Spatio-temporal objects to proxy a PostgreSQL table

Edzer Pebesma

July 17, 2018

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 1
2 Setting up a database 2
3 A proxy class 3
4 Selection based on time period and/or region 3
5 Closing the database connection 5
6 Limitations and alternatives 5

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 using the following libraries:

R> library(RPostgreSQL)
We will first set the characteristics of the database.

```r
R> dbname = "postgis"
R> user = "edzer"
R> password = "pw"
R> #password = ""
```

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

```r
R> drv <- dbDriver("PostgreSQL")
R> con <- dbConnect(drv, dbname=dbname, user=user, password=password,
+ host='localhost', port='5432')
```

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:

```r
R> dbRemoveTable(con, "rural_attr")
R> dbRemoveTable(con, "rural_space")
R> dbRemoveTable(con, "rural_time")
R> 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:

```r
R> data(air)
R> rural = STFDF(stations, dates, data.frame(PM10 = as.vector(air)))
R> rural = as(rural, "STSDF")
R> p = rural@sp
R> sp = SpatialPointsDataFrame(p, data.frame(geom_id=1:length(p)))
R> library(rgdal)
R> OGRstring = paste("PG:dbname=", dbname, " user=", user,
+ password=", password, " host=localhost", sep = "")
R> print(OGRstring)
R> writeOGR(sp, OGRstring, "rural_space", driver = "PostgreSQL")
```

In case you have problems replicating this, verify that your rgdal installation provides the PostgreSQL driver, e.g. by checking that

```r
R> subset(ogrDrivers(), name == "PostgreSQL")$write
```

prints a TRUE, and not a logical(0).

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

\footnote{It is assumed that the database is \textit{spatially enabled}, i.e. it understands how simple features are stored. The standard for this from the open geospatial consortium is described \url{here}.}
Finally, we will write the full attribute data table to PostgreSQL, along with its indexes to the spatial and temporal tables:

```
R> idx = rural@index
R> names(rural@data) = "pm10" # lower case
R> df = cbind(data.frame(geom_id = idx[,1], time_id = idx[,2]), rural@data)
R> 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.

```
R> setClass("ST_PG", contains = "ST",
+ # slots = c(space_table = "character",
+ representation(space_table = "character",
+ time_table = "character",
+ attr_table = "character",
+ attr = "character",
+ con = "PostgreSQLConnection"))
```

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

```
R> rural_proxy = new("ST_PG",
+ #ST(rural@sp, rural@time, rural@endTime),
+ as(rural, "ST"),
+ 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:

```
R> .SqlTime = function(x, j) {
+   stopifnot(is.character(j))
+   require(xts)
+   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 = ",")
+   what
`
The following selection method selects a time period only, as defined by the methods in package \texttt{xts}. A time period is defined as a valid ISO8601 string, e.g. 2005-05 is the full month of May for 2005.

\begin{verbatim}
R> pm10_20050101 = rural_proxy[, "2005-01-01"]
R> summary(pm10_20050101)
R> summary(rural[, "2005-01-01"])
R> pm10_NRW = rural_proxy[DE_NUTS1[10,],]
R> summary(pm10_NRW)
R> summary(rural[DE_NUTS1[10,],])
\end{verbatim}

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:

\begin{verbatim}
R> pm10_20050101 = pm10_20050101[T,]
R> dim(pm10_NRW)
R> pm10_NRW = pm10_NRW[T,]
R> dim(pm10_NRW)
\end{verbatim}

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

\begin{verbatim}
R> object.size(rural)
R> object.size(pm10_20050101)
R> object.size(pm10_NRW)
\end{verbatim}
5 Closing the database connection

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

\begin{verbatim}
R> dbDisconnect(con)
R> dbUnloadDriver(drv)
\end{verbatim}

6 Limitations and alternatives

The example code in this vignette is meant as an example and is not meant as a full-fledged database access mechanism for spatio-temporal data bases. 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.

An alternative to access from R large, possibly massive spatio-temporal data bases for the case where the data base is accessible through a sensor observation service (SOS) is provided by the R package sos4R, which is also on CRAN.