

Certificat Data Scientist

Structured Query Language (SQL) et R

Nicolas Klutchnikoff

October 2020

Introduction

Très souvent les données sont regroupées dans des **bases de données**. Ces outils offrent à la fois :

- un système d'organisation des données;
- une manière d'y accéder de façon efficiente.

Le langage SQL (Structured Query Language) est majoritairement utilisé pour formuler les requêtes qui permettent de manipuler les données. Il est utilisé par les bases de données les plus populaires :

- PostgreSQL;
- MariaDB;
- SQLite.

L'objectif n'est pas d'apprendre SQL (même si nous allons voir quelques notions) mais plutôt d'utiliser les outils de **R** pour accéder aux données présentes sur des serveurs de bases de données...

Pour cela nous allons utiliser une couche d'abstraction appelée **DBI**. Ce package offre une interface de communication entre **R** et différentes bases de données de type SQL à l'aide de pilotes dédiés.

```
library(DBI)
```

Les bases de données de type SQL utilisent le paradigme individus/variables :

- les bases contiennent des tables (équivalentes aux data-frames);
- les tables contiennent des colonnes (ou champs) qui regroupent des informations de même type;
- les enregistrements ou entrées d'une tables correspondent aux lignes de cette table.

Les tables sont reliées entre elles grâce à des identifiants (clés primaires/clés étrangères).

```
con <- dbConnect(  
  RPostgres::Postgres(),  
  dbname = "DATABASE_NAME",  
  host = "HOST",  
  port = 5432,  
  user = "USERNAME",  
  password = "PASSWORD")
```

On remarque au passage la fonction `RPostgres::Postgres()` qui fournit un pilote (ou driver) pour la base de données voulue.

On peut remplacer cette fonction par `RMariaDB::MariaDB()` pour se connecter à une base de données MariaDB.

Dans la suite nous allons utiliser SQLite. C'est une base de données qui n'est pas basée sur le principe client/serveur. SQLite permet de travailler sur des bases de données stockées dans des fichiers voire directement en mémoire vive. C'est donc très simple à mettre en œuvre.

```
con <- dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

On ouvre ici une connexion vers une base de données SQLite contenue en mémoire vive !

Pour l'instant il n'y a aucune table :

```
dbListTables(con)
```

```
## character(0)
```

Peupler une base de données

```
df <- data.frame(  
  x = runif(25),  
  label = sample(c("A", "B"), size = 25, replace = TRUE)  
)  
dbWriteTable(con, name = "Exemple", value = df)  
dbListTables(con)
```

```
## [1] "Exemple"
```

Nous venons ainsi de créer une première table dans la base de données à l'aide d'un data-frame (fonctionne aussi avec un **tibble**). Les champs (colonnes) sont bien identiques aux variables du data-frame :

```
dbListFields(con, "Exemple")
```

```
## [1] "x"      "label"
```



```
SELECT j FROM df WHERE i GROUP BY by
```

Ce pseudo code rappelle les pseudo-codes suivants :

- data-table :

```
dt[i, j, by]
```

- tibble et dplyr :

```
df %>% group_by(by) %>% filter(i) %>% mutate(j)
```

En effet SQL et la proximité entre BdD et data-frame a inspiré les créateurs des deux packages.

Première requête

```
res <- dbSendQuery(con, "SELECT * FROM Exemple WHERE label = 'A'")  
res
```

```
## <SQLiteResult>  
##   SQL  SELECT * FROM Exemple WHERE label = 'A'  
##   ROWS Fetched: 0 [incomplete]  
##           Changed: 0
```

On note que **res** est un objet d'un type particulier : **SQLiteResult**. Avant de voir comment s'en servir, prenons de bonnes habitudes en libérant les ressources locales et distantes liées au résultat de la requête

```
dbClearResult(res)
```

Collecter les données

```
res <- dbSendQuery(con, "SELECT * FROM Exemple WHERE label = 'A'")
while(!dbHasCompleted(res)){
  chunk <- dbFetch(res, n = 8)
  print(res)
  print(chunk[,2])
}
```

```
## <SQLiteResult>
##   SQL  SELECT * FROM Exemple WHERE label = 'A'
##   ROWS Fetched: 8 [incomplete]
##           Changed: 0
## [1] "A" "A" "A" "A" "A" "A" "A" "A"
## <SQLiteResult>
##   SQL  SELECT * FROM Exemple WHERE label = 'A'
##   ROWS Fetched: 14 [complete]
##           Changed: 0
## [1] "A" "A" "A" "A" "A" "A"
dbClearResult(res)
```

Quelques fonctions utiles (1)

```
con %>% dbExistsTable("Example")
```

```
## [1] FALSE
```

```
con %>% dbExistsTable("Exemple")
```

```
## [1] TRUE
```

```
dbRemoveTable(con, "Example") # produit une erreur !
```

```
if(dbExistsTable(con, "Example")){  
  dbRemoveTable(con, "Example")  
}
```

Quelques fonctions utiles (2)

```
query <- "SELECT x FROM Exemple WHERE label = 'A' ORDER BY x LIMIT 2"  
df <- dbGetQuery(con, query)  
df
```

```
##           x  
## 1 0.001449368  
## 2 0.010623835
```

La requête est soumise, exécutée et les données produites sont collectées puis retournées sous la forme d'un data-frame.

```
class(df)
```

```
## [1] "data.frame"
```

Il faut **TOUJOURS** fermer la connexion à la base de données!

```
dbDisconnect(con)
```

Exemple complet

LE vélo STAR

```
con <- dbConnect(RSQLite::SQLite(), dbname = "data/LEveloSTAR.sqlite3")
dbListTables(con)
```

```
## [1] "Etat"      "Topologie"
```

```
dbListFields(con, "Etat")
```

```
## [1] "id"                "nom"
## [3] "latitude"          "longitude"
## [5] "etat"              "nb_emplacements"
## [7] "emplacements_disponibles" "velos_disponibles"
## [9] "date"              "data"
```

```
dbListFields(con, "Topologie")
```

```
## [1] "id"                "nom"                "adresse_numero"
## [4] "adresse_voie"      "commune"            "latitude"
## [7] "longitude"         "id_correspondance"  "mise_en_service"
## [10] "nb_emplacements"   "id_proche_1"        "id_proche_2"
## [13] "id_proche_3"       "terminal_cb"
```


Problème

La gare de Rennes se trouve à la position GPS (48.103712, -1.672342). On souhaite trouver les trois stations les plus proches et afficher certaines informations utiles :

- le nom et l'adresse des stations
- parmi les stations en fonctionnement et disposant d'au moins un vélo.

Les informations nécessaires se trouvent dans les deux tables. Il va donc falloir procéder méthodiquement et faire des jointures de tables. Le pseudo code est de la forme :

```
SELECT left.**, right.**  
FROM left  
LEFT JOIN right  
ON (left.id=right.id)
```

dans un cas simple

On commence par construire la requête **SELECT**.

```
q1 <- "  
left.id AS id,  
right.nom AS nom,  
(  
  COALESCE(right.adresse_numero, '') ||  
  ' ' ||  
  COALESCE(right.adresse_voie, '')  
) AS adresse,  
left.distance AS distance"
```

Puis on construit une nouvelle table comme table primaire.

```
q2 <- "  
(  
  SELECT id,  
    POWER((latitude - 48.103712), 2.0) +  
    POWER((longitude + 1.672342), 2.0) AS distance  
  FROM Etat  
  WHERE ((etat = 'En fonctionnement') AND (velos_disponibles > 0))  
) AS left  
"
```

Requête (3)

```
query <- paste(
  "SELECT", q1,
  "FROM", q2,
  "LEFT JOIN Topologie AS right
  ON (left.id = right.id)
  ORDER BY distance
  LIMIT 3")
```

```
df <- dbGetQuery(con, query)
df
```

```
##   id          nom          adresse    distance
## 1 15 Gares - Solférino    18 Place de la Gare 1.350434e-06
## 2 45 Gares Sud - Féval    19 B Rue de Châtillon 2.761371e-06
## 3 84 Gares - Beaumont 22 Boulevard de Beaumont 2.783296e-06
```

Pour finir

Comme toujours :

```
dbDisconnect(con)
```

Il faut apprendre un peu de SQL.

C'est un investissement rentable seulement si on l'utilise régulièrement.

Comment faire si l'on en a une utilisation occasionnelle? **dplyr**!

Vu la proximité des concepts (voulue) entre la grammaire de **dplyr** et SQL, il existe un traducteur pour SQL. En réalité **dplyr** se veut un langage de manipulation des données assez général et il existe aussi un traducteur pour data-table. Il y a néanmoins une perte d'efficacité... le prix de la conversion!

SQL et dplyr

Connexion et initialisations

```
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "data/LEveloSTAR.sqlite3")
etat_db <- tbl(con, "Etat")
topologie_db <- tbl(con, "Topologie")

class(etat_db)
```

```
## [1] "tbl_SQLiteConnection" "tbl_dbi"          "tbl_sql"
## [4] "tbl_lazy"            "tbl"
etat_db
```

```
## # Source:   table<Etat> [?? x 10]
## # Database: sqlite 3.33.0
## #   [/home/xavier/CloudIMT/Enseignement/FormationParis/BDD/02
## #   sql/data/LEveloSTAR.sqlite3]
##       id nom      latitude longitude etat  nb_emplacements emplacements_di-
##       <int> <chr>    <dbl>      <dbl> <chr>    <int>          <int>
## 1     1 Répu~    48.1      -1.68 En f~    30           25
## 2     2 Mair~    48.1      -1.68 En f~    24            6
## 3     3 Cham~    48.1      -1.68 En f~    24            8
## 4    10 Musé~    48.1      -1.67 En f~    16            4
## 5     12 TNB     48.1      -1.67 En f~    28           16
## 6     14 Laën~    48.1      -1.67 En f~    16            3
## 7     17 Char~    48.1      -1.68 En f~    24           17
## 8     20 Pont~    48.1      -1.68 En f~    20            9
## 9     22 Ober~    48.1      -1.66 En f~    20           13
## 10    25 Offi~    48.1      -1.68 En f~    10            6
## # ... with more rows, and 3 more variables: velos_disponibles <int>,
## #   date <dbl>, data <chr>
```

Exemple (1)

```
res <- etat_db %>%  
  arrange(latitude) %>%  
  select(nom, latitude) %>%  
  head(2)
```

res

```
## # Source:      lazy query [?? x 2]  
## # Database:    sqlite 3.33.0  
## #   [/home/xavier/CloudIMT/Enseignement/FormationParis/BDD/02  
## #   sql/data/LEveloSTAR.sqlite3]  
## # Ordered by: latitude  
##   nom      latitude  
##   <chr>     <dbl>  
## 1 Alma      48.1  
## 2 Italie    48.1
```


Exemple (2)

```
res <- etat_db %>%  
  arrange(latitude) %>%  
  select(nom, latitude) %>%  
  head(2) %>%  
  collect()
```

res

```
## # A tibble: 2 x 2  
##   nom      latitude  
##   <chr>      <dbl>  
## 1 Alma        48.1  
## 2 Italie       48.1
```

```
res <- etat_db %>%  
  arrange(latitude) %>%  
  select(nom, latitude) %>%  
  head(2) %>%  
  show_query()
```

```
## <SQL>  
## SELECT `nom`, `latitude`  
## FROM (SELECT *  
## FROM `Etat`  
## ORDER BY `latitude`)  
## LIMIT 2
```

À la main, on aurait codé plus simplement

```
SELECT nom, latitude  
FROM Etat  
ORDER BY latitude  
LIMIT 2
```

Ça n'a pas grande importance car il y a des optimisations internes du code pour que les vitesses d'exécution soient similaires.

```
res <- etat_db %>%  
  filter(latitude == min(latitude)) %>%  
  select(nom, latitude) # Ce code produit une erreur  
  
res <- etat_db %>%  
  summarise(m_l = min(latitude))
```

On ne peut pas utiliser toutes les fonctions de R car le code est exécuté côté base de données.

Ici, on a en réalité deux fonctions `min()` différentes :

- une fonction de fenêtrage (window function) qui n'existe pas en SQLite (mais qui existe en PostgreSQL);
- une fonction d'agrégation (aggregate function) qui existe en SQLite.