R et les bases de données

B.Thieurmel - benoit.thieurmel@datastorm.fr

ENEDIS : R & SQL - 22/01/2021

FRAMEWORK DBI

Les packages existants

Il existe au minimum un package qui permet de se connecter à chaque type de bases de données, par exemple :

- **RMySQL** : base de données MySQL
- **RPostgres** : base de données PostgreSQL (à prioriser *vs* le package plus ancien **RPostgreSQL**, qui n'est plus maintenu et moins performant)
- **ROracle** : base de données Oracle
- **RSQLServer** : base de données MS SQL Server
- **RSQLite** : base de données SQLite
- **RJDBC** : connexion à différents types de bases de données via du Java
- **RODBC** : connexion à différents types de bases de données via du ODBC
- **mongolite** : base de données NoSQL mongodb
- ....

DBI : syntaxe usuelle de connexion (1/2)

Le package **DBI** propose une interface commune d'utilisation des différentes bases de données.

Les autres packages comme **RMySQL** ou **RPostgres** servent alors essentiellement à fournir le *driver* permettant la connexion à la base de données correspondante.

La syntaxe usuelle d'utilisation est la suivante :

1. ouverture de la connexion avec `dbConnect()`
2. requêtage avec `dbGetQuery()`, avec le passage directement d'une requête **SQL**
3. fermeture de la connexion avec `dbDisconnect()`

DBI : syntaxe usuelle de connexion (2/2)

```
require(DBI)
require(RPostgres)

# ouverture d'une connexion
con <- dbConnect(RPostgres::Postgres(), # RMySQL::MySQL()
                 host = "myserver",
                 port = 123,
                 dbname = "database",
                 user = "benoit",
                 password = "security")

# exécution d'une requête et récupération du résultat
res <- dbGetQuery(con, "SELECT * FROM mytable")
```

```r
# Fermeture de la connexion
dbDisconnect(con)
```

DBI : principales fonctions

- `dbConnect()` & `dbDisconnect()` : connexion / déconnexion à la base
- `dbListTables()` & `dbListFields()` : récupération des tables disponibles et des champs présents
- `dbGetQuery()` : envoi d'une requête SQL retournant des données (`SELECT ...`) et récupération du résultat
- `dbReadTable()` & `dbWriteTable()` : lecture / écriture d'une table dans sa globalité
- `dbSendQuery()` : exécution d'une requête
  - suivie d'un `dbFetch()` si on souhaite récupérer le résultat
  - et dans tous les cas de `dbClearResult()`
  - va nous permettre notamment d'insérer des lignes / supprimer des tables, ...
- ...

DBI : dbConnect

Le premier argument **obligatoire** de `dbConnect` est donc le *driver* permettant de se connecter à la base de données. Ce *driver* étant en général fourni pour le biais d'un autre package.

Les principaux paramètres de connexion sont ensuite les suivants :

- **user** : nom d'utilisateur
- **password** : mot de passe
- **host** : adresse ip ou nom de l'hôte hébergeant la base
- **port** : numéro du port ouvert de la base
- **dbname** : nom de la base de données

Quelques exceptions cependant dans le nom des arguments, comme par exemple lors de la connexion à une base MS SQL...

DBI : dbGetQuery

L'utilisation de la fonction `dbGetQuery` avec le passage directe d'une requête **SQL** est ensuite la façon la plus simple d'interroger la base de données :

```r
# connexion à une base temporaire SQLite in-memory
con <- dbConnect(RSQLite::SQLite(), ":memory:")

# ecriture d'une table
dbWriteTable(con, "mtcars", mtcars)

# exécution d'une requête et récupération du résultat
res <- dbGetQuery(con, "SELECT * FROM mtcars WHERE cyl = 4");head(res, n = 2)
```

```
##    mpg cyl  disp hp drat   wt  qsec vs am gear carb
## 1 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
## 2 24.4   4 146.7 62 3.69 3.19 20.00  1  0    4    2
```

```r
# déconnexion
dbDisconnect(con)
```

ATTENTION à bien respecter les règles associées à la base de données, comme l'utilisation de simple quote ' ou la casse des caractères

DBI : dbSendQuery

`dbSendQuery` exécute une requête sur la connexion. Elle ne retourne pas par défaut le résultat, ce qui possible avec un `dbFetch()`, et il faut dans tous les cas finir par un `dbclearResult()` pour libérer la mémoire.

```
# connexion à une base temporaire SQLite in-memory
con <- dbConnect(RSQLite::SQLite(), ":memory:"); dbWriteTable(con, "mtcars", mtcars)
# exécution de la requête
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4"); rs
```

```
## <SQLiteResult>
##   SQL  SELECT * FROM mtcars WHERE cyl = 4
##   ROWS Fetched: 0 [incomplete]
##        Changed: 0
```

```
# récupération du résultat
data <- dbFetch(rs); head(data, n = 2)
```

```
##    mpg cyl  disp hp drat   wt  qsec vs am gear carb
## 1 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
## 2 24.4   4 146.7 62 3.69 3.19 20.00  1  0    4    2
```

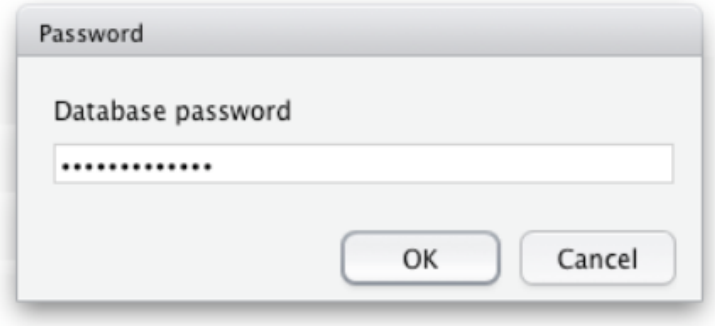
```
# nettoyage & déconnexion
dbClearResult(rs) ; dbDisconnect(con)
```

SECURISER LES INFORMATIONS DE CONNEXION

Via une fenêtre utilisateur ?

Dans RStudio avec `rstudioapi::askForPassword()`, ou avec `getPass::getPass()`

```
# ouverture d'une connexion
con <- dbConnect(RPostgres::Postgres(),
                 host = "myserver",
                 port = 123,
                 dbname = "database",
                 user = "benoit",
                 password = rstudioapi::askForPassword("Database password"))
```



avec les options dans R ?

Dans un script **privé et non partagé**, définir des options de connexion :

```
options(database_userid = "myuserid")
options(database_password = "mypassword")
```

Puis dans le script **principal et partagé** :

```
# ouverture d'une connexion
con <- dbConnect(RPostgres::Postgres(),
                 host = "myserver",
                 port = 123,
                 dbname = "database",
                 user = getOption("database_userid"),
                 password = getOption("database_password"))
```

avec des variables d'environnement ?

Même principe qu'avec les options, sauf que les variables d'environnement doivent être définies directement sur la machine, ou bien dans le fichier *.Renviron* présent à la racine.

```
# .Renviron
database_userid = "username"
database_password = "password"
```

Puis dans le script **principal et partagé** :

```
# ouverture d'une connexion
con <- dbConnect(RPostgres::Postgres(),
                 host = "myserver",
                 port = 123,
                 dbname = "database",
                 user = Sys.getenv("database_userid"),
                 password = Sys.getenv("database_password"))
```

dans un fichier YAML ?

exemple.yml :

```
# Liste de configuration pour une base de données
db:
  host : 10.244.36.68
  port : 5432
  uid : user
  pwd : pwd
  dbname : database
```

Dans R, on charge ensuite le fichier avec la fonction `yaml.load_file` :

```
require(yaml)
conf <- yaml.load_file("exemple.yml")
con <- dbConnect(RPostgres::Postgres(),
                 host = conf$db$host, port = conf$db$port,
                 dbname = conf$db$dbname, user = conf$db$uid,
                 password = conf$db$pwd)
```

avec le package config ?

Le package **config** facilite la gestion d'environnements spécifiques, en se basant simplement sur une fichier `config.yml`.

- par défaut, **config** cherchera ce fichier dans le répertoire de travail courant. Mais nous pouvons également passer le chemin du fichier.

Plus d'informations : https://cran.r-project.org/web/packages/config/vignettes/introduction.html

---

SECURISER LES REQUETES

Concaténation avec paste ?

Dès lors que l'on souhaite développer des requêtes paramétrables, il est facile et tentant d'utiliser simplement la fonction `paste` pour créer la requête finale.

```
cyl_subset <- 4
# exécution de la requête
res <- dbGetQuery(con,
  paste(
    "SELECT * FROM mtcars WHERE cyl = ",
    cyl_subset
  )
)
```

Cela amène potentiellement une faille de sécurité car, sans contrôles préalables des arguments passés, l'utilisateur peut très bien insérer une reqûete dangereuse et commettre une attaque...

Il est donc conseillé d'utiliser des requêtes dîtes **paramétrables**, ce qui possible de plusieurs façons :

- en utilisant `dbSendQuery()` et `dbBind()`
- ou le package **glue** et la fonction `glue_sql()`

dbSendQuery & dbBind (1/2)

1. création d'une requête avec `dbSendQuery`, en utilisant des `?` pour le/les paramètre(s)
2. utilisation de `dbBind` pour passer la/les valeur(s)
3. récupération avec `dbFetch`
4. Appels multiples possibles

```
# un argument simple
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(con, "mtcars", mtcars)

simple_bind <-  dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = ?")
dbBind(simple_bind, list(4))
head(dbFetch(simple_bind), n = 2)
```

```
##    mpg cyl  disp hp drat   wt  qsec vs am gear carb
## 1 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
## 2 24.4   4 146.7 62 3.69 3.19 20.00  1  0    4    2
```

dbSendQuery & dbBind (2/2)

```
# plusieurs arguments simples
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(con, "mtcars", mtcars)

multiple_bind <-  dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = ? AND am = ?")
```

```
dbBind(multiple_bind, list(4, 0))
head(dbFetch(multiple_bind), n = 2)
```

```
##    mpg cyl  disp hp drat   wt qsec vs am gear carb
## 1 24.4   4 146.7 62 3.69 3.19 20.0  1  0    4    2
## 2 22.8   4 140.8 95 3.92 3.15 22.9  1  0    4    2
```

```
# second appel
dbBind(multiple_bind, list(4, 1))
head(dbFetch(multiple_bind), n = 2)
```

```
##    mpg cyl  disp hp drat   wt  qsec vs am gear carb
## 1 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
## 2 32.4   4  78.7 66 4.08 2.20 19.47  1  1    4    1
```

Inconvénient majeur : : impossible de passer des arguments plus complexes, comme des ensembles de valeurs pour l'opérateur SQL `IN` par exemple

glue_sql (1/2)

1. création de la requête avec `glue_sql`, en utilisant des `{nom_var}` pour le/les paramètre(s)
2. récupération directement avec `dbGetQuery`

```
# un argument simple
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(con, "mtcars", mtcars)

simple_glue <- glue::glue_sql(
  "SELECT * FROM mtcars WHERE cyl = {val_cyl}",
  val_cyl = 4, # passage des variables
  .con = con # et de la connexion
)

simple_glue
```

```
## <SQL> SELECT * FROM mtcars WHERE cyl = 4
```

```
head(dbGetQuery(con, simple_glue), 2)
```

```
##    mpg cyl  disp hp drat   wt  qsec vs am gear carb
## 1 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
## 2 24.4   4 146.7 62 3.69 3.19 20.00  1  0    4    2
```

glue_sql (2/2)

glue va gérer les arguments plus complexes avec l'ajout d'un * et l'utilisation de parenthèses `({nom_var*})`

```
# deux arguments, dont un complexe
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(con, "mtcars", mtcars)

multiple_glue <- glue::glue_sql(
  "SELECT * FROM mtcars WHERE cyl IN ({val_cyl*}) AND am = {val_am}",
  val_cyl = c(4,6),
  val_am = 0,
  .con = con
)
```

```
multiple_glue
```

```
## <SQL> SELECT * FROM mtcars WHERE cyl IN (4, 6) AND am = 0
```

```
head(dbGetQuery(con, multiple_glue), 2)
```

```
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 2 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

DBPLYR

dbplyr, c'est quoi ?

Le package **dbplyr** permet d'utiliser la syntaxe du package **dplyr**, mais directement sur des bases de données. Le code **R** sera alors convertit automatiquement en une requête SQL, et cette requête sera exécuter le moment venu quand on souhaitera récupérer le résultat dans **R**. Il fonctionne à date avec les connexions / bases de données suivantes :

- **RMySQL**
- **RPostgreSQL** et **RPostgres**
- **RSQLite**
- **odbc**
- **bigrquery** (Google's)

dbplyr

dplyr cheatsheet

Manipuler les donnees avec dplyr

Les bases de dplyr : sélection des colonnes

`select` permet de sélectionner des colonnes d'un tableau de données

- les noms sans quote, avec un `-` devant si on veut plutôt enlever certaines colonnes, ou en utilisant l'opérateur `:` pour une plage

```
mtcars %>% select(cyl, disp) # uniquement cyl et disp
mtcars %>% select(-cyl, -disp) # tout sauf cyl et disp
mtcars %>% select(cyl:wt) # de cyl à wt
```

- en utilisant des chaînes de caractères et `any_of` ou `all_of`

```
mtcars %>% select(any_of(c("cyl", "disp"))) # dyl et disp
```

- avec des expressions régulières et les fonctions `starts_with`, `ends_width`, `contains` ou `matches`

```
mtcars %>% select(starts_with("c")) # les colonnes commençant par c
```

Les bases de dplyr : filtre sur les lignes

`filter` permet de filtrer les lignes à conserver

- en passant une ou plusieurs expressions **R** qui retournent des booléans (`>=`, `==`, `!=`, `%in%`, ...)
- dans le cas de plusieurs expressions, elles seronts combinées avec le `&` logique

```r
mtcars %>% filter(am == 0) # renvoie les lignes avec am = 0
mtcars %>% filter(am == 0 | am == 1) # am = 0 ou am = 1
mtcars %>% filter(am == 0 & vs == 1) # am = 0 et vs = 1
mtcars %>% filter(am == 0, vs == 1) # am = 0 et vs = 1
```

- et on peut utiliser toutes les fonctions **R** disponibles

```r
mtcars %>% filter(mpg == max(mpg))
```

- utilisation des noms de colonnes en caractères ? avec `.data`, qui fait référence à l'ensemble des données (`list`)

```r
var = "am" ; value = 0
mtcars %>% filter(.data[[var]] == value)
```

Les bases de dplyr : ordonner les données

`arrange` permet d'ordonner les données

- Par défaut par ordre croissant, en utilisant le nom de la colonne sans quote

```r
mtcars %>% arrange(mpg) # tri par mpg croissant
mtcars %>% arrange(am, mpg) # am croissant, puis mpg croissant
```

- utilisation des `desc` pour un tri décroissant

```r
mtcars %>% arrange(desc(mpg)) # tri par mpg desc
mtcars %>% arrange(desc(am), mpg) # am desc, puis mpg croissant
```

- `.data` avec des chaînes de caractères

```r
var <- "mpg"
mtcars %>% arrange(desc(.data[[var]]))
```

Les bases de dplyr : créer des nouvelles colonnes

`mutate` permet de créer une ou plusieurs nouvelles colonnes. **Syntaxe :** nom (avec ou sans quote) = expression

```r
mtcars %>% mutate(wt_sq = wt ^ 2) # ajout de wt au carrée
mtcars %>% mutate("wt_sq" = wt ^ 2) # idem
```

- plusieurs colonnes à la suite, les colonnes étant créer dans l'ordre, elles sont donc disponibles si besoin pour la création des colonnes suivantes

```r
mtcars %>% mutate(wt_sq = wt ^ 2, wt_sq_sup10 = wt_sq > 10)
```

- et toujours `.data` avec des chaînes de caractères...!

```r
var <- "wt" ; mtcars %>% mutate(wt_sq = .data[[var]] ^ 2)
```

- conditions multiples ? `case_when`

```r
mtcars %>% mutate(am_vs = case_when(
  am == 0 & vs == 0 ~ "Nothing", am == 0 & vs == 1 ~ "Vs only",
  am == 1 & vs == 0 ~ "Am only", am == 1 & vs == 1 ~ "Am & Vs",
  TRUE ~ "Other" # valeur par défaut
))
```

Les bases de dplyr : agrégation par sous-population (1/2)

La fonction `group_by` va nous permettre de définir les sous-population sur lesquelles nous allons pouvoir effectuer des opérations

- En enchaînant sur un `mutate`, nous calculerons par exemple une nouvelle variable pour chaque sous-population et le résultat sera affecté à l'ensemble du jeu de données initial

```
# on enchaîne les opérations avec %>%
# le group_by doit placé avant le mutate
mtcars_mod <- mtcars %>% group_by(cyl) %>%
  mutate(mean_mpg = mean(mpg)) %>%
  select(mpg, cyl, mean_mpg)
head(mtcars_mod, n = 4)
```

```
## # A tibble: 4 x 3
## # Groups:   cyl [2]
##     mpg   cyl mean_mpg
##   <dbl> <dbl>    <dbl>
## 1  21       6     19.7
## 2  21       6     19.7
## 3  22.8     4     26.7
## 4  21.4     6     19.7
```

Les bases de dplyr : agrégation par sous-population (2/2)

- En enchaînant sur un `summarize`, on pourra calculer des indicateurs sur nos sous-populations et récupérer directement le tableau agrégé (`summarize` peut être utiliser sur l'ensemble des données, sans `group_by`)
- Les colonnes définissant les sous-populations seront par défaut présentes dans le résultat
- Utilisation de la fonction `n()` pour compter le nombre de lignes

```
mtcars %>% group_by(am, vs) %>%
  summarise(
    min_mpg = min(mpg),
    max_mpg = max(mpg),
    n_rows = n()
  )
```

```
## `summarise()` regrouping output by 'am' (override with `.groups` argument)
```

```
## # A tibble: 4 x 5
## # Groups:   am [2]
##      am    vs min_mpg max_mpg n_rows
##   <dbl> <dbl>   <dbl>   <dbl>  <int>
## 1     0     0    10.4    19.2     12
## 2     0     1    17.8    24.4      7
## 3     1     0    15      26        6
## 4     1     1    21.4    33.9      7
```

Les bases de dplyr : autres fonctions utiles

- `slice()` : sélectionner des lignes du tableau selon leur position
- `relocate()` : changer l'ordre des colonnes, `rename()` : changer le noms des colonnes
- `sample_n()` et `sample_frac()` : sélectionner un échantillon aléatoire
- `lead()` et `lag()` : créer des variables décalées
- `distinct()` : conserver uniquement les observations uniques

fusion de tables

- `inner_join()` : les lignes présentes dans le premier jeu de données x **ET** le seconde jeu de données y sont conservées dans la table finale, par rapport aux clefs définies
- `full_join()` : les lignes présentes dans le premier jeu de données x **OU** le seconde jeu de données y
- `left_join()` : les lignes présentes uniquement dans x
- `right_join()` : les lignes présentes uniquement dans y

Utilisation avec les bases de données

1. Ouverture de la connexion à la base de données avec `dbConnect`
2. Branchement à une table de la base de données avec `tbl`
3. Utilisation *classique* de la syntaxe **dplyr**.
   - **dplyr** va se charger de créer la requête **SQL** correspondante
   - Il est possible de récupérer cette requête avec `show_query()`
   - la requête ne sera exécutée que quand le code **R** suivant aura effectivement besoin du résultat
4. Récupération du résultat dans **R** avec `collect()`

dbplyr : exemples (1/4)

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
# ecriture des données avec copy_to
# plus complet que dbWriteTable
mtcars$name <- row.names(mtcars)
copy_to(dest = con, df = mtcars,
        name = "mtcars", indexes = list("name")
)
# branchement à la table sql mtcars
mtcars_con <- tbl(con, "mtcars") ; mtcars_con
```

```
## # Source:   table<mtcars> [?? x 12]
## # Database: sqlite 3.33.0 [:memory:]
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb name
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4 Mazda RX4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4 Mazda RX4 ~
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1 Datsun 710
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1 Hornet 4 D~
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2 Hornet Spo~
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1 Valiant
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4 Duster 360
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2 Merc 240D
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2 Merc 230
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4 Merc 280
## # ... with more rows
```

dbplyr : exemples (1/4)

```
# syntaxe dplyr
first_query <- mtcars_con %>%
  filter(am == 0, vs == 1,  cyl == 6)  %>%
  mutate(wt_sq = wt ^ 2) %>%
  select(mpg, am, vs, cyl, name, wt, wt_sq)
```

```
first_query # objet stocké
```

```
## # Source:   lazy query [?? x 7]
## # Database: sqlite 3.33.0 [:memory:]
##     mpg    am    vs   cyl name              wt wt_sq
##   <dbl> <dbl> <dbl> <dbl> <chr>          <dbl> <dbl>
## 1  21.4     0     1     6 Hornet 4 Drive  3.22  10.3
## 2  18.1     0     1     6 Valiant         3.46  12.0
## 3  19.2     0     1     6 Merc 280        3.44  11.8
## 4  17.8     0     1     6 Merc 280C       3.44  11.8
```

```
first_query %>% show_query() # requête associée
```

```
## <SQL>
## SELECT `mpg`, `am`, `vs`, `cyl`, `name`, `wt`, POWER(`wt`, 2.0) AS `wt_sq`
## FROM `mtcars`
## WHERE ((`am` = 0.0) AND (`vs` = 1.0) AND (`cyl` = 6.0))
```

dbplyr : exemples (3/4)

```
group_query <- mtcars_con %>%
  group_by(am, vs)  %>%
  summarise(
    min_mpg = min(mpg, na.rm = T),
    max_mpg = max(mpg, na.rm = T),
    n = n()
  )

group_query %>% show_query() # requête associée
```

```
## <SQL>
## SELECT `am`, `vs`, MIN(`mpg`) AS `min_mpg`, MAX(`mpg`) AS `max_mpg`, COUNT(*) AS `n`
## FROM `mtcars`
## GROUP BY `am`, `vs`
```

```
group_query %>% collect() # récupération du résultat
```

```
## # A tibble: 4 x 5
## # Groups:   am [2]
##      am    vs min_mpg max_mpg     n
##   <dbl> <dbl>   <dbl>   <dbl> <int>
## 1     0     0    10.4    19.2    12
## 2     0     1    17.8    24.4     7
## 3     1     0    15      26       6
## 4     1     1    21.4    33.9     7
```

dbplyr : exemples (4/4)

dbplyr convertit donc les calculs **R** qu'il connaît en la fonction **SQL** associée. Dans le cas où il ne la connaîtrait pas, il passera alors directement le code **R** au **SQL** :

```
sql_query <- mtcars_con %>%
  group_by(am, vs)  %>%
  summarise(
    n_r = n(), # fonction R connue
```

```
    n_sql  = COUNT(), # inconnue
    mean_sql = AVG(mpg) # inconnue
  )

sql_query %>% show_query()
```

```
## <SQL>
## SELECT `am`, `vs`, COUNT(*) AS `n_r`, COUNT() AS `n_sql`, AVG(`mpg`) AS `mean_sql`
## FROM `mtcars`
## GROUP BY `am`, `vs`
```

```
sql_query %>% collect()
```

```
## # A tibble: 4 x 5
## # Groups:   am [2]
##      am    vs   n_r n_sql mean_sql
##   <dbl> <dbl> <int> <int>    <dbl>
## 1     0     0    12    12     15.1
## 2     0     1     7     7     20.7
## 3     1     0     6     6     19.8
## 4     1     1     7     7     28.4
```

Références / pour aller plus loin

Database using R

Le package config

dbplyr

dplyr cheatsheet

Manipuler les donnees avec dplyr