbColumn(conn, name = c("schema", "table"), colname = "field", action = "drop",
    cascade = TRUE, exec = FALSE)
```

---

dbComment *Comment table/view/schema.*

Description

Comment on a table, a view or a schema.

Usage

```
dbComment(
  conn,
  name,
  comment,
  type = c("table", "view", "schema"),
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL table, view or schema name.
comment	A character string specifying the comment.
type	The type of the object to comment, either "table", "view", or "schema"
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

TRUE if the comment was successfully applied.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-comment.html

Examples

```
## examples use a dummy connection from DBI package
conn<-DBI::ANSI()
dbComment(conn, name = c("schema", "table"), comment = "Comment on a view.",
    type = "view", exec = FALSE)
dbComment(conn, name = "test_schema", comment = "Comment on a schema.", type = "schema",
    exec = FALSE)
```

---

dbDrop *Drop table/view/schema.*

Description

Drop a table, a view or a schema.

Usage

```
dbDrop(
  conn,
  name,
  type = c("table", "schema", "view", "materialized view"),
  ifexists = FALSE,
  cascade = FALSE,
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL table, schema, or view name.
type	The type of the object to drop, either "table", "schema", "view", or "materialized view".
ifexists	Do not throw an error if the object does not exist. A notice is issued in this case.
cascade	Automatically drop objects that depend on the object (such as views).
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

TRUE if the table/schema/view was successfully dropped.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-droptable.html, http://www.postgresql.org/docs/current/static/sql-dropview.html, http://www.postgresql.org/docs/current/static/sql-dropschema.html

Examples

```
## examples use a dummy connection from DBI package
conn<-DBI::ANSI()
dbDrop(conn, name = c("schema", "view_name"), type = "view", exec = FALSE)
dbDrop(conn, name = "test_schema", type = "schema", cascade = "TRUE", exec = FALSE)
```

---

dbIndex *Create an index.*

Description

Defines a new index on a PostgreSQL table.

Usage

```
dbIndex(
  conn,
  name,
  colname,
  idxname,
  unique = FALSE,
  method = c("btree", "hash", "rtree", "gist"),
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL table name.
colname	A character string, or a character vector specifying the name of the column to which the key will be associated; alternatively, a character vector specifying the name of the columns to build the index.
idxname	A character string specifying the name of the index to be created. By default, this uses the name of the table (without the schema) and the name of the columns as follows: <table_name>_<column_names>_idx.
unique	Logical. Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

method	The name of the method to be used for the index. Choices are "btree", "hash", "rtree", and "gist". The default method is "btree", although "gist" should be the index of choice for PostGIS spatial types (geometry, geography, raster).
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

TRUE if the index was successfully created.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-createindex.html; the PostGIS documentation for GiST indexes: http://postgis.net/docs/using_postgis_dbmanagement.html#id541286

Examples

```
## Examples use a dummy connection from DBI package
conn <- DBI::ANSI()

## GIST index
dbIndex(conn, name = c("sch", "tbl"), colname = "geom", method = "gist",
    exec = FALSE)

## Regular BTREE index on multiple columns
dbIndex(conn, name = c("sch", "tbl"), colname = c("col1", "col2",
    "col3"), exec = FALSE)
```

---

dbSchema *Check and create schema.*

Description

Checks the existence, and if necessary, creates a schema.

Usage

```
dbSchema(conn, name, display = TRUE, exec = TRUE)
```

Arguments

conn	A connection object (required, even if exec = FALSE).
name	A character string specifying a PostgreSQL schema name.
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE). Note: if exec = FALSE, the function still checks the existence of the schema, but does not create it if it does not exists.

Value

TRUE if the schema exists (whether it was already available or was just created).

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-createschema.html

Examples

```
## Not run:
    dbSchema(name = "schema", exec = FALSE)

## End(Not run)
```

---

dbTableInfo *Get information about table columns.*

Description

Get information about columns in a PostgreSQL table.

Usage

```
dbTableInfo(conn, name, allinfo = FALSE)
```

Arguments

conn	A connection object to a PostgreSQL database.
name	A character string specifying a PostgreSQL schema (if necessary), and table or view name (e.g., name = c("schema", "table")).
allinfo	Logical, Get all information on table? Default is column names, types, nullable, and maximum length of character columns.

Value

data frame

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
dbTableInfo(conn, c("schema", "table"))

## End(Not run)
```

---

dbVacuum *Vacuum.*

Description

Performs a VACUUM (garbage-collect and optionally analyze) on a table.

Usage

```
dbVacuum(
  conn,
  name,
  full = FALSE,
  verbose = FALSE,
  analyze = TRUE,
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL table name.
full	Logical. Whether to perform a "full" vacuum, which can reclaim more space, but takes much longer and exclusively locks the table.
verbose	Logical. Whether to print a detailed vacuum activity report for each table.
analyze	Logical. Whether to update statistics used by the planner to determine the most efficient way to execute a query (default to TRUE).
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).

Value

TRUE if query is successfully executed.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostgreSQL documentation: http://www.postgresql.org/docs/current/static/sql-vacuum.html

Examples

```
## examples use a dummy connection from DBI package
conn<-DBI::ANSI()
dbVacuum(conn, name = c("schema", "table"), full = TRUE, exec = FALSE)
```

---

dbWriteDataFrame *Write/read in data frame mode to/from database table.*

Description

Write data.frame to database table, with column definitions, row names, and a new integer primary key column. Read back into R with dbReadDataFrame, which recreates original data frame.

Usage

```
dbWriteDataFrame(conn, name, df, overwrite = FALSE, only.defs = FALSE)

dbReadDataFrame(conn, name, df = NULL)
```

Arguments

conn	A connection object to a PostgreSQL database
name	Character, schema and table of the PostgreSQL table
df	The data frame to write (for dbReadDataFrame, this allows to update an existing data.frame with definitions stored in the database)
overwrite	Logical; if TRUE, a new table (name) will overwrite the existing table (name) in the database. Note: overwriting a view must be done manually (e.g., with dbDrop).
only.defs	Logical; if TRUE, only the table definitions will be written.

Details

Writing in data frame mode is only for new database tables (or for overwriting an existing one). It will save all column names as they appear in R, along with column data types and attributes. This is done by adding metadata to a lookup table in the table's schema named ".R_df_defs" (will be created if not present). It also adds two fields with fixed names to the database table: ".R_rownames" (storing the row.names of the data frame), and ".db_pkid", which is a new integer primary key. Existing columns in the data.frame matching these names will be automatically changed.

The `rpostgis` database table read functions `dbReadDataFrame` and `pgGetGeom` will use the metadata created in data frame mode to recreate a data.frame in R, if it is available. Otherwise, it will be imported using default `RPostgreSQL::dbGetQuery` methods.

All `Spatial*DataFrames` must be written with `pgInsert`. For more flexible writing of `data.frames` to the database (including all writing into existing database tables), use `pgInsert` with `df.mode = FALSE`.

Value

`TRUE` for successful write with `dbWriteDataFrame`, `data.frame` for `dbReadDataFrame`

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
library(sp)
data(meuse)

## Write the data.frame to the database:
dbWriteDataFrame(conn, name = "meuse_data", df = meuse)

## Reads it back into a different object:
me2 <- dbReadDataFrame(conn, name = "meuse_data")

## Check equality:
all.equal(meuse, me2)
## Should return TRUE.

## End(Not run)
```

`pgGetBoundary` *Retrieve bounding envelope of geometries or rasters.*

Description

Retrieve bounding envelope (rectangle) of all geometries or rasters in a PostGIS table.

Usage

```
pgGetBoundary(conn, name, geom = "geom", clauses = NULL)
```

Arguments

conn	A connection object to a PostgreSQL database
name	A character string specifying a PostgreSQL schema and table/view name holding the geometry (e.g., `name = c("schema","table")`)
geom	character, Name of the column in `name` holding the geometry/(geography) or raster object (Default = `"geom"`)
clauses	character, additional SQL to append to modify select query from table. Must begin with an SQL clause (e.g., "WHERE ...", "ORDER BY ...", "LIMIT ..."); same usage as in `pgGetGeom`.

Value

SpatialPolygon

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
pgGetBoundary(conn, c("schema", "polys"), geom = "polygon")
pgGetBoundary(conn, c("schema", "rasters"), geom = "rast")

## End(Not run)
```

---

pgGetGeom *Load a PostGIS geometry from a PostgreSQL table/view/query into R.*

Description

Retrieve point, linestring, or polygon geometries from a PostGIS table/view/query, and convert it to an R sp object (`Spatial*` or `Spatial*DataFrame`).

Usage

```
pgGetGeom(
  conn,
  name,
  geom = "geom",
  gid = NULL,
  other.cols = TRUE,
  clauses = NULL,
```

```
  boundary = NULL,
  query = NULL
)
```

Arguments

conn	A connection object to a PostgreSQL database
name	A character string specifying a PostgreSQL schema and table/view name holding the geometry (e.g., `name = c("schema","table")`)
geom	The name of the geometry/(geography) column. (Default = "geom")
gid	Name of the column in name holding the IDs. Should be unique for each record to return. `gid=NULL` (default) automatically creates a new unique ID for each row in the sp object.
other.cols	Names of specific columns in the table to retrieve, in a character vector (e.g. `other.cols=c("col1","col2")`.) The default (`other.cols = TRUE`) is to attach all columns in a Spatial*DataFrame. Setting `other.cols=FALSE` will return a Spatial-only object (no data frame).
clauses	character, additional SQL to append to modify select query from table. Must begin with an SQL clause (e.g., "WHERE ...", "ORDER BY ...", "LIMIT ..."); see below for examples.
boundary	sp object or numeric. A Spatial* object, whose bounding box will be used to select geometries to import. Alternatively, four numbers (e.g. `c([top], [bottom], [right], [left])`) indicating the projection-specific limits with which to subset spatial data. `boundary = NULL` (default) will not subset by spatial extent. Note this is not a true 'clip'- all features intersecting the bounding box with be returned unmodified.
query	character, a full SQL query including a geometry column. For use with query mode only (see details).

Details

The query mode version of pgGetGeom allows the user to enter a complete SQL query (query) that returns a Geometry column, and save the query as a new view (name) if desired. If (name) is not specified, a temporary view with name ".rpostgis_TEMPview" is used only within the function execution. In this mode, the other arguments can be used normally to modify the Spatial* object returned from the query.

Definitions for tables written in "data frame mode" are automatically applied using this function, including proj4strings of the Spatial*-class object. Note that if the proj4string of the original dataset is not found to be equivalent to the database proj4string (using pgSRID), it will not be applied.

Value

sp-class (SpatialPoints*, SpatialMultiPoints*, SpatialLines*, or SpatialPolygons*)

Author(s)

David Bucklin <david.bucklin@gmail.com>

Mathieu Basille <basille@ufl.edu>

Examples

```
## Not run:
## Retrieve a Spatial*DataFrame with all data from table
## 'schema.tablename', with geometry in the column 'geom'
pgGetGeom(conn, c("schema", "tablename"))
## Return a Spatial*DataFrame with columns c1 & c2 as data
pgGetGeom(conn, c("schema", "tablename"), other.cols = c("c1","c2"))
## Return a Spatial*-only (no data frame),
## retaining id from table as rownames
pgGetGeom(conn, c("schema", "tablename"), gid = "table_id",
  other.cols = FALSE)
## Return a Spatial*-only (no data frame),
## retaining id from table as rownames and with a subset of the data
pgGetGeom(conn, c("schema", "roads"), geom = "roadgeom", gid = "road_ID",
    other.cols = FALSE, clauses  = "WHERE road_type = 'highway'")
## Query mode
pgGetGeom(conn, query = "SELECT r.gid as id, ST_Buffer(r.geom, 100) as geom
                           FROM
                             schema.roads r,
                             schema.adm_boundaries b
                           WHERE
                             ST_Intersects(r.geom, b.geom);")

## End(Not run)
```

---

pgGetRast *Load raster from PostGIS database.*

Description

Retrieve rasters from a PostGIS table.

Usage

```
pgGetRast(
  conn,
  name,
  rast = "rast",
  bands = 1,
  boundary = NULL,
  clauses = NULL
)
```

Arguments

conn	A connection object to a PostgreSQL database
name	A character string specifying a PostgreSQL schema and table/view name holding the geometry (e.g., `name = c("schema","table")`)

rast	Name of the column in name holding the raster object
bands	Index number(s) for the band(s) to retrieve (defaults to 1). The special case (bands = TRUE) returns all bands in the raster.
boundary	sp object or numeric. A Spatial* object, whose bounding box will be used to select the part of the raster to import. Alternatively, four numbers (e.g. c([top], [bottom], [right], [left])) indicating the projection-specific limits with which to clip the raster. boundary = NULL (default) will return the full raster.
clauses	character, optional SQL to append to modify select query from table. Must begin with 'WHERE'.

Details

Default is to return a raster-class object RasterLayer for one-band, RasterBrick for multiple bands. sp-class rasters (SpatialGrid*s or SpatialPixels*) written using pgWriteRast will attempt to re-import as the same data class.

Value

RasterLayer

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
pgGetRast(conn, c("schema", "tablename"))
pgGetRast(conn, c("schema", "DEM"), boundary = c(55,
    50, 17, 12))

## End(Not run)
```

---

pgInsert *Inserts data into a PostgreSQL table.*

Description

This function takes a take an R sp object (Spatial* or Spatial*DataFrame), or a regular data.frame, and performs the database insert (and table creation, when the table does not exist) on the database.

Usage

```
pgInsert(
  conn,
  name,
  data.obj,
  geom = "geom",
  df.mode = FALSE,
  partial.match = FALSE,
  overwrite = FALSE,
  new.id = NULL,
  row.names = FALSE,
  upsert.using = NULL,
  alter.names = FALSE,
  encoding = NULL,
  return.pgi = FALSE,
  df.geom = NULL,
  geog = FALSE
)

## S3 method for class 'pgi'
print(x, ...)
```

Arguments

conn	A connection object to a PostgreSQL database
name	A character string specifying a PostgreSQL schema and table name (e.g., `name = c("schema","table")`) If not already existing, the table will be created. If the table already exists, the function will check if all R data frame columns match database columns, and if so, do the insert. If not, the insert will be aborted. The argument `partial.match` allows for inserts with only partial matches of data frame and database column names, and `overwrite` allows for overwriting the existing database table.
data.obj	A `Spatial*` or `Spatial*DataFrame`, or `data.frame`
geom	character string. For `Spatial*` datasets, the name of geometry/(geography) column in the database table. (existing or to be created; defaults to `"geom"`). The special name "geog" will automatically set `geog` to TRUE.
df.mode	Logical; Whether to write the (Spatial) data frame in data frame mode (preserving data frame column attributes and row.names). A new table must be created with this mode (or overwrite set to TRUE), and the `row.names`, `alter.names`, and `new.id` arguments will be ignored (see `dbWriteDataFrame` for more information).
partial.match	Logical; allow insert on partial column matches between data frame and database table. If `TRUE`, columns in R data frame will be compared with the existing database table `name`. Columns in the data frame that exactly match the database table will be inserted into the database table.
overwrite	Logical; if true, a new table (`name`) will overwrite the existing table (`name`) in the database. Note: overwriting a view must be done manually (e.g., with `dbDrop`).

new.id	Character, name of a new sequential integer ID column to be added to the table for insert (for spatial objects without data frames, this column is created even if left NULL and defaults to the name "gid"). If partial.match = TRUE and the column does not exist in the database table, it will be discarded.
row.names	Whether to add the data frame row names to the database table. Column name will be '.R_rownames'.
upsert.using	Character, name of the column(s) in the database table or constraint name used to identify already-existing rows in the table, which will be updated rather than inserted. The column(s) must have a unique constraint already created in the database table (e.g., a primary key). Requires PostgreSQL 9.5+.
alter.names	Logical, whether to make database column names DB-compliant (remove special characters/capitalization). Default is FALSE. (This must be set to FALSE to match with non-standard names in an existing database table.)
encoding	Character vector of length 2, containing the from/to encodings for the data (as in the function iconv). For example, if the dataset contain certain latin characters (e.g., accent marks), and the database is in UTF-8, use encoding = c("latin1", "UTF-8"). Left NULL, no conversion will be done.
return.pgi	Whether to return a formatted list of insert parameters (i.e., a pgi object; see function details.)
df.geom	Character vector, name of a character column in an R data.frame storing PostGIS geometries, this argument can be used to insert a geometry stored as character type in a data.frame (do not use with Spatial* data types). If only the column name is used (e.g., df.geom = "geom"), the column type will be a generic (GEOMETRY); use a two-length character vector (e.g., df.geom = c("geom", "(POINT,4326)") to also specify a specific PostGIS geometry type and SRID for the column. Only recommended for for new tables/overwrites, since this method will change the existing column type.
geog	Logical; Whether to write the spatial data as a PostGIS 'GEOGRAPHY' type. By default, FALSE, unless geom = "geog".
x	A list of class pgi
...	Further arguments not used.

Details

If new.id is specified, a new sequential integer field is added to the data frame for insert. For Spatial*-only objects (no data frame), a new ID column is created by default with name "gid".

If the R package wkb is installed, this function will use writeWKB for certain datasets (non-Multi types, non-Linestring), which is faster for large datasets. In all other cases the rgeos function writeWKT is used.

In the event of function or database error, the database uses ROLLBACK to revert to the previous state.

If the user specifies return.pgi = TRUE, and data preparation is successful, the function will return a pgi object (see next paragraph), regardless of whether the insert was successful or not. This object can be useful for debugging, or re-used as the data.obj in pgInsert; (e.g., when data preparation is slow, and the exact same data needs to be inserted into tables in two separate tables or databases).

If return.pgi = FALSE (default), the function will return TRUE for successful insert and FALSE for failed inserts.

Use this function with codedf.mode = TRUE to save data frames from Spatial*-class objects to the database in "data frame mode". Along with normal dbwriteDataFrame operation, the proj4string of the spatial data will also be saved, and re-attached to the data when using pgGetGeom to import the data. Note that other attributes of Spatial* objects are **not** saved (e.g., coords.nrs, which is used to specify the column index of x/y columns in SpatialPoints*).

pgi objects are a list containing four character strings: (1) in.table, the table name which will be created or inserted into (2) db.new.table, the SQL statement to create the new table, (3) db.cols.insert, a character string of the database column names to insert into, and (4) insert.data, a character string of the data to insert.

Value

Returns TRUE if the insertion was successful, FALSE if failed, or a pgi object if specified.

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
library(sp)
data(meuse)
coords <- SpatialPoints(meuse[, c("x", "y")])
spdf <- SpatialPointsDataFrame(coords, meuse)

## Insert data in new database table
pgInsert(conn, name = c("public", "meuse_data"), data.obj = spdf)

## The same command will insert into already created table (if all R
## columns match)
pgInsert(conn, name = c("public", "meuse_data"), data.obj = spdf)

## If not all database columns match, need to use partial.match = TRUE,
## where non-matching columns are not inserted
colnames(spdf@data)[4] <- "cu"
pgInsert(conn, name = c("public", "meuse_data"), data.obj = spdf,
    partial.match = TRUE)

## End(Not run)
```

---

pgListGeom *List geometries/rasters*

Description

List all geometry/(geography) or raster columns available in a PostGIS database.

Usage

```
pgListGeom(conn, geog = TRUE)

pgListRast(conn)
```

Arguments

conn	A PostgreSQL database connection.
geog	Logical. For pgListGeom, whether to include PostGIS geography-type columns stored in the database

Value

If `exec = TRUE`, a data frame with schema, table, geometry/(geography) or raster (for pgListRast) column, and geometry/(geography) type.

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
pgListGeom(conn)

pgListRast(conn)

## End(Not run)
```

---

pgMakePts *Add a POINT or LINESTRING geometry field.*

Description

Add a new POINT or LINESTRING geometry field.

Usage

```
pgMakePts(
  conn,
  name,
  colname = "geom",
  x = "x",
  y = "y",
  srid,
  index = TRUE,
  display = TRUE,
```

```
  exec = TRUE
)

pgMakeStp(
  conn,
  name,
  colname = "geom",
  x = "x",
  y = "y",
  dx = "dx",
  dy = "dy",
  srid,
  index = TRUE,
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object.
name	A character string specifying a PostgreSQL schema and table name (e.g., name = c("schema","table"))
colname	A character string specifying the name of the new geometry column.
x	The name of the x/longitude field.
y	The name of the y/latitude field.
srid	A valid SRID for the new geometry.
index	Logical. Whether to create an index on the new geometry.
display	Logical. Whether to display the query (defaults to TRUE).
exec	Logical. Whether to execute the query (defaults to TRUE).
dx	The name of the dx field (i.e. increment in x direction).
dy	The name of the dy field (i.e. increment in y direction).

Value

If exec = TRUE, returns TRUE if the geometry field was successfully created.

Author(s)

Mathieu Basille <basille@ufl.edu>

See Also

The PostGIS documentation for ST_MakePoint: http://postgis.net/docs/ST_MakePoint.html, and for ST_MakeLine: http://postgis.net/docs/ST_MakeLine.html, which are the main functions of the call.

Examples

```
## Examples use a dummy connection from DBI package
conn <- DBI::ANSI()

## Create a new POINT field called 'pts_geom'
pgMakePts(conn, name = c("schema", "table"), colname = "pts_geom",
    x = "longitude", y = "latitude", srid = 4326, exec = FALSE)

## Create a new LINESTRING field called 'stp_geom'
pgMakeStp(conn, name = c("schema", "table"), colname = "stp_geom",
    x = "longitude", y = "latitude", dx = "xdiff", dy = "ydiff",
    srid = 4326, exec = FALSE)
```

---

pgPostGIS *Check and create PostGIS extension.*

Description

The function checks for the availability of the PostGIS extension, and if it is available, but not installed, install it. Additionally, can also install Topology, Tiger Geocoder and SFCGAL extensions.

Usage

```
pgPostGIS(
  conn,
  topology = FALSE,
  tiger = FALSE,
  sfcgal = FALSE,
  display = TRUE,
  exec = TRUE
)
```

Arguments

conn	A connection object (required, even if `exec = FALSE`).
topology	Logical. Whether to check/install the Topology extension.
tiger	Logical. Whether to check/install the Tiger Geocoder extension. Will also install extensions "fuzzystrmatch", "address_standardizer", and "address_standardizer_data_us" if all are available.
sfcgal	Logical. Whether to check/install the SFCGAL extension.
display	Logical. Whether to display the query (defaults to `TRUE`).
exec	Logical. Whether to execute the query (defaults to `TRUE`).

Value

`TRUE` if PostGIS is installed.

Author(s)

Mathieu Basille <basille@ufl.edu>

Examples

```
## 'exec = FALSE' does not install any extension, but nevertheless
## check for available and installed extensions:
## Not run:
    pgPostGIS(con, topology = TRUE, tiger = TRUE, sfcgal = TRUE,
        exec = FALSE)

## End(Not run)
```

---

pgSRID *Find (or create) PostGIS SRID based on CRS object.*

Description

This function takes CRS-class object and a PostgreSQL database connection (with PostGIS extension), and returns the matching SRID(s) for that CRS. If a match is not found, a new entry can be created in the PostgreSQL `spatial_ref_sys` table using the parameters specified by the CRS. New entries will be created with `auth_name = 'rpostgis_custom'`, with the default value being the next open value between 880001-889999 (a different SRID value can be entered if desired.)

Usage

```
pgSRID(conn, crs, create.srid = FALSE, new.srid = NULL)
```

Arguments

`conn`	A connection object to a PostgreSQL database.
`crs`	CRS object, created through a call to CRS.
`create.srid`	Logical. If no matching SRID is found, should a new SRID be created? User must have write access on `spatial_ref_sys` table.
`new.srid`	Integer. Optional SRID to give to a newly created SRID. If left NULL (default), the next open value of `srid` in `spatial_ref_sys` between 880001 and 889999 will be used.

Value

SRID code (integer).

Author(s)

David Bucklin <david.bucklin@gmail.com>

Examples

```
## Not run:
drv <- dbDriver("PostgreSQL")
conn <- dbConnect(drv, dbname = "dbname", host = "host", port = "5432",
    user = "user", password = "password")
(crs <- CRS("+proj=longlat"))
pgSRID(conn, crs)
(crs2 <- CRS(paste("+proj=stere", "+lat_0=52.15616055555555 +lon_0=5.38763888888889",
    "+k=0.999908 +x_0=155000 +y_0=463000", "+ellps=bessel",
    "+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812",
    "+units=m")))
pgSRID(conn, crs2, create.srid = TRUE)

## End(Not run)
```

`pgWriteRast` *Write raster to PostGIS database table.*

Description

Sends R raster to a PostGIS database table.

Usage

```
pgWriteRast(
  conn,
  name,
  raster,
  bit.depth = NULL,
  blocks = NULL,
  constraints = TRUE,
  overwrite = FALSE,
  append = FALSE
)
```

Arguments

`conn`	A connection object to a PostgreSQL database
`name`	A character string specifying a PostgreSQL schema (if necessary) and table name to hold the raster (e.g., `name = c("schema","table")`)
`raster`	An R `RasterLayer`, `RasterBrick`, or `RasterStack` from raster package; a `SpatialGrid*` or `SpatialPixels*` from sp package
`bit.depth`	The bit depth of the raster. Will be set to 32-bit (unsigned int, signed int, or float, depending on the data) if left null, but can be specified (as character) as one of the PostGIS pixel types (see http://postgis.net/docs/RT_ST_BandPixelType.html)

blocks	Optional desired number of blocks (tiles) to split the raster into in the resulting PostGIS table. This should be specified as a one or two-length (columns, rows) integer vector.
constraints	Whether to create constraints from raster data. Recommended to leave TRUE unless applying constraints manually (see http://postgis.net/docs/RT_AddRasterConstraints.html). Note that constraint notices may print to the console, depending on the PostgreSQL server settings.
overwrite	Whether to overwrite the existing table (name).
append	Whether to append to the existing table (name).

Details

RasterLayer names will be stored in an array in the column "band_names", which will be restored in R when imported with the function pgGetRast.

Rasters from the sp package are converted to raster package objects prior to insert.

If blocks = NULL the attempted block size will be around 10,000 pixels in size (100 x 100 cells), so number of blocks will vary by raster size. If a specified number of blocks is desired, set blocks to a one or two-length integer vector. Note that fewer, larger blocks generally results in faster write times.

Value

TRUE for successful import.

Author(s)

David Bucklin <david.bucklin@gmail.com>

See Also

Function follows process from http://postgis.net/docs/using_raster_dataman.html#RT_Creating_Rasters.

Examples

```
## Not run:
pgWriteRast(conn, c("schema", "tablename"), raster_name)

# basic test
r <- raster::raster(nrows=180, ncols=360, xmn=-180, xmx=180,
    ymn=-90, ymx=90, vals=1)
pgWriteRast(conn, c("schema", "test"), raster = r,
    bit.depth = "2BUI", overwrite = TRUE)

## End(Not run)
```

rpostgis *R interface to a PostGIS database.*

Description

'rpostgis' provides an interface between R and 'PostGIS'-enabled 'PostgreSQL' databases to transparently transfer spatial data. Both vector (points, lines, polygons) and raster data are supported in read and write modes. Also provides convenience functions to execute common procedures in 'PostgreSQL/PostGIS'. For a list of documented functions, use `library(help = "rpostgis")`.

Details

A typical session starts by establishing the connection to a working PostgreSQL database:

library(rpostgis) con <- dbConnect("PostgreSQL", dbname = <dbname>, host = <host>, user = <user>, password = <password>)

For example, this could be:

con <- dbConnect("PostgreSQL", dbname = "rpostgis", host = "localhost", user = "postgres", password = "postgres")

The next step typically involves checking if PostGIS was installed on the working database, and if not try to install it:

pgPostGIS(con)

The function should return `TRUE` for all `pg-` functions to work.

Finally, at the end of an interactive session, the connection to the database should be closed:

dbDisconnect(con)

Author(s)

Mathieu Basille (<basille@ufl.edu>) and David Bucklin (<david.bucklin@gmail.com>)

Index