# Spatio-temporal objects to proxy a PostgreSQL table

ifgi
Institute for Geoinformatics
University of Münster

Edzer Pebesma

July 15, 2011

**Abstract**

This vignette describes and implements a class that proxies data sets in a PostgreSQL database with classes in the spacetime package. This might allow access to data sets too large to fit into R memory.

## Contents



## 1 Introduction

Massive data are difficult to analyze with R, because R objects reside in memory. Spatio-temporal data easily become massive, either because the spatial domain contains a lot of information (satellite imagery), or many time steps are available (high resolution sensor data), or both. This vignette shows how data residing in a data base can be read into R using spatial or temporal selection.

In case the commands are not evaluated because CRAN packages cannot access an external data base, a document with evaluated commands is found here.

This vignette was run in the following session:

```
> library(RPostgreSQL)
> library(spacetime)
> sessionInfo()
```

```
R version 2.13.1 (2011-07-08)
Platform: i486-pc-linux-gnu (32-bit)

locale:
 [1] LC_CTYPE=en_US.utf8       LC_NUMERIC=C
 [3] LC_TIME=en_US.utf8        LC_COLLATE=C
 [5] LC_MONETARY=C             LC_MESSAGES=en_US.utf8
 [7] LC_PAPER=en_US.utf8       LC_NAME=C
 [9] LC_ADDRESS=C              LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.utf8 LC_IDENTIFICATION=C

attached base packages:
[1] stats4    stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] RPostgreSQL_0.1-7 DBI_0.2-5          rgeos_0.1-8        stringr_0.4
 [5] cshapes_0.2-8     rJava_0.8-8        adehabitatLT_0.2   CircStats_0.2-4
 [9] boot_1.3-2        adehabitatMA_0.2   ade4_1.4-17        gpclib_1.5-1
[13] trip_1.1-9        diveMove_1.2.5     caTools_1.11       bitops_1.0-4.1
[17] RColorBrewer_1.0-2 rgdal_0.7-2       mapdata_2.1-3      gstat_1.0-0
[21] plm_1.2-7         sandwich_2.2-6     MASS_7.3-13        Formula_1.0-0
[25] nlme_3.1-101      bdsmatrix_1.0      maps_2.1-5         maptools_0.8-6
[29] lattice_0.19-30   spacetime_0.5-1    xts_0.8-0          zoo_1.6-5
[33] sp_0.9-84         foreign_0.8-44

loaded via a namespace (and not attached):
[1] grid_2.13.1  plyr_1.5.1   tools_2.13.1
```

## 2 Setting up a database

We will first set the characteristics of the database[1]

```
> dbname = "postgis"
> user = "user"
> password = "password"
```

Next, we will create a driver and connect to the database:

```
> drv <- dbDriver("PostgreSQL")
> con <- dbConnect(drv, dbname = dbname, user = user, password = password)
```

It should be noted that these first two commands are specific to PostgreSQL; from here on, commands are generic and should work for any database connector that uses the interface of package DBI.

We now remove a set of tables (if present) so they can be created later on:

[1]It is assumed that the database is *spatially enabled*, i.e. it understands how simple features are stored. The standard for this from the open geospatial consortium is described here.

```
> dbRemoveTable(con, "rural_attr")
> dbRemoveTable(con, "rural_space")
> dbRemoveTable(con, "rural_time")
> dbRemoveTable(con, "space_select")
```

Now we will create the table with spatial features (observation locations). For this, we need the `rgdal` function `writeOGR`, which by default creates an index on the geometry:

```
> data(air)
> rural = as(rural, "STSDF")
> p = rural@sp
> sp = SpatialPointsDataFrame(p, data.frame(geom_id = 1:length(p)))
> library(rgdal)
> OGRstring = paste("PG:dbname=", dbname, " user=", user, " password=",
+     password, sep = "")
> writeOGR(sp, OGRstring, "rural_space", driver = "PostgreSQL")
```

Second, we will write the table with times to the database, and create an index to time:

```
> df = data.frame(time = index(rural@time), time_id = 1:nrow(rural@time))
> dbWriteTable(con, "rural_time", df)
> idx = "create index time_idx on rural_time (time);"
> dbSendQuery(con, idx)
```

Finally, we will write the full attribute data table to PosgreSQL, along with its indexes to the spatial and temporal tables:

```
> idx = rural@index
> names(rural@data) = "pm10"
> df = cbind(data.frame(geom_id = idx[, 1], time_id = idx[, 2]),
+     rural@data)
> dbWriteTable(con, "rural_attr", df)
```

# 3 A proxy class

The following class has as components a spatial and temporal data structure, but no spatio-temporal attributes (they are assumed to be the most memory-hungry). The other slots refer to the according tables in the PostGIS database, the name(s) of the attributes in the attribute table, and the database connection.

```
> setClass("ST_PG", representation("ST", space_table = "character",
+     time_table = "character", attr_table = "character", attr = "character",
+     con = "PostgreSQLConnection"))

[1] "ST_PG"
```

Next, we will create an instance of the new class:

```
> rural_proxy = new("ST_PG", ST(rural@sp, rural@time), space_table = "rural_space",
+     time_table = "rural_time", attr_table = "rural_attr", attr = "pm10",
+     con = con)
```

# 4 Selection based on time period and/or region

The following two helper functions create a character string with an SQL command that for a temporal or spatial selection:

```
> .SqlTime = function(x, j) {
+     stopifnot(is.character(j))
+     t = .parseISO8601(j)
+     t1 = paste("'", t$first.time, "'", sep = "")
+     t2 = paste("'", t$last.time, "'", sep = "")
+     what = paste("geom_id, time_id", paste(x@attr, collapse = ","),
+         sep = ", ")
+     paste("SELECT", what, "FROM", x@attr_table, "AS a JOIN",
+         x@time_table, "AS b USING (time_id) WHERE b.time >= ",
+         t1, "AND b.time <=", t2, ";")
+ }
> .SqlSpace = function(x, i) {
+     stopifnot(is(i, "Spatial"))
+     writeOGR(i, OGRstring, "space_select", driver = "PostgreSQL")
+     what = paste("geom_id, time_id", paste(x@attr, collapse = ","),
+         sep = ", ")
+     paste("SELECT", what, "FROM", x@attr_table, "AS a JOIN (SELECT p.wkb_geometry, p.geo
+         x@space_table, " AS p, space_select AS q", "WHERE ST_Intersects(p.wkb_geometry,
+         "AS b USING (geom_id);")
+ }
```

The following selection method selects a time period only, as defined by the methods in package `xts`. A time period is defined as a valid ISO8601 string, e.g. 2005-05 is the full month of May for 2005.

```
> setMethod("[", "ST_PG", function(x, i, j, ..., drop = TRUE) {
+     stopifnot(missing(i) != missing(j))
+     if (missing(j))
+         sql = .SqlSpace(x, i)
+     else sql = .SqlTime(x, j)
+     print(sql)
+     df = dbGetQuery(x@con, sql)
+     STSDF(x@sp, x@time, df[x@attr], as.matrix(df[c("geom_id",
+         "time_id")]))
+ })

[1] "["

> pm10_20050101 = rural_proxy[, "2005-01-01"]
> summary(pm10_20050101)
> summary(rural[, "2005-01-01"])
> pm10_NRW = rural_proxy[DE_NUTS1[10, ], ]
> summary(pm10_NRW)
> summary(rural[DE_NUTS1[10, ], ])
```

Clearly, the temporal and spatial components are not subsetted, so do not reflect the actual selection made; the attribute data however do; the following selection step "cleans" the unused features/times:

```
> dim(pm10_NRW)
> pm10_NRW = pm10_NRW[T, ]
> dim(pm10_NRW)
```

Comparing sizes, we see that the selected object is smaller:

```
> object.size(rural)
> object.size(pm10_20050101)
> object.size(pm10_NRW)
```

# 5 Closing the database connection

The following commands close the database connection and release the driver resources:

```
> dbDisconnect(con)
> dbUnloadDriver(drv)
```

# 6 Limitations

The example code in this vignette is not meant as a full-fledged database access mechanism for spatio-temporal data. In particular, the selection here can do only *one* of spatial locations (entered as features) or time periods. If database access is only based on time, a spatially enabled database (such as PostGIS) would not be needed.

For massive databases, data would typically not be loaded into the database from R first, but from somewhere else.