Certificat Data Scientist

Structured Query Language (SQL) avec R

Pierre LAFAYE DE MICHEAUX

Novembre 2023

Introduction

Présentation

Très souvent les données sont regroupées dans des **bases de données**. Des outils spécialisés 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 Systèmes de Gestion de Bases de Données (SGBD) les plus populaires :

- PostgreSQL ;
- MariaDB ;
- SQLite ;
- MySQL.

Présentation

Définition : Un système de gestion de base de données (abr. SGBD) est un logiciel système servant à stocker, à manipuler ou gérer, et à partager des données dans une base de données, en garantissant la qualité, la pérennité et la confidentialité des informations, tout en cachant la complexité des opérations.

Objectifs

L'ojectif dans ce cours 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 (DataBase Interface). Le package `DBI` (https://dbi.r-dbi.org/) offre une interface de communication entre `R` et différentes bases de données de type SQL à l'aide de pilotes dédiés.

```
library(DBI)
```

```
## Warning: package 'DBI' was built under R version 4.3.2
```

Bases de données relationnelles

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 dans le jargon des bases de données) qui regroupent des informations de même type ;
- les enregistrements ou entrées d'une table correspondent aux lignes de cette table (individus ou unités statistiques).

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

Connexion à un serveur

```r
con <- DBI::dbConnect(
  drv = 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.

Pour se connecter à d'autres types de base de données, on peut remplacer cet appel de fonction par :

- `RMariaDB::MariaDB()`
- `RSQLite::SQLite()`
- `RMySQL::MySQL()`

SQLite

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 sur votre ordinateur, 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 (nommée `con`) 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),
  y = rpois(25, 1)
)
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" "y"
```

SQL en bref !

```sql
SELECT label, SUM(2*x) AS somme FROM table_df WHERE x > 0.5 GROUP BY label
```

Ce code rappelle ce que l'on peut faire aves les `tibble` et le package `dplyr` :

```r
df %>% select(label,x) %>% filter(x > 0.5) %>%
     mutate(y = 2 * x) %>%
     group_by(label) %>% summarise(somme = sum(y))
```

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 pour les SGBD autres que SQLite) liées au résultat de la requête

```
dbClearResult(res)
sum(df$label == "A")
```

```
## [1] 13
```

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: 13 [complete]
##        Changed: 0
## [1] "A" "A" "A" "A" "A"
```

```
dbClearResult(res)
```

Quelques fonctions utiles (1)

```
con %>% dbExistsTable("Example") # avec un 'a'
```

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

```
con %>% dbExistsTable("Exemple") # avec un 'e'
```

```
## [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 0.01634318
## 2 0.20159683
```

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"
```

Pour finir

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

```
dbDisconnect(con)
```

Exemple complet

L'objectif n'est pas de comprendre le langage SQL mais juste de voir que l'on peut, depuis R, faire des requêtes SQL complexes.

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 de la gare 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

Requête (1)

On commence par construire le contenu de 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"
```

`COALESCE` renvoie la première expression non NULL parmi ses arguments, ou NULL si tous ses arguments sont NULL. `||` est un opérateur de concaténation de chaînes.

`AS` permet de créer un alias pour renommer temporairement/créer une colonne ou une table, et ainsi faciliter la lecture des requêtes (ici `id`, `nom`, `adresse`, `distance`).

Requête (2)

Puis on construit une nouvelle table `left` comme table primaire (contenu de la requête `FROM`).

```
q2 <- "
(
  SELECT id,
  SQRT(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
"
```

https://www.google.com/maps/search/48.103712,+-1.672342

Note : l'unité de distance est en degrés (voir http://villemin.gerard.free.fr/aGeograp/Distance.htm).

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 0.001162082
## 2 45 Gares Sud - Féval    19 B Rue de Châtillon 0.001661737
## 3 84  Gares - Beaumont 22 Boulevard de Beaumont 0.001668321
```

Pour finir

Comme toujours :

```
dbDisconnect(con)
```

Requête (4)

La requête SQL au complet :

```sql
SELECT left.id AS id, left.distance AS distance
       right.nom AS nom,
       (
         COALESCE(right.adresse_numero, '') || ' ' || COALESCE(right.adresse_voie, '')
       ) AS adresse
FROM (
  SELECT id,
         SQRT(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
LEFT JOIN Topologie AS right
ON (left.id = right.id)
ORDER BY distance
LIMIT 3
```

Une remarque

Il faut apprendre un peu de SQL. Aller plus loin 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") # tbl creates a table from a data source
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 3.43.2 [C:\Users\pierr\Documents\CEPE\CEPE-RBasesDonnees-2023\02 sql\data\LEveloSTAR
##       id nom     latitude longitude etat  nb_emplacements emplacements_disponi~1
##    <int> <chr>      <dbl>     <dbl> <chr>           <int>                  <int>
##  1     1 Républ~     48.1     -1.68 En f~              30                     25
##  2     2 Mairie      48.1     -1.68 En f~              24                      6
##  3     3 Champ ~     48.1     -1.68 En f~              24                      8
##  4    10 Musée ~     48.1     -1.67 En f~              16                      4
##  5    12 TNB         48.1     -1.67 En f~              28                     16
##  6    14 Laënnec     48.1     -1.67 En f~              16                      3
##  7    17 Charle~     48.1     -1.68 En f~              24                     17
##  8    20 Pont d~     48.1     -1.68 En f~              20                      9
##  9    22 Oberth~     48.1     -1.66 En f~              20                     13
## 10    25 Office~     48.1     -1.68 En f~              10                      6
## # i more rows
## # i abbreviated name: 1: emplacements_disponibles
```

Exemple (1)

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

```
## # Source:     SQL [2 x 2]
## # Database:   sqlite 3.43.2 [C:\Users\pierr\Documents\CEPE\CEPE-RBasesDonnee
## # 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
```

Traduction (1)

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

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

Traduction (2)

```r
res <- etat_db %>%
  filter(latitude == min(latitude)) %>%
  select(nom, latitude)
```

```r
res <- etat_db %>%
  summarise(m_l = min(latitude))
```

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

```r
res <- etat_db %>%
  summarise(m_l = pnorm(latitude)) # L'objet res contient une erreur
```

```r
dbDisconnect(con)
```