Description of the pgUtils package

Johannes Rainer*

March 22, 2011

Division Molecular Pathophysiology
Biocenter, Medical University Innsbruck
Fritz-Pregl Strasse 3
6020 Innsbruck, Austria, http://bioinfo.i-med.ac.at
and
Tyrolean Cancer Research Institute
Innrain 66, 6020 Innsbruck, Austria, http://www.tcri.at

Contents

1 Introduction

The package *pgUtils* depends on the *RPostgreSQL* package and provides some utility functions for databases in special for a PostgreSQL database. The package provides functions for creating database tables with autoincrementing primary keys and the posibility to referencing to other tables using foreign keys to allow a referential integrity of the data. Other functions can be used to insert or update the information in database tables. All functions run within transactions, so if any error occurs during a database call, the original state of the database before the call can be restored. Additionally the packages provides a simple logging mechanism, which writes log messages to a file (which can be specified).

*johannes.rainer@i-med.ac.at

2 Creating a database table

First of all a connection to the database has to be established. This can be done using the `dbConnect`. The code below connects to the database *template1* which is installed by default in every PostgreSQL database. As we do not want to store any information into this datbase we create a new database by sending the appropriate SQL call and connect then to this new database.

```
> library(pgUtils)
> con.su <- dbConnect(PostgreSQL(), host = "localhost", user = "postgres",
+     dbname = "template1")
> if (sum(dbListDatabases(con.su)[, "datname"] == "pgutils") >
+     0) {
+     dbSendQuery(con.su, "DROP DATABASE pgutils")
+ }
> res <- dbSendQuery(con.su, "CREATE DATABASE pgutils")
> dbDisconnect(con.su)
> con <- dbConnect(PostgreSQL(), host = "localhost", user = "postgres",
+     dbname = "pgutils")
> log.file <- "test.log"
```

The first database table that we will create is a simple table with 3 columns, two of them contain numbers and one character strings. The `log.file <- "test.log"` specifies the name for the log file. By default this file is called *pgUtils.log* and resides in the current working directory.

```
> createDBTable(con, name = "firsttest", attributes = c("a", "b",
+     "c"), data.types = c("TEXT", "REAL", "REAL"))
> dbColnames(con, "firsttest")

character(0)
```

This call created a table called *firsttest* with three columns. The function automatically created a additional column with the name `firsttest_pk` that is used as primary key column and which is automatically incremented upon data insertion. The `data.types` attribute allows to specify the data types for the columns (if not submitted all data types will automatically set to *TEXT*). To insert data into the database the function `insertIntoTable` can be used.

```
> MyTable <- data.frame(a = c("some", "text"), b = c(2, 3), c = c(1.3,
+     3.5))
> MyTable

     a b   c
1 some 2 1.3
2 text 3 3.5

> insertIntoTable(con, name = "firsttest", data = MyTable)
```

To read the data from the database the `dbGetQuery` function from the *RpostgreSQL* package can be used.

```
> result <- dbGetQuery(con, "SELECT * FROM firsttest")
```

3 Handling table references

Relational databases allow to combine informations between database tables and to concatenate the informations. In the next example we will also create two database tables that are related to each other (to be exact we create a *1 to n* relation, that means that n entries of the one table are related (belong) to one entry in the other table).

```
> createDBTable(con, name = "species", attributes = c("speciesname",
+     "value"), data.types = c("TEXT"))
> insertIntoTable(con, name = "species", data = data.frame(speciesname = c("hobbit",
+     "human", "orc"), value = c("good", "bad", "very bad")))
> createDBTable(con, name = "individual", attributes = c("name",
+     "age"), data.types = c("TEXT"), references = "species")
> ref <- new("AutoReference", source.table = "individual", ref.table = "species",
+     source.table.column = "speciesref", ref.table.column = "speciesname")
> insertIntoTable(con, name = "individual", data = data.frame(name = c("frodo",
+     "fred", "bilbo"), speciesref = c("hobbit", "human", "hobbit"),
+     age = c(23, 32, 111)), references = ref)
```

The main points about the calls above are, that it is important to define which table relates to which and which column of the submitted data table can be used to establish the relation between the tables (in the example above this is done with the `AutoReference` object and the column *speciesref* in the second data table). The function `insertIntoTable` uses this information to insert the primary keys of the entry to which the rows of the second table link into theyr foreign key field. The database table *species* has the following data stored

```
> dbGetQuery(con, "SELECT * FROM species")

  species_pk speciesname    value
1          1      hobbit     good
2          2       human      bad
3          3         orc very bad
```

and the table *individual*

```
> dbGetQuery(con, "SELECT * FROM individual")

  individual_pk species_fk  name age
1             1          1 frodo  23
2             2          2  fred  32
3             3          1 bilbo 111
```

Both tables have a primary key column (usually the name of the table ending in _pk) and the *individual* table has also a foreign key column (usually the name of the referenced table ending in _fk) that is used to link to the *species* table. So n rows of the *individual* table relate to one entry in the *species* table.

The information between the two tables can now be joined like

```
> dbGetQuery(con, "SELECT * FROM individual JOIN species ON (species_fk=species_pk) WHERE speciesname='hobbit'")

  individual_pk species_fk  name age species_pk speciesname value
1             1          1 frodo  23          1      hobbit  good
2             3          1 bilbo 111          1      hobbit  good
```

Another nice feature of this foreign keys concept is, that it is not possible to delete entries from the database that are related by another entry. In our case it means that we cannot delete the *hobbit* entry as long as there are *hobbits* in our *individual* table.

In this way the referential integrity will be mantained and the data in the database will not get inconsistent.