**PostGIS extends capabilities of PostgreSQL database to deal with spatial data. Using PostGIS, your database supports geographic queries to be run directly in SQL. In this blog post, we will connect and interact with a PostGIS database from R, using {DBI} and {sf}.**

**Package {sf} and PostGIS are friends**



Package {sf} is similar to PostGIS database in multiple ways:

* R package {sf} was created to supports simple features (thus, named “sf”). “Simple features” is an ISO standard defined by the Open Geospatial Consortium (OGC) to standardize storage and access to geographical datasets. PostGIS supports objects and functions specified in the OGC “Simple Features for SQL” specification.
* Most {sf} functions starts with st\_ to follow the denomination of functions in PostGIS. For instance, function sf::st\_union has its equivalent ST\_Union in Postgis. If you were wondering why, now you know !
* Spatial data with {sf} are manipulated in a SQL way. Simple features in {sf} allows to manipulate vector spatial objects using {dplyr}, which syntax is similar to SQL functions names. For instance, dplyr::select, which can be applied on {sf} objects, has its equivalent SELECT in SQL syntax.

**Configuration of your session**

**Packages**

Main packages used in this post are {DBI}, {RPostgres}, {sqlpetr} and {sf}. If you want to properly install spatial packages on Ubuntu Server, you may want to refer to our post: [installation of R on ubuntu and tips for spatial packages](https://rtask.thinkr.fr/blog/installation-of-r-3-5-on-ubuntu-18-04-lts-and-tips-for-spatial-packages/).

library(DBI)

library(viridis)

library(RPostgres)

library(sqlpetr)

library(sf)

library(rpostgis)

library(dplyr)

library(dbplyr)

library(rnaturalearth)

library(ggplot2)

# ThinkR internal package for color palette and other brand identity

library(thinkridentity)

**Verification of GDAL drivers**

To be able to connect PostGIS database in R, you will need a GDAL version that supports PostGIS driver. You can verify this using sf::st\_drivers().

# GDAL drivers

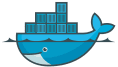
st\_drivers() %>%

filter(grepl("Post", name))

## name long\_name write copy is\_raster is\_vector vsi

## 1 PostgreSQL PostgreSQL/PostGIS TRUE FALSE FALSE TRUE FALSE

**Build a Docker with Postgres and PostGIS enabled**

For reproducibility of this blog post, it is thus natural to use Docker. We build a container with a Postgres database having PostGIS enabled.

* Postgis on DockerHub: <https://hub.docker.com/r/mdillon/postgis>
* Credentials: superuser: “postgres”, pwd: “mysecretpassword”

*To use this Docker container in production on your server, do not forget to set a persistant drive.*

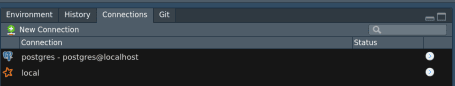
docker\_cmd <- "run --detach --name some-postgis --publish 5432:5432 --env POSTGRES\_PASSWORD=mysecretpassword -d mdillon/postgis"

system2("docker", docker\_cmd, stdout = TRUE, stderr = TRUE)

## [1] "368b4733bdd333ee9c4501b44c3e6738f2cf313730fb18db8582202ccb42adf3"

**Connection to the PostGIS database from R**

You can connect to any database using {DBI} package, provided that you have the correct drivers installed. If you use Rstudio as your IDE to develop in R, I would recommend using package {sqlpetr} to connect to your Postgres database. This package has a “connexion contract” allowing to explore your database directly in the Rstudio Connection Pane.



# With sqlpetr

con <- sp\_get\_postgres\_connection(

host = "localhost",

dbname = "postgres",

port = 5432,

user = "postgres",

password = "mysecretpassword",

seconds\_to\_test = 30,

connection\_tab = TRUE

)

**Write and read World data in PostGIS**

In the following examples, we will play with a World map from package {rnaturalearth}.  
*Equal Earth projection is used for maps representation here.*

ne\_world <- rnaturalearth::ne\_countries(scale = 50, returnclass = "sf")

# Choose one available to you

world\_map\_crs <- "+proj=eqearth +wktext"

# Use your custom colours

ne\_world %>%

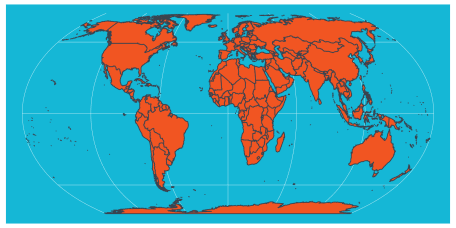
st\_transform(world\_map\_crs) %>%

ggplot() +

geom\_sf(fill = thinkr\_cols("warning"), colour = thinkr\_cols("dark")) +

theme(panel.background = element\_rect(fill = thinkr\_cols("primary")))

)



We can write the data in our PostGIS database directly using sf::st\_write. Default parameters overwrite = FALSE, append = FALSE prevent overwriting your data if already in the database. {sf} will guess the writing method in the database.

st\_write(ne\_world, dsn = con, layer = "ne\_world",

overwrite = FALSE, append = FALSE)

Then, we can read data from the database using sf::st\_read.

world\_sf <- st\_read(con, layer = "ne\_world")

**Use SQL queries before loading the spatial data**

When using st\_read, the dataset is completely loaded in the memory of your computer. If your spatial dataset is big, you may want to query only a part of it. Classical SQL queries can be used, for instance to extract Africa only, using st\_read with parameter query.

query <- paste(

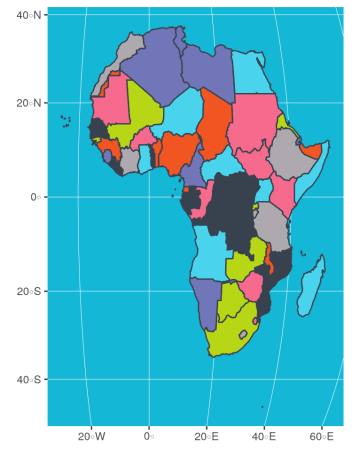
'SELECT "name", "name\_long", "geometry"',

'FROM "ne\_world"',

'WHERE ("continent" = \'Africa\');'

)

africa\_sf <- st\_read(con, query = query)



**Tips: Do you need help for your SQL query ?**

PostGIS allows you to make SQL queries with geomatics functions that you can also define before reading your dataset with {sf}. However, if you are not totally comfortable with SQL but you are a ninja with {dplyr}, then you will be fine. By using dplyr::show\_query, you can get the SQL syntax of your {sf} operations. Let’s take an example.  
With {sf}, union of countries by continent could be written as follows:

world\_sf %>%

group\_by(continent) %>%

summarise()

Indeed, the complete {sf} code would be:

world\_sf %>%

group\_by(continent) %>%

summarise(geometry = st\_union(geometry))

Now, let us connect to the Postgres database with {dplyr}. When connected to a database with {dplyr}, queries are executed directly in SQL in the database, not in the computer memory.  
Then use show\_query to get the translation in SQL proposed by {dplyr}.

# Read with dplyr directly in database

world\_tbl <- tbl(con, "ne\_world")

# Create query

world\_tbl %>%

group\_by(continent) %>%

summarise(

geometry = st\_union(geometry)) %>%

show\_query()

##

## SELECT "continent", ST\_UNION("geometry") AS "geometry"

## FROM "ne\_world"

## GROUP BY "continent"

{dplyr} has not been tailored to translate spatial operations in SQL, but as you can see below, difference with the correct query is very small. Only the PostGIS function is not correctly written (ST\_Union).

query <- paste(

'SELECT "continent", ST\_Union(geometry) AS "geometry"',

'FROM "ne\_world"',

'GROUP BY "continent";'

)

union\_sf <- st\_read(con, query = query, quiet = TRUE)

# Plot (Choose your own palette)

union\_sf %>%

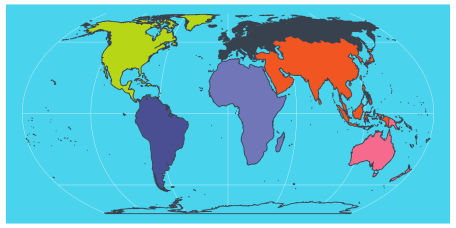
st\_transform(world\_map\_crs) %>%

ggplot() +

geom\_sf(aes(fill = continent), colour = thinkr\_cols("dark")) +

scale\_fill\_thinkr\_d(palette = "discrete2\_long", guide = FALSE) +

theme(panel.background = element\_rect(fill = thinkr\_cols("primary\_light")))



**Be careful. You can not trust operations with {dplyr}/{dbplyr} on a spatial database because the ‘geometry’ column is not imported as spatial column with tbl() and you would loose it. However, you can use {dplyr} syntax to explore your dataset and all columns out of the spatial information. All operations will be realised inside the database, which assures not to overload the memory.**

**What about rasters ?**

If you follow this blog, you know that I like rasters. Package {sf} is only dealing with spatial vector objects. It can not manage rasters in a PostGIS database. To do so, you can have a look at package {rpostgis} with functions pgGetRast() and pgWriteRast(). This package works with Rasters from package {raster}. You will need to connect using package {RPostgreSQL}, which does not have a Rstudio “connexion contract”. *Note that you can have two connections to your database, one with {sqlpetr} to keep the Rstudio Connection pane, and the other with {RPostgreSQL} to connect with Raster objects.*

* Connect to the database with {RPostgreSQL}

library(rpostgis)

con\_raster <- RPostgreSQL::dbConnect("PostgreSQL",

host = "localhost", dbname = "postgres",

user = "postgres", password = "mysecretpassword", port = 5432

)

# check if the database has PostGIS

pgPostGIS(con\_raster)

## [1] TRUE

* Create an empty raster and save it in the database

# From documentation basic test

r <- raster::raster(

nrows = 180, ncols = 360, xmn = -180, xmx = 180,

ymn = -90, ymx = 90, vals = 1

)

# Write Raster in the database

pgWriteRast(con\_raster,

name = "test", raster = r, overwrite = TRUE

)

## [1] TRUE

* Read the raster from the database

r\_read <- pgGetRast(con\_raster, name = "test")

print(r\_read)

## class : RasterLayer

## dimensions : 180, 360, 64800 (nrow, ncol, ncell)

## resolution : 1, 1 (x, y)

## extent : -180, 180, -90, 90 (xmin, xmax, ymin, ymax)

## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0

## data source : in memory

## names : layer

## values : 1, 1 (min, max)

* Do not forget to close the database connection after your operations

# Disconnect database

RPostgreSQL::dbDisconnect(con\_raster)

## [1] TRUE

**Disconnect database and close Docker container**

* Disconnect database connection made with {sqlpetr}
* Stop and remove Docker container if wanted

# Disconnect database

DBI::dbDisconnect(con)

# Stop Docker container

system2("docker", "stop some-postgis")

system2("docker", "rm --force some-postgis")

# Verify docker

system2("docker", "ps -a")

**Packages used in this blogpost**

## [1] "attachment" "DBI" "dbplyr" "dplyr" "ggplot2" "knitr" "raster"

## [8] "rnaturalearth" "rpostgis" "RPostgres" "RPostgreSQL" "sf" "sqlpetr" "thinkridentity"