

Module format Parquet

Apprendre à utiliser des fichiers au format Parquet avec R

Sommaire

- 1 C'est quoi un fichier parquet ?**
- 2 Comment utiliser/interroger un fichier parquet ?**
- 3 Écrire des fichiers parquet**
- 4 Manipuler des fichiers parquets avec duckdb**
- 5 Pour en savoir plus**

0.1 Avant-propos

Ce support ne couvre pas tous les aspects des traitements qu'il est possible de réaliser avec les fichiers au format Parquet mais il constitue une base sur laquelle s'appuyer si vous rencontrez des fichiers Parquet sur Cerise ou ailleurs.

1 C'est quoi un fichier parquet ?

1.1 C'est quoi le format Parquet ?

- Un nouveau format de données ...
 - ... qui compresse efficacement les fichiers
 - ... interopérable
 - conçu pour que les données soient chargées rapidement en mémoire

⋮

1.2 Caractéristiques du format Parquet

- Un stockage au format binaire (pas lisible par un humain)
- Un stockage orienté colonne (par opposition aux csv qui sont orientés lignes)
- Un fichier Parquet contient à la fois les données et des métadonnées

```

1 read_parquet("data/output.parquet", as_data_frame = FALSE)$schema
2
3 # > Table
4 # 3 rows x 4 columns
5 # $colonne1 <double not null>
6 # $colonne2 <string not null>
7 # $colonne3 <bool not null>
8 # $colonne4 <date32[day] not null>
9
10 read_parquet("data/output.parquet", as_data_frame = FALSE)$schema$metadata
11
12 # $auteur
13 # [1] "DEMESIS/BQIS"
14
15 # $description
16 # [1] "Table test de formation"
17
18 # $date creation

```

1.3 Avantages du format Parquet

- Des fichiers moins volumineux qu'en csv 500 Mo en Parquet vs 5 Go en csv

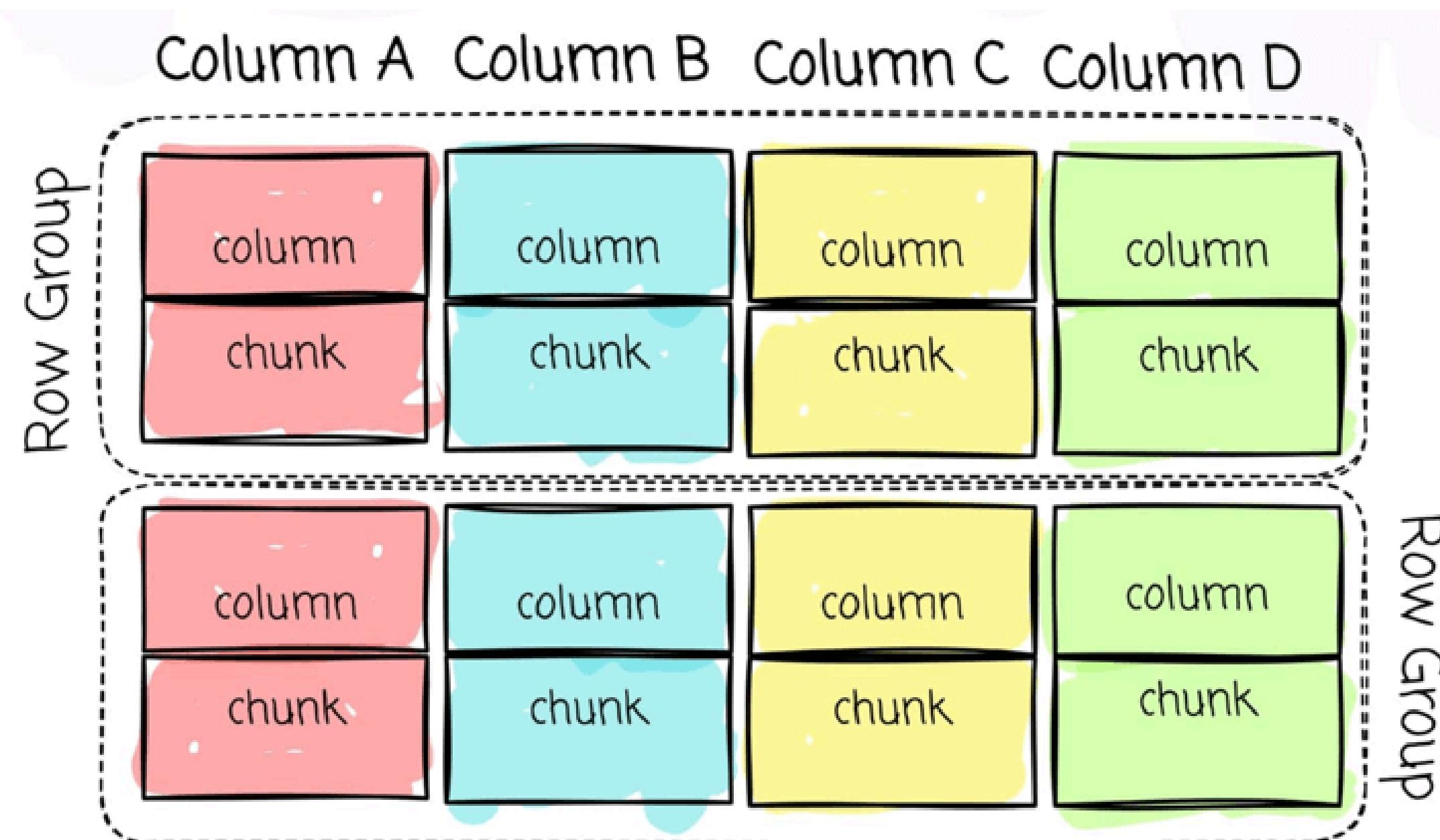
- Des requêtes plus rapides et efficaces

Seulement les données nécessaires sont lues, pas tout le fichier

- Des données conformes à la mise à disposition par le producteur (par exemple, plus de problème de codes communes...)

=> Un format très efficace pour l'analyse de données mais peu adapté à l'ajout de données en continu ou à la modification fréquente de données existantes.

1.4 Constitution d'un fichier parquet (1/2)



*Source : [voir ici](#)

1.5 Constitution d'un fichier parquet (2/2)

Un fichier Parquet est organisé en **row groups** (groupes de lignes) :

Chaque `row group` contient l'ensemble des colonnes du jeu de données.

Les données de chaque colonne sont stockées de manière colonnaire au sein du row group :

Chaque colonne est subdivisée en chunks (morceaux) stockés séquentiellement.

Cette structuration permet :

- Une lecture sélective efficace (on peut charger uniquement les colonnes nécessaires).
- Une compression optimisée, car les données similaires sont regroupées.
- Une excellente performance pour les requêtes analytiques sur de grands volumes.

1.6 Un format qui se démocratise

- L'Insee diffuse des données du [recensement de la population](#) au format Parquet
- Voir [le guide d'utilisation joint](#) pour manipuler ces données
- Premières diffusions sur [data.gouv](#) avec les bureaux de vote, les demandes de valeurs foncières, indicateurs pénaux...)
- Prévisualisations des fichiers Parquet possibles avec le nouvel explorateur de données du SSP Cloud ou avec avec l'outil [ParquetViewer](#).

:
:
:
:
:

2 Comment utiliser/interroger un fichier parquet ?

2.1 Lire un fichier avec `read_parquet()`

```
1 library(arrow)      # Le package arrow est nécessaire pour travailler avec des fichiers parquet
2 library(dplyr)      # Pour utiliser dplyr
3 library(tictoc)     # Pour le benchmark
```

Pour l'exemple, nous allons prendre une table des exploitations du RA 2020 d'une centaine de MO qui contient 416 478 lignes et 255 colonnes.

```
1 tic()
2 RA2020 <- arrow::read_parquet("data/RA2020_exploitations.parquet")
3 toc()
4 > 1.14 sec elapsed
```

Le résultat obtenu est un objet directement utilisable dans R (ici un `data.frame`).

Il est possible de sélectionner les colonnes que l'on souhaite importer dans R directement dans la fonction `read_parquet` :

```
1 tic()
2 RA2020_extrait <- arrow::read_parquet("data/RA2020_exploitations.parquet",
                                         col_select = c("NOM_DOSSIER", "SIEGE_REG", "SAU_TOT"))
3 toc()
4 > 0.06 sec elapsed
```

2.2 Comparaison avec la lecture d'un fichier rds

Voyons l'écart avec la lecture d'un fichier rds :

```
1 tic()  
2 RA2020 <- readRDS("data/RA2020_exploitations.rds")  
3 toc()  
4 > 6.15 sec elapsed
```

=> Le temps nécessaire au chargement de la table est d'environ 6 secondes !
L'écart est significatif rien que sur la lecture (X 6).

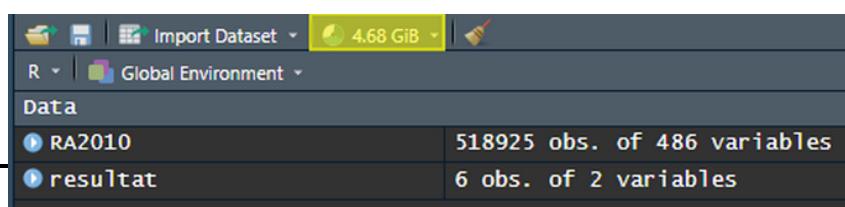
2.3 Des requêtes avec dplyr comme d'habitude

RA2020 est un data.frame : on peut donc utiliser la syntaxe dplyr :

```

1 resultat <- RA2020 |>
2   filter(SIEGE_REG == "93") |>
3   group_by(SIEGE_DEP) |>
4   summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE))
5
6 # A tibble: 6 × 2
7   SIEGE_DEP total_SAU
8   <chr>      <dbl>
9   1 04        158946.
10  2 05        91979.
11  3 06        41141.
12  4 13        145713.
13  5 83        77785.
14  6 84        112888.
```

- Le temps d'exécution de la requête est d'environ 9 secondes.
- Les ressources consommées sont importantes



2.4 Lire et exploiter un fichier parquet volumineux

Voici ci-dessous la syntaxe recommandée pour requêter un fichier parquet volumineux :

```
1 # Établir la connexion aux données
2 RA2020 <- open_dataset("data/RA2020_exploitations.parquet") |>
3   filter(SIEGE_REG == "93") |>
4   group_by(SIEGE_DEP) |>
5   summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE)) |>
6   collect()
```

=> Avec cette syntaxe, la requête va automatiquement utiliser les variables du fichier Parquet dont elle a besoin (en l'occurrence SIEGE_REG, SIEGE_DEP et SAU_TOT) et minimiser l'occupation de la mémoire vive.

Revenons dans le détail sur cette syntaxe...

2.5 La fonction `open_dataset()` (1/4)

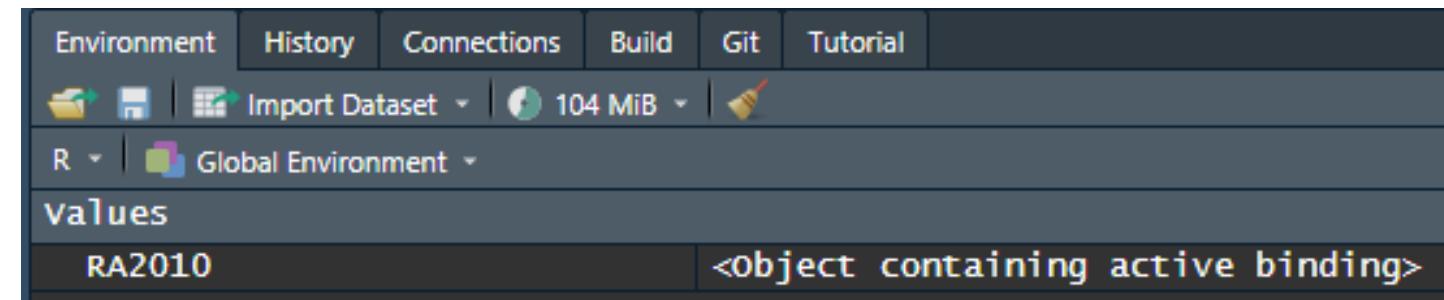
Comme la fonction `read_parquet()`, la fonction `open_dataset()` permet de lire des données stockées en format Parquet.

Le résultat obtenu avec la fonction `open_dataset()` n'est plus un **data.frame** mais un **Arrow Table** qui est une structure de données spécifique.

```
1 RA2020 <- open_dataset("data/RA2020_exploitations.parquet")
2
3 class(RA2020)
4
5 > [1] "FileSystemDataset" "Dataset" "ArrowObject" "R6"
```

2.6 La fonction `open_dataset()` (2/4)

La fonction `open_dataset()` crée un objet qui apparaît dans Values.



L'affichage dans la console d'un Arrow Table affiche uniquement les métadonnées.

```

1 RA2020
2
3 > FileSystemDataset with 1 Parquet file
4 NOM_DOSSIER: string
5 TYPE_QUESTIONNAIRE: string
6 SEUIL_IFS: string
7 CHAMP_GEO: string
8 COEF_F: double
9 NUMSTRATE: string
10 STRATE: string
11 SIEGENAT: string
12 SIEGE_CODE_COM: string
13 SIEGE_LIEUDIT: string
14 SIEGE_LIEUDIT_CODE_DOM: string
15 SIEGE_LIB_COM: string
16 ...

```

2.7 La fonction `open_dataset()` (3/4)

Pour afficher le contenu d'un Arrow Table, il faut d'abord le convertir en data.frame avec la fonction `collect()`.

```

1 RA2020 <- RA2020 |> collect()
2
3 class(RA2020)
4
5 > [1] "data.frame"
6
7 # L'opération ci-dessus est à éviter pour des tables volumineuses, si besoin de visualiser la table, on préférera :
8
9 extrait_RA2020 <- RA2020 |> slice_head(n = 100) |> collect()

```

Toutefois rien ne presse car la grande différence entre manipuler un data.frame et un Arrow Table tient au **moteur d'exécution** :

- Si on manipule un data.frame avec la syntaxe de dplyr, alors c'est le **moteur d'exécution de dplyr** qui fait les calculs 
- Si on manipule un Arrow Table avec la syntaxe de dplyr, alors c'est le **moteur d'exécution d'arrow** (nommé ~~acero~~) qui fait les calculs. Et le moteur d'exécution d'arrow est beaucoup plus efficace et rapide 

2.8 La fonction `open_dataset()` (4/4)

Il est recommandé de privilégier la fonction `open_dataset()` à la fonction `read_parquet()` pour au moins 2 raisons :

- `open_dataset()` crée une connexion au fichier Parquet mais elle n'importe pas les données contenues dans ce fichier => **une consommation de RAM moins importante !**
- `open_dataset()` peut se connecter à un fichier Parquet unique mais aussi à des **fichiers Parquets partitionnés** (voir plus loin)

2.9 Consulter les métadonnées d'un fichier Parquet (1/3)

Pour obtenir des informations générales sur le fichier (par exemple titre, auteur, date...), il faut utiliser la fonction `read_parquet()` avec l'argument `as_data_frame = FALSE` pour pouvoir accéder aux métadonnées globales via `$schema$metadata`.

```
1 # Lire le fichier Parquet en tant qu'Arrow Table avec read_parquet()
2 arrow_table <- arrow::read_parquet("output.parquet", as_data_frame = FALSE)
3
4 # Accéder aux métadonnées du fichier
5 arrow_table$schema$metadata
```

2.10 Consulter les métadonnées d'un fichier Parquet (2/3)

Le package `{nanoparquet}` permet aussi d'obtenir d'autres infos facilement depuis des fichiers parquet :

```

1 library(nanoparquet)
2
3 parquet_info("data/RA2020_exploitations.parquet")
4
5 # > # A data frame: 1 × 7
6   file_name           num_cols num_rows num_row_groups file_size parquet_version created_by
7   <chr>                <int>    <dbl>        <int>    <dbl>          <int> <chr>
8 1 data/RA2020_exploitations.parquet     255     416478         1  39896331          2 parquet-cpp-arrow version 9.0.0

```

2.11 Consulter les métadonnées d'un fichier Parquet (3/3)

- Les types des colonnes :

```

1 nanoparquet::parquet_column_types("data/output.parquet")
2
3 # A data frame: 4 × 6
4 #   file_name    name      type     r_type   repetition_type logical_type
5 # * <chr>        <chr>    <chr>    <chr>       <I<list>>
6 #   1 test.parquet colonne1 DOUBLE    double  REQUIRED      <NULL>
7 #   2 test.parquet colonne2 BYTE_ARRAY character REQUIRED      <STRING>
8 #   3 test.parquet colonne3 BOOLEAN   logical REQUIRED      <NULL>
9 #   4 test.parquet colonne4 INT32     Date    REQUIRED      <DATE>

```

Attention, Parquet propose **2 niveaux de type** :

- Le **bas niveau**
- Le **logical type**

La fonction **parquet_column_types()** retourne les types de **bas niveau** dans la colonne **type** et les **logical_type** dans la colonne **logical_type**.

- Pour aller plus loin sur les métadonnées :

```

1 parquet_metadata("data/RA2020_exploitations.parquet")

```

2.12 L'évaluation/exécution différée (1/4)

Cela signifie qu'**arrow** se contente de mémoriser les instructions, sans faire aucun calcul tant que l'utilisateur ne le demande pas explicitement.

Il existe **2 fonctions pour déclencher l'évaluation d'un traitement arrow** mais qui présente des différences :

- `collect()` qui renvoie le résultat du traitement sous la forme d'un **data.frame/tibble**
- `compute()` qui renvoie le résultat du traitement sous la forme d'un **Arrow Table**.

La grande différence entre manipuler un tibble et manipuler un Arrow Table tient au moteur d'exécution :

- Avec un **data.frame/tibble** => moteur d'exécution de dplyr
- Avec un **Arrow Table** => moteur d'exécution d'arrow (nommé acero), plus efficace que celui de dplyr

Dans les traitements intermédiaires, on privilégiera la fonction `compute()` pour pouvoir utiliser le plus possible le moteur acero.

2.13 L'évaluation/exécution différée (2/4)

```

1 SAU_DEP <- RA2020 |>
2   group_by(SIEGE_DEP) |>
3   summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE))
4 class(SAU_DEP)
5 > [1] "arrow_dplyr_query"
6
7 resultats <- SAU_DEP |>
8   filter(SIEGE_DEP == "13") |>
9   collect()
10 > # A tibble: 1 × 2
11   SIEGE_DEP total_SAU
12   <chr>      <dbl>
13   13          145713.

```

Dans l'exemple ci-dessus, **la première étape ne réalise aucun calcul par elle-même, car elle ne comprend ni collect() ni compute()**. L'objet `SAU_DEP` n'est pas une table et ne contient pas de données, il contient simplement une requête (query) décrivant les opérations à mener sur la table du RA.

arrow analyse la requête avant de l'exécuter, et optimise le traitement pour minimiser le travail.

Dans notre exemple, arrow repère que la requête ne porte en fait que sur le département 13, et commence donc par filtrer les données sur le département avant de sommer la SAU les équipements, de façon à ne conserver que le minimum de données nécessaires et à ne réaliser que le minimum de calculs.

2.14 L'évaluation/exécution différée (3/4)

L'évaluation/exécution différée est très puissante mais présente des limites.

On serait tentés d'écrire un traitement entier *en mode lazy* (sans aucun `compute()` ni `collect()` dans les étapes intermédiaires) et de faire un unique `compute()` ou `collect()` tout à la fin du traitement afin que toutes les opérations soient optimisées en une seule étape.

Malheureusement, le moteur acero a ses limites notamment sur des traitements trop complexes (ce qui génère des plantages de sessions R).

2.15 L'évaluation/exécution différée (4/4)

QUELQUES CONSEILS POUR ÉLABORER LA BONNE STRATÉGIE AVEC L'ÉVALUATION DIFFÉRÉE :

- Décomposer le traitement en plusieurs étapes puis exécuter chaque étape séparément (avec un `compute()`)
- Définir la bonne longueur des étapes intermédiaires en gardant en tête :
 - D'avoir des étapes de traitement qui **ne dépassent pas 40 lignes de code**
 - Que le séquencement des étapes soit **cohérent** avec l'objet du traitement
 - Plus les **données sont volumineuses** OU les **opérations unitaires sont complexes**, plus les étapes de traitement doivent être **courtes/prudentes**

2.16 Quelques manques sur le moteur acero (1/2)

La liste des fonctions du *tidyverse* supportées par acero est disponible [sur cette page](#).

Il y a (encore) quelques grands absents, notamment :

- `pivot_wider()` et `pivot_longer()` n'ont pas d'équivalent avec **acero**.
- **les empilements de plusieurs tables avec une seule fonction** (`bind_rows()` dans `dplyr`).

Avec des Arrow Tables, il faut appeler plusieurs fois ces fonctions (en l'occurrence `union()`). Par exemple :

```
1 resultats <- table1 |>
2   union(table2) |>
3   union(table3) |>
4   compute()
```

2.17 Quelques manques sur le moteur acero (2/2)

- les *window functions* (ajouter à une table des informations issues d'une agrégation par groupe) comme par exemple :

```
1 res <- RA2020 |>
2   group_by(SIEGE_REG) |>
3   mutate(total_SAU = sum(SAU_TOT)) |>
4   collect()
5
6 > Error: window functions not currently supported in Arrow
7 Call collect() first to pull data into R.
```

Remarque : le code ci-dessus fonctionne par contre en remplaçant le `mutate()` par un `summarise()`.

2.18 Comment contourner le problème d'acero ? (1/3)

Plusieurs solutions existent :

1. Comme suggéré par R, renoncer à manipuler les données sous forme d'Arrow Table avec le moteur acero en passant par un `collect()` et poursuivre le traitement avec le moteur d'exécution de dplyr (avec des performances moins importantes).
2. Étudier le message d'erreur renvoyé par R et chercher à réécrire d'une autre façon le traitement.

2.19 Comment contourner le problème d'acero ? (2/3)

Exemple pour le point 2 issu d'utilitr :

```

1 resultats <- bpe_ens_2018_arrow |>
2   group_by(DEP) |>
3   summarise(
4     nb_boulangeries = sum(NB_EQUIP * (TYPEQU == "B203")),
5     nb_poissonneries = sum(NB_EQUIP * (TYPEQU == "B206"))
6   ) |>
7   compute()
8
9 > ! NotImplemented: Function 'multiply_checked' has no kernel matching input types (double, bool); pulling data into R

```

L'erreur vient de l'opération `sum(NB_EQUIP * (TYPEQU == “B203”))`: **arrow** ne parvient pas à faire la multiplication entre `NB_EQUIP` (un nombre réel) et `(TYPEQU == “B203”)` (un booléen).

2.20 Comment contourner le problème d'acero ? (3/3)

=> La solution est très simple: il suffit de convertir (TYPEQU == “B203”) en nombre entier avec la fonction **as.integer()** qui est supportée par acero.

Le code suivant peut alors être entièrement exécuté par acero:

```

1 resultats <- bpe_ens_2018_arrow |>
2   group_by(DEP) |>
3   summarise(
4     nb_boulangeries = sum(NB_EQUIP * as.integer(TYPEQU == "B203")),
5     nb_poissonneries = sum(NB_EQUIP * as.integer(TYPEQU == "B206"))
6   ) |>
7   compute()

```

2.21 En conclusion sur le package arrow

Le package arrow présente 3 avantages majeurs :

- **Performances élevées** : arrow est très efficace et très rapide pour la manipulation de données tabulaires (nettement plus performant que dplyr par exemple)
- **Usage réduit des ressources** : arrow est conçu pour ne charger en mémoire que le minimum de données. Cela permet de réduire considérablement les besoins en mémoire, même lorsque les données sont volumineuses
- **Facilité d'apprentissage grâce aux approches dplyr et SQL**: arrow peut être utilisé avec les verbes de dplyr (select, mutate, etc.) et/ou avec le langage SQL grâce à DuckDB (voir plus loin).

2.22 Exercice 1

Exercice 1 (premiers contacts avec un fichier parquet + rappels sur les fonctions)

- Consulter les types de colonne de ce fichier
 - Ouvrir le fichier parquet situé sous [~/CERISE/03-Espace-de-Diffusion/030_Structures_exploitations/3020_Recensements/RA_2020/01_BASES_DIFFUSION_RA2020/RA_2020_parquet/RA2020_EXPLOITATIONS_240112.parquet](#)
 - Consulter les 100 premières lignes de ce fichier
 - Récupérer dans un vecteur trié les codes régions des lieux principaux de production (SIEGE_REG)
 - Récupérer dans un vecteur trié les libellés régions des lieux principaux de production (SIEGE_LIB_REG)
 - Ecrire une fonction calculs_RA() qui - pour une région et une table donnée en entrée - conserve uniquement les lignes correspondantes selon la colonne SIEGE_REG, puis groupe la table par SIEGE_DEP et calcule la surface totale SAU (SAU_TOT), la surface totale de céréales (CEREALES_SUR_TOT) et la surface totale d'oléagineux (OLEAG_SUR_TOT) et enfin la part de la surface des céréales dans la SAU totale et la part de la surface des oléagineux dans la SAU totale.
 - Utiliser ensuite la fonction calculs_RA() pour calculer ces indicateurs sur l'ensemble des régions présentes dans la table du RA2020 et stocker les résultats dans des fichiers Excel sous votre espace personnel.
- TIPS : pensez à utiliser {purrr} et {openxlsx} par exemple.*

2.23 Exercice 2

Exercice 2 (`collect()` vs `compute()`)

- Dans votre espace de travail, créer les 2 fichiers parquet suivants :

```

1 data_a <- tibble(
2   id = rep(1:1000000, each = 10),
3   annee = rep(2016:2025, times = 1000000),
4   a = sample(letters, 10000000, replace = TRUE)
5 )
6
7 data_b <- tibble(
8   id = rep(1:1000000, each = 10),
9   annee = rep(2016:2025, times = 1000000),
10  b = runif(10000000, 1, 100)
11 )
12
13 data_c <- tibble(
14   lettres = sample(letters, 10000000, replace = TRUE),
15   classe = sample(c("pommes","poires","melon","fraise"), 10000000, replace = TRUE)
16 )
17
18 write_parquet(data_a, "data_a.parquet")
19 write_parquet(data_b, "data_b.parquet")
20 write_parquet(data_c, "data_c.parquet")
21
22 rm(data_a)
23 rm(data_b)
24 rm(data_c)
25 gc()

```

- La suite de l'exercice sur la slide suivante...

2.24 Exercice 2 (suite)

Exercice 2 (`collect()` vs `compute()`)

a. AVEC `collect()`

Charger les fichiers parquet `data_a` et `data_b` sous forme de data.frame

Créer la table `etape1` en réalisant une jointure à gauche de `data_a` avec `data_b`.

Charger le fichier parquet `data_c` sous forme de data.frame

Filtrer la table `etape1` sur les années supérieures à 2020 puis faire la somme de la colonne `b` selon la colonne `a`

Ajouter la colonne `classe` issue de la table `data_c` dans le tableau final.

b. AVEC `compute()`

Réaliser les mêmes traitements que A) avec des `compute()` et réduire le temps d'exécution.

3 Écrire des fichiers parquet

3.1 Données peu volumineuses: écrire un seul fichier Parquet (1/2)

En tant que responsable de sources, vous pouvez être amenés à écrire et déposer des fichiers Parquet, par exemple sous Cerise.

Pour cela, on utilise la fonction `write_parquet()`.

Un 1er exemple simple à partir d'un fichier rds:

```
1 # Lecture du fichier rds
2 msa_ns <- readRDS("data/msa_ns_src_2023.rds")
3
4 # Écriture des données en format Parquet
5 write_parquet(x = msa_ns, sink = "data/msa_ns_src_2023.parquet")
```

	 msa_ns_src_2023.parquet	90.3 MB	Jan 10, 2025, 4:21 PM
	 msa_ns_src_2023.rds	79.6 MB	Jan 10, 2025, 4:19 PM

3.2 Données peu volumineuses: écrire un seul fichier Parquet (2/2)

Un autre exemple un peu plus compliqué à partir de fichier csv contenu dans un zip sur internet :

```
1 # Chargement des packages
2 library(arrow)
3 library(readr)
4
5 # Téléchargement du fichier zip
6 download.file("https://www.insee.fr/fr/statistiques/fichier/2540004/dpt2021_csv.zip", destfile = "data/dpt2021_csv.zip")
7 # Décompression du fichier zip
8 unzip("data/dpt2021_csv.zip", exdir = "data")
9
10 # Lecture du fichier CSV
11 dpt2021 <- read_delim(file = "data/dpt2021.csv")
12
13 # Écriture des données en format Parquet
14 write_parquet(x = dpt2021, sink = "data/dpt2021.parquet")
```

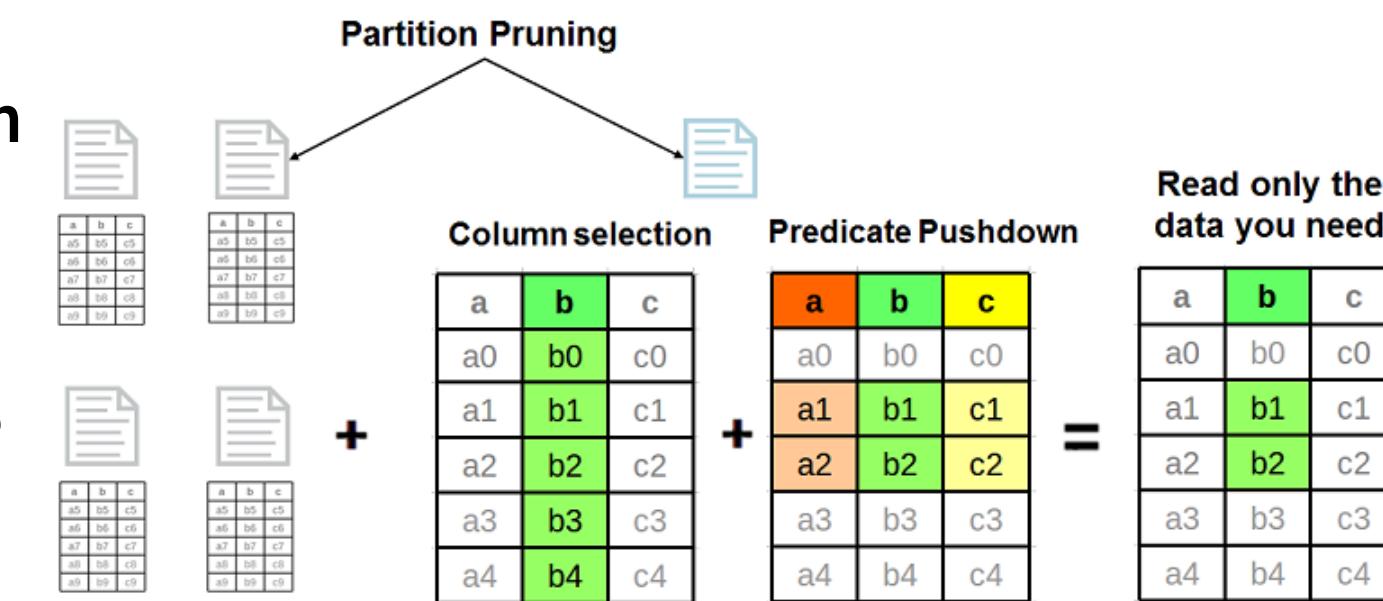
3.3 Données volumineuses: écrire un fichier Parquet partitionné (1/3)

Pourquoi partitionner ?

Par définition, il n'est pas possible de charger seulement quelques lignes d'un fichier Parquet : **on importe nécessairement des colonnes entières.**

Lorsque le fichier Parquet est partitionné, **arrow est capable de filtrer les lignes à importer à l'aide de clés de partitionnement**, ce qui permet d'accélérer l'importation des données.

Le partitionnement permet de travailler sur des fichiers Parquet de plus petite taille et donc de consommer moins de mémoire vive.



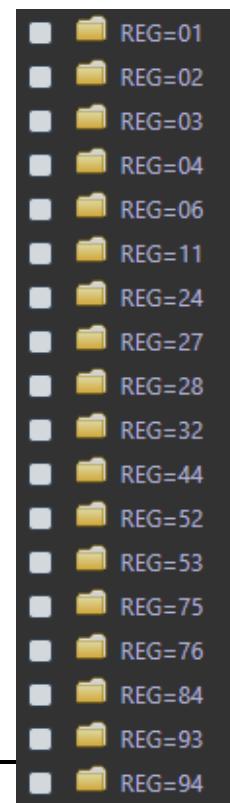
3.4 Données volumineuses: écrire un fichier Parquet partitionné (2/3)

Ça veut dire quoi partitionné ?

Partitionner un fichier revient à le “découper” selon une clé de partitionnement (une ou plusieurs variables)

En pratique, l'ensemble des données sera stockée dans plusieurs fichiers au format Parquet.

Voici par exemple comment se présente un fichier Parquet partitionné selon les régions :



3.5 Données volumineuses: écrire un fichier Parquet partitionné (3/3)

Pour écrire des fichiers Parquet partitionnés, on utilise la fonction `write_dataset()`.

Partitionnons notre fichier issu de la MSA par type d'exploitation et sexe :

```
1 write_dataset(
2   dataset = msa_ns,
3   path = "data/msa_ns",
4   partitioning = c("TYPE_EXP", "SEXE"), # les variables de partitionnement
5   format = "parquet"
6 )
```

Voici un aperçu de l'arborescence créée (:

```
1 data/msa_ns
2   └── TYPE_EXP=1
3     ├── SEXE=1
4     |   └── part-0.parquet
5     └── SEXE=2
6       └── part-0.parquet
7   └── TYPE_EXP=2
8     ├── SEXE=1
9     |   └── part-0.parquet
10    └── SEXE=2
11      └── part-0.parquet
12   └── TYPE_EXP=3
13     ├── SEXE=1
14     |   └── part-0.parquet
15     └── SEXE=2
16       └── part-0.parquet
```

3.6 Forcer les types des colonnes lors de l'écriture de fichiers Parquet

En tant que responsable de sources par exemple, vous pouvez forcer le typage des colonnes d'un fichier Parquet.
Pour cela, assurez-vous que les colonnes du data.frame R sont au bon type.
Si ce n'est pas le cas, utilisez les fonctions de conversion `as.character()`, `as.integer()`, `as.Date()` ...

```
1 #Créez un DataFrame R en spécifiant le type de chaque colonne
2 df <- data.frame(
3   colonne1 = as.integer(c(1, 2, 3)),
4   colonne2 = as.character(c("A", "B", "C")),
5   colonne3 = as.logical(c(TRUE, FALSE, TRUE)),
6   colonne4 = as.numeric(35.43, 29.93, 17.02))
7
8 write_parquet(table, "output.parquet")
```

Une autre façon de forcer le typage des colonnes est d'utiliser un [schema](#). Cependant, cela apporte de la complexité au code pas forcément utile.

~~En cas de besoin, un exemple est disponible sur [cette page](#).~~

3.7 Ajouter des métadonnées générales sur un fichier Parquet (1/3)

A partir d'un data.frame R et avant l'écriture du fichier Parquet correspondant, il est possible d'ajouter des métadonnées générales sur un fichier Parquet.

Soit le data.frame suivant :

```
1 df <- data.frame(  
2   colonne1 = c(1, 2, 3),  
3   colonne2 = c("A", "B", "C"),  
4   colonne3 = c(TRUE, FALSE, TRUE),  
5   colonne4 = as.Date(c("2025-01-01", "2025-01-02", "2025-01-03")))
```

3.8 Ajouter des métadonnées générales sur un fichier Parquet (2/3)

Avec uniquement le package {arrow}, on peut ajouter des métadonnées comme ceci :

```
1 # Conversion en Arrow Table
2 df_arrow <- arrow_table(df)
3
4 # Ajouter des métadonnées
5 df_arrow <- df_arrow$ReplaceSchemaMetadata(c(
6   auteur = "DEMESIS/BQIS",
7   description = "Table test de formation",
8   date_creation = as.character(Sys.Date())
9 ))
10
11 write_parquet(df_arrow, "data/output.parquet")
```

3.9 Ajouter des métadonnées générales sur un fichier Parquet (3/3)

Avec le package {nanoparquet}, le code est plus court.

Attention ici, la fonction `write_parquet()` utilisée est issue du package {nanoparquet} et non {arrow} (qui ne propose pas l'argument `metadata` et dont les noms des autres arguments sont différents).

```
1 nanoparquet::write_parquet(  
2   x = df,  
3   file = "data/output.parquet",  
4   metadata = c(  
5     auteur = "DEMESIS/BQIS",  
6     description = "Table test de formation",  
7     date_creation = as.character(Sys.Date()))  
8 )
```

3.10 Industrialiser la conversion de vos fichiers ?

- Le package R **parquetize** permet de faciliter la conversion de données au format Parquet.
- Plusieurs formats supportés csv, json, rds, fst, SAS, SPSS, Stata, sqlite...
- Propose des **solutions de contournement** pour les fichiers très volumineux.

Un exemple issu de la documentation :

```

1 Conversion from a local rds file to a partitioned parquet file :: 12
2 rds_to_parquet(
3   path_to_file = system.file("extdata","iris.rds",package = "parquetize"),
4   path_to_parquet = tempfile(fileext = ".parquet"),
5   partition = "yes",
6   partitioning = c("Species")
7 )
8
9 #> Reading data...
10 #> Writing data...
11 #> ✓ Data are available in parquet dataset under /tmp/RtmpNiaDm/file1897441ca0c0.parquet
12 #> Writing data...
13
14 #> Reading data...

```

3.11 Comment bien utiliser des fichiers partitionnés avec arrow (1/2)

La fonction `open_dataset()` permet d'ouvrir une connexion vers un fichier Parquet partitionné.

L'utilisation de la fonction `open_dataset()` est similaire au cas dans lequel on travaille avec un seul fichier Parquet.

Il y a toutefois 2 différences :

- Le chemin indiqué n'est pas celui d'un fichier .parquet, mais le chemin d'un répertoire dans lequel se trouve le fichier Parquet partitionné Il est préférable d'indiquer le nom et le type de la ou des variable(s) de partitionnement.

3.12 Comment bien utiliser des fichiers partitionnés avec arrow (2/2)

Un exemple avec les données de la MSA :

```

1 # Etablir la connexion au fichier Parquet partitionné
2 donnees_msa <- open_dataset(
3   "data/msa_ns", # Ici, on met le chemin d'un répertoire
4   hive_style = TRUE,
5   partitioning = arrow::schema(TYPE_EXP = arrow::utf8(), SEXE = arrow::utf8()) # Les variables de partitionnement
6 )
7
8 # Définir la requête
9 resultats_msa <- donnees_msa |>
10  filter(TYPE_EXP == "2" & SEXE == "1") |> # Ici, on filtre selon les clés de partitionnement
11  select(DEPT, RC_CHEF) |>
12  collect()

```

Ce qui donne :

```

1 resultats_msa
2
3 > resultats_msa
4 # A tibble: 62,195 × 2
5   DEPT    RC_CHEF
6   * <chr> <chr>
7   1 11    1461
8   2 11    2910
9   3 11    1528
10  4 11    4493

```

3.13 Conseils lors de l'utilisation de fichiers partitionnés

Afin de tirer au mieux profit du partitionnement, il est conseillé de **filtrer les données de préférence selon les variables de partitionnement** (dans notre exemple, `TYP_EXP` et `SEX`).

Il est fortement recommandé de **spécifier le type des variables de partitionnement avec l'argument `partitioning`**.

Cela évite des erreurs typiques: le code du département est interprété à tort comme un nombre et aboutit à une erreur à cause de la Corse...

L'argument `partitioning` s'utilise en construisant un schéma qui précise le type de chacune des variables de partitionnement.

Voir [cette page](#) pour la liste des types supportés.



*Liberté
Égalité
Fraternité*

3.14 Dernier conseil avec arrow

Il est recommandé de définir les deux options suivantes au début de votre script.

Cela autorise arrow à utiliser plusieurs processeurs à la fois, ce qui accélère les traitements :

```
1 Autoriser arrow à utiliser plusieurs processeurs en même temps
2 options(arrow.use_threads = TRUE)
3
4 # Définir le nombre de processeurs utilisés par arrow
5 # 10 processeurs sont suffisants dans la plupart des cas
6 arrow:::set_cpu_count(parallel::detectCores() %% 2)
```

4 Manipuler des fichiers parquets avec duckdb

4.1 Qu'est-ce que duckdb ?

DuckDB est un projet open-source qui propose un moteur SQL optimisé pour **réaliser des travaux d'analyse statistique sur des bases de données**.

Plusieurs avantages :

- Un **moteur portable** utilisable avec **plusieurs langages** (R, Python, Javascript...) et **plusieurs OS** (Windows, Linux, MacOS...)
- Une **installation** et une **utilisation** très facile
- Un **moteur SQL** capable d'utiliser des données au format Parquet **sans les charger complètement en mémoire.**



Il faut bien distinguer le projet DuckDB du package R duckdb qui propose simplement une façon d'utiliser Duckdb avec R.

4.2 À quoi sert le package duckdb + installation ?

Du point de vue d'un statisticien utilisant R, le package duckdb permet de faire trois choses :

- Importer des données (exemples: fichiers CSV, fichiers Parquet)
- Manipuler des données avec la syntaxe dplyr, ou avec le langage SQL
- Écrire des données au format Parquet.

Installation de duckdb

Il suffit d'installer le package duckdb, qui contient à la fois DuckDB et une interface pour que R puisse s'y connecter.

Bonne nouvelle sur la version de Cerise mis à disposition en 2025, le package duckdb sera installé par défaut !

4.3 Connexions avec duckdb

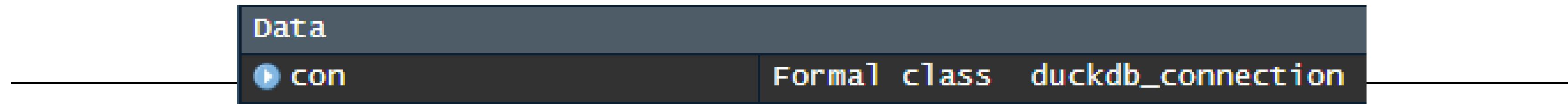
Pour utiliser duckdb, il n'est pas nécessaire de connaître le langage SQL car il est possible d'utiliser duckdb avec la syntaxe dplyr.

duckdb est une base de données distante et s'utilise comme telle : il faut ouvrir une connexion, puis "charger" les données dans la base de données pour les manipuler.

```
1 con <- DBI::dbConnect(drv = duckdb::duckdb())
```

Plusieurs remarques :

- Cette commande crée une nouvelle base de données duckdb **dans la mémoire vive**.
- Cette base de données ne contient aucune donnée lorsqu'elle est créée. L'objet `con` apparaît dans l'onglet Data de l'environnement RStudio, mais la liste des tables n'y est pas directement accessible





*Liberté
Égalité
Fraternité*

4.4 Déconnexion avec duckdb

À la fin du traitement ou du programme, on ferme la connexion avec le code ci-dessous. L'option `shutdown` est importante : elle permet de fermer complètement la session duckdb et de libérer la mémoire utilisée.

```
1 DBI::dbDisconnect(con, shutdown = TRUE)
```

! Important

Si on n'utilise pas cette option, il arrive souvent que des connexions à moitié ouvertes continuent à consommer des ressources !!!



4.5 Chargement de données issues de la session R

La fonction `duckdb_register()` permet de charger dans duckdb des données présentes en mémoire dans la session R.

Cette méthode a l'avantage de ne pas recopier les données: elle se contente d'établir un lien logique entre la base de données duckdb et un objet de la session R.

L'objet créé dans la base est une vue dans le catalogue “temp”. La durée d'existence de cette vue est le temps de la connexion.

```
1 # Création de la vue "RA2020_duckdb"
2 con |> duckdb::duckdb_register(
3   name = "RA2020_duckdb",
4   df = RA2020)
```

4.6 Accéder à une table d'une base DuckDB

Pour vérifier que le chargement des données a bien fonctionné, la fonction `tbl()` permet d'accéder à une table de la base de données grâce à son nom (entre double quotes).

```

1 con |> tbl("RA2020_duckdb")
2
3 # Source:   table<RA2020_duckdb> [?? x 255]
4 # Database: DuckDB v1.1.0 [damien.dotta@Windows 10 x64:R 4.3.0/:memory:]
5   NOM_DOSSIER  TYPE_QUESTIONNAIRE  SEUIL_IFS  CHAMP_GEO  COEF_F  NUMSTRATE  STRATE    SIEGENAT  SIEGE_CODE_COM  SIEGE_LIEUDIT
6   <chr>        <chr>          <chr>      <chr>      <dbl> <chr>       <chr>      <chr>     <chr>           <chr>
7 1 AAAAAAAA    2               1           1         1.22 2500000004 25000ERICA 10    14406        NA
8 2 BBBBBBBB    2               1           1         10.3 2105161843 2105161843 10   51303        NA
9 3 CCCCCCCC    3               1           2         1   0101030406 EXH_GEO      10   97407 CHEMIN DES ANGLAIS

```

4.7 Chargement de données sur le disque au format Parquet (1/2)

Pour charger des données situés sur Cerise par exemple, la fonction `tbl()` peut aussi être directement utilisée en renseignant le chemin du fichier Parquet.

```

1 con %>%tbl("read_parquet('data/RA2020_exploitations.parquet')")
2 # Ou plus succinct (pas besoin de read_parquet() si pas besoin d'y passer des arguments)
3 con %>%tbl('data/RA2020_exploitations.parquet')
4
5 # Source: SQL [?? x 255]
6 # Database: DuckDB v1.1.0 [damien.dotta@Windows 10 x64:R 4.3.0/:memory:]
7 NOM_DOSSIER TYPE_QUESTIONNAIRE SEUIL_IFS CHAMP_GEO COEF_F NUMSTRATE STRATE SIEGENAT SIEGE_CODE_COM SIEGE_LIEUDIT
8 <chr> <chr> <chr> <chr> <dbl> <chr> <chr> <chr> <chr>
9 1 AAAAAAA 2 1 1 1.22 2500000004 25000ERICA 10 14406 NA
10 2 BBBB BBBB 2 1 1 10.3 2105161843 2105161843 10 51303 NA
11 3 CCCCCC 3 1 2 1 0101030406 EXH_GEO 10 97407 CHEMIN DES ANGLAIS

```



4.8 Chargement de données sur le disque au format Parquet (2/2)

Avec des fichiers partitionnés, on peut utiliser la syntaxe suivante.

`**/* .parquet` est un motif qui indique que vous souhaitez lire, dans tous les sous-dossiers quelque soit le niveau (**), l'ensemble des fichiers parquets (* .parquet) qui s'y trouvent.

```

1 con %>%tbl('data/msa_ns/**/*.parquet')
2
3 # Source: SQL [?? x 293]
4 # Database: DuckDB v1.1.0 [damien.dotta@Windows 10 x64:R 4.3.0/:memory:]
5 A_IDENT      DEPT  RESID CANTON COMMUNE DOM_FISC SUP   NAF      SMI  SITU_FAM NAIS_JO NAIS_MO NAIS_AN AF_MAJ AF_MIN AF_AVA
6 <chr>        <chr> <chr> <chr>  <chr>    <dbl> <chr> <dbl>  <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
7 1 XXXXXXXXXXXXXX... 11    11    10    288      1 0    NA    0     1 25    05    1969    0    0    0
8 2 YYYYYYYYYYYYYY... 11    11    01    281      1 2903  NA    0     1 30    01    2000    0    0    0
9 3 ZZZZZZZZZZZZZZ... 11    11    16    269      1 368  NA    0     1 07    01    2000    0    0    0

```



4.9 Afficher la liste des tables d'une base Duckdb (1/2)

La fonction `dbListTables()` est utile pour afficher les noms des tables présentes dans une base de données.

Une illustration pour mieux comprendre :

```

1 # On se déconnecte
2 DBI::dbDisconnect(con, shutdown = TRUE)
3
4 # On crée une nouvelle connexion
5 con <- DBI::dbConnect(drv = duckdb::duckdb())
6
7 # Affichage de la liste des tables
8 dbListTables(con)
9 # > character(0)
10
11 con |> duckdb::duckdb_register(
12   name = "iris_duckdb",
13   df = iris)
14
15 con %>% tbl("read_parquet('data/RA2020_exploitations.parquet')")

```

=> Question : combien de tables va maintenant renvoyer la fonction `dbListTables()` ?



4.10 Afficher la liste des tables d'une base Duckdb (2/2)

Réponse... UNE SEULE !

```
1 # Affichage de la liste des tables
2 dbListTables(con)
3 # > "iris_duckdb"
4
5 # On se déconnecte
6 DBI::dbDisconnect(con, shutdown = TRUE)
```

En effet, lorsqu'on utilise la fonction `tbl()`, celle-ci ne charge pas les données ni dans la mémoire de R ni dans celle de DuckDB.



Liberté
Égalité
Fraternité

4.11 Afficher la liste des colonnes d'une table d'une base Duckdb

La fonction `dbListFields()` est utile pour afficher les noms des colonnes d'une table présente dans une base de données.

```
1 # On crée une nouvelle connexion
2 con <- DBI::dbConnect(drv = duckdb::duckdb())
3
4 con |> duckdb::duckdb_register(
5   name = "iris_duckdb",
6   df = iris)
7
8 con |> DBI::dbListFields("iris_duckdb")
9 # > [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
10
11 # On se déconnecte
12 DBI::dbDisconnect(con, shutdown = TRUE)
```

4.12 Accéder aux logical_types des colonnes d'un fichier parquet avec duckdb

En complément des instructions déjà vues dans la partie 2 de la formation, on peut accéder aux logical_types des colonnes d'un fichier Parquet en utilisant la requête suivante :

```
1 con <- DBI::dbConnect(drv = duckdb::duckdb())
2 dbGetQuery(con, "DESCRIBE FROM read_parquet('data/output.parquet')")
3
4 # column_name column_type null key default extra
5 # 1    colonne1      DOUBLE  YES <NA>   <NA>   <NA>
6 # 2    colonne2      VARCHAR YES <NA>   <NA>   <NA>
7 # 3    colonne3      BOOLEAN YES <NA>   <NA>   <NA>
8 # 4    colonne4      DATE    YES <NA>   <NA>   <NA>
```

La requête suivant est très utile pour accéder aux méta-données d'un fichier Parquet avec duckdb :

Remarque : La fonction `dbGetQuery()` du package `DBI` permet de récupérer un `data.frame` dont le contenu est le résultat d'une requête.

4.13 Accéder aux métadonnées générales d'un fichier parquet avec duckdb

La fonction `parquet_metadata()` peut également être utilisée dans une requête écrite avec duckdb :

```
1 con <- DBI::dbConnect(drv = duckdb::duckdb())
2 dbGetQuery(con, "SELECT * FROM parquet_metadata('data/fr_immp_transactions.parquet')")
```

row_group_id	row_group_num_rows	row_group_num_colu...	row_group_bytes	column_id	file_off...	num_values	path_in_schema	type	stats_min	stats_max	stats_null_count	stats_distinct_count	stats_min_value	stats_max_value	compression
0	1048576	13	54792718	0	0	1048576	id_transaction	INT32	1	1985836	0	0	1	1985836	SNAPPY
0	1048576	13	54792718	1	0	1048576	date_transaction	INT64	2014-01-02 00:00:00	2024-06-29 00:00:00	0	0	2014-01-02 00:00:00	2024-06-29 00:00:00	SNAPPY
0	1048576	13	54792718	2	0	1048576	prix	DOUBLE	-0.0	641855000.0	0	0	-0.0	641855000.0	SNAPPY
0	1048576	13	54792718	3	0	1048576	departement	BYTE_ARRAY			0	0	01	14	SNAPPY
0	1048576	13	54792718	4	0	1048576	id_ville	INT32	1	834	0	0	1	834	SNAPPY
0	1048576	13	54792718	5	0	1048576	ville	BYTE_ARRAY			0	0	ABBELOURT	YZEURE	SNAPPY
0	1048576	13	54792718	6	0	1048576	code_postal	INT32	1000	48250	0	0	1000	48250	SNAPPY
0	1048576	13	54792718	7	0	1048576	adresse	BYTE_ARRAY			0	0	A CHANVES	9Z RUE GABRIEL VICAI...	SNAPPY
0	1048576	13	54792718	8	0	1048576	type_batiment	BYTE_ARRAY			0	0	Appartement	Maison	SNAPPY
0	1048576	13	54792718	9	0	1048576	n_pieces	INT32	0	90	0	0	0	90	SNAPPY
0	1048576	13	54792718	10	0	1048576	surface_habitable	INT32	0	3640	0	0	0	3640	SNAPPY
0	1048576	13	54792718	11	0	1048576	latitude	DOUBLE	42.5861157352246	50.1638368446025	0	0	42.5861157352246	50.1638368446025	SNAPPY
0	1048576	13	54792718	12	0	1048576	longitude	DOUBLE	-1.14668362067864	7.68079585150826	0	0	-1.14668362067864	7.68079585150826	SNAPPY
1	1048576	13	54577651	0	0	1048576	id_transaction	INT32	1198799	3782564	0	0	1198799	3782564	SNAPPY
1	1048576	13	54577651	1	0	1048576	date_transaction	INT64	2014-01-01 00:00:00	2024-06-29 00:00:00	0	0	2014-01-01 00:00:00	2024-06-29 00:00:00	SNAPPY
1	1048576	13	54577651	2	0	1048576	prix	DOUBLE	-0.0	2086000000.0	0	0	-0.0	2086000000.0	SNAPPY
1	1048576	13	54577651	3	0	1048576	departement	BYTE_ARRAY			0	0	14	29	SNAPPY
1	1048576	13	54577651	4	0	1048576	id_ville	INT32	1	764	0	0	1	764	SNAPPY
1	1048576	13	54577651	5	0	1048576	ville	BYTE_ARRAY			0	0	ABBANS DESSOUS	YVRAC-ET-MALLEYRA...	SNAPPY
1	1048576	13	54577651	6	0	1048576	code_postal	INT32	5700	43450	0	0	5700	43450	SNAPPY
1	1048576	13	54577651	7	0	1048576	adresse	BYTE_ARRAY			0	0	A BERROTTE	9X RUE DE LA PARCHE...	SNAPPY
1	1048576	13	54577651	8	0	1048576	type_batiment	BYTE_ARRAY			0	0	Appartement	Maison	SNAPPY

=> On retrouve des informations très intéressantes sur le mode de compression utilisé lors de l'écriture du fichier Parquet (SNAPPY), des statistiques sur les row groups (min, max), l'encodage...



4.14 Requête avec dplyr

Le package R `duckdb` a été écrit de façon à pouvoir manipuler les données avec la syntaxe de `dplyr`, c'est très pratique !

Comme avec le package `{arrow}` on utilise cette syntaxe avec les fonctions `collect()` ou `compute()`.

```

1 # On crée une nouvelle connexion
2 con <- DBI::dbConnect(drv = duckdb::duckdb())
3
4 # Etablissement de la connexion au fichier Parquet
5 RA2020_dataset <- con %>% tbl('data/RA2020_exploitations.parquet')
6
7 # Traitement avec dplyr et un collect()
8 resultat <- RA2020_dataset |>
9   filter(SIEGE_REG == "93") |>
10  group_by(SIEGE_DEP) |>
11  summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE)) |>
12  collect()
13
14 # On se déconnecte
15 DBI::dbDisconnect(con, shutdown = TRUE)

```

```

1 > resultat
2 # A tibble: 6 × 2
3   SIEGE_DEP total_SAU
4   <chr>      <dbl>
5   1 84        112888.
6   2 05        91979.
7   3 04        158946.
8   4 13        145713.
9   5 83        77785.
10  6 06        41141.

```



Liberté
Égalité
Fraternité

4.15 Examen de la requête SQL construite par duckdb

Quand on manipule des objets avec duckdb, on construit des requêtes SQL.

Le package duckdb se contente de traduire le code dplyr en SQL.

La fonction `show_query()` permet de consulter la requête SQL qui a été exécutée par duckdb.

```
1 # Traitement avec dplyr et un collect()
2 résultat <- RA2020_dataset |>
3   filter(SIEGE_REG == "93") |>
4   group_by(SIEGE_DEP) |>
5   summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE)) |>
6   show_query()
7
8 # > <SQL>
9 SELECT SIEGE_DEP, SUM(SAU_TOT) AS total_SAU
10 FROM "data/RA2020_exploitations.parquet"
11 WHERE (SIEGE_REG = '93')
12 GROUP BY SIEGE_DEP
```



Liberté
Égalité
Fraternité

4.16 Requête avec SQL

Si vous avez des connaissances en SQL, il est bien sûr tout à fait possible de requêter une base DuckDB avec ce langage.

La requête est alors passée dans la fonction `DBI::dbGetQuery()` :

```
1 # On crée une nouvelle connexion
2 con <- DBI::dbConnect(drv = duckdb::duckdb())
3
4 chemin_donnees <- 'C:/Users/damien.dotta/DEMESIS/Formations_R/Formation_R_perfectionnement/formation-R-perf-06-parquet/data'
5
6 # Execution de la requete
7 resultatSQL <- DBI::dbGetQuery(
8   con,
9   paste0("SELECT SIEGE_DEP, SUM(SAU_TOT) AS total_SAU
10    FROM '", file.path(chemin_donnees, 'RA2020_exploitations.parquet'),"'
11   WHERE (SIEGE_REG = '93')
12   GROUP BY SIEGE_DEP"))
13 # resultatSQL est un data.frame directement utilisable dans R
```

4.17 Sauvegarder des résultats intermédiaires - dans des fichiers Parquet

Lorsque le traitement est long, vous devez le découper et stocker quelque part vos résultats intermédiaires de manière à ne pas tout recalculer entièrement à chaque fois.

1. Une 1ère solution consiste à écrire ces résultats intermédiaires dans des fichiers Parquet :

```

1 # On crée une nouvelle connexion
2 con <- DBI::dbConnect(drv = duckdb::duckdb())
3
4 # Etablissement de la connexion au fichier Parquet
5 RA2020_dataset <- con %>%tbl('data/RA2020_exploitations.parquet')
6
7 # A la fin du traitement, on écrit un fichier parquet intermédiaire
8 RA2020_dataset |>
9   filter(SIEGE_REG == "93") |>
10  # Conversion dans un format compatible avec arrow
11  arrow::to_arrow() |>
12  arrow::write_parquet("data/table_intermediaire.parquet")

```

```

1 # Reprise des traitements + tard avec le moteur SQL de duckdb
2 final <- arrow::open_dataset("data/table_intermediaire.parquet") |>
3   # Conversion dans un format compatible avec duckdb
4   arrow::to_duckdb(con) |>
5   group_by(SIEGE_DEP) |>
6   summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE)) |>
7   collect()
8
9 # On se déconnecte
10 DBI::dbDisconnect(con, shutdown = TRUE)

```

4.18 Sauvegarder des résultats intermédiaires - avec une BDD

2. Une 2ème solution consiste à écrire dans une BDD ces résultats intermédiaires :

```

1 # On crée une nouvelle connexion
2 # + création d'une BDD vide avec le paramètre dbdir
3 con <- DBI::dbConnect(drv = duckdb::duckdb(),
4                      dbdir = "data/BDD_formation.db")
5
6 # Etablissement de la connexion au fichier Parquet
7 RA2020_dataset <- con %>%tbl('data/RA2020_exploitations.parquet')
8
9 # Traitement avec dplyr et un compute()
10 RA2020_dataset |>
11   filter(SIEGE_REG == "93") |>
12   compute(name = "resultat_interm",
13           temporary = FALSE)
14
15 # suppression des tables créés
16 DBI::dbRemoveTable(con, "resultat_interm")

```

```

1 # Affichage de la liste des tables
2 dbListTables(con)
3 # [1] "resultat_interm"
4
5 # Reprise des traitements + tard avec le moteur SQL de duckdb
6 final2 <- con |>tbl("resultat_interm") |>
7   group_by(SIEGE_DEP) |>
8   summarise(total_SAU = sum(SAU_TOT, na.rm = TRUE)) |>
9   collect()
10
11 # On se déconnecte
12 DBI::dbDisconnect(con, shutdown = TRUE)

```



*Liberté
Égalité
Fraternité*

4.19 A retenir sur les 2 méthodes de sauvegarde des résultats intermédiaires

- La première méthode avec `{arrow}` et la sauvegarde dans des fichiers Parquet est généralement la plus rapide.
- La seconde avec `compute()` et la sauvegarde dans une BDD est la plus efficace en terme d'occupation mémoire.



*Liberté
Égalité
Fraternité*

4.20 Options à utiliser

Lors d'une connexion à une base, des paramètres de `dbConnect()` peuvent être très utiles lorsqu'on travaille sur un espace comme Cerise où les ressources sont partagées.

Par exemple :

```
1 con_dbnew <- dbConnect(  
2   drv = duckdb::duckdb(),  
3   dbdir=db_file,  
4   config=list("memory_limit"="10GB", # On définit un plafond de 10Go de RAM  
5             threads = "4")) # On plafonne l'utilise par duckdb de 4 coeurs (sur les 30 dispos sur Cerise)
```

4.21 Consulter le contenu d'une base

Reprendons la base `BDD_formation.db` créée précédemment.

```

1 # On crée une nouvelle connexion
2 con <- DBI::dbConnect(drv = duckdb::duckdb(),
3                         dbdir = "data/BDD_formation.db")
4
5 # Affichage de la liste des tables
6 dbListTables(con)
7 # > [1] "resultat_interm"

```

Ajoutons-y une vue :

```

1 # Création de la vue "RA2020_duckdb"
2 con |> duckdb::duckdb_register(
3   name = "RA2020_duckdb",
4   df = RA2020)

```

```

1 # Affichage de la liste des tables
2 dbListTables(con)
3 # > [1] "RA2020_duckdb"    "resultat_interm"

```

Qu'en est-il vraiment ?

```

1 DBI::dbGetQuery(con, "SELECT * FROM INFORMATION_SCHEMA.TABLES")
2 # table_catalog table_schema      table_name table_type
3 1 BDD_formation          main resultat_interm BASE TABLE
4 2                 temp      main RA2020_duckdb     VIEW

```

=> Après déconnexion, seule la table “resultat_interm” sera conservée dans la BDD.

```

1 # On se déconnecte
2 DBI::dbDisconnect(con, shutdown = TRUE)

```



4.22 Requêter des données Parquet extérieures (data.gouv) avec Cerise et duckdb (1/5)

Un exemple de code pour requêter directement la base Parquet des bureaux de vote directement depuis Cerise !!!

Configuration générale :

```

1 # Installation d'une version récente de duckdb
2 install.packages("duckdb", repos = "https://packagemanager.posit.co/cran/__linux__/centos8/latest")
3
4 library(duckdb) ; library(glue)
5 cnx <- dbConnect(duckdb())
6
7 # Vérification de la version du package
8 dbGetQuery(cnx, "select version()")
9
10 dbExecute(cnx, "
11   SET http_proxy='http://rie.proxy.national.agri:8080';
12   SET http_proxy_username='${AGRICOLL_U}';
13   SET http_proxy_password='${AGRICOLL_P}'")
14
15 # Installation et chargement de l'extension nécessaire httpfs
16 dbExecute(cnx, "INSTALL httpfs")
17 dbExecute(cnx, "LOAD httpfs")

```



*Liberté
Égalité
Fraternité*

4.23 Requêter des données Parquet extérieures (data.gouv) avec Cerise et duckdb (2/5)

Requête :

```
1 dataset <- "https://static.data.gouv.fr/resources/bureaux-de-vote-et-adresses-de-leurs-electeurs/20230626-135723/table-adresses-reu.parquet"
2
3 dbGetQuery(cnx, glue("DESCRIBE FROM '{dataset}'"))
4
5 dbDisconnect(cnx, shutdown = TRUE)
```



4.24 Requêter des données Parquet extérieures (data.gouv) avec Cerise et duckdb (3/5)

Le résultat de la requête :

	column_name	column_type	null	key	default	extra
1	code_commune_ref	VARCHAR	YES	<NA>	<NA>	<NA>
2	reconstitution_code_commune	VARCHAR	YES	<NA>	<NA>	<NA>
3	id_brut_bv_reu	VARCHAR	YES	<NA>	<NA>	<NA>
4	id	VARCHAR	YES	<NA>	<NA>	<NA>
5	geo_adresse	VARCHAR	YES	<NA>	<NA>	<NA>
6	geo_type	VARCHAR	YES	<NA>	<NA>	<NA>
7	geo_score	DOUBLE	YES	<NA>	<NA>	<NA>
8	longitude	DOUBLE	YES	<NA>	<NA>	<NA>
9	latitude	DOUBLE	YES	<NA>	<NA>	<NA>
10	api_line	VARCHAR	YES	<NA>	<NA>	<NA>
11	nb_bv_commune	UINTTEGER	YES	<NA>	<NA>	<NA>
12	nb_adresses	UINTTEGER	YES	<NA>	<NA>	<NA>

4.25 Requêter des données Parquet extérieures (data.gouv) avec Cerise et duckdb (4/5)

Autre exemple pour requêter la table Parquet des unités légales de SIRENE :

Requête à passer après configuration du proxy et chargement de l'extension https :

```

1 cnx <- dbConnect(duckdb())
2
3 dataset <- "https://static.data.gouv.fr/resources/base-sirene-des-entreprises-et-de-leurs-établissements-siren-siret/20231214-131836/stockuni"
4
5 req <- glue_sql(
6   CREATE TABLE extract_siren AS
7     FROM {dataset}
8     WHERE SIREN in ('345149587', '388467441')
9   ", .con = cnx)
10
11 dbSendQuery(cnx, req)
12
13 recuper_siren <- dbReadTable(cnx, "extract_siren")
14
15 dbDisconnect(cnx, shutdown = TRUE)

```

4.26 Requêter des données Parquet extérieures (data.gouv) avec Cerise et duckdb (4/4)

Le résultat de la requête :

```

1 # A tibble: 2 × 34
2   siren      statutDiffusionUniteLegale unitePurgeeUniteLegale dateCreationUniteLeg...¹ sigleUniteLegale sexeUniteLegale prenom1UniteLegale pren...
3   <chr>      <chr>                  <lgcl>                <date>                 <chr>          <chr>          <chr>          <chr>
4   1 345149587 0                      NA                   1985-01-01            INALCA          NA           NA           NA
5   2 388467441 0                      NA                   1992-09-01            NA             NA           NA           NA
6   # i abbreviated name: `¹dateCreationUniteLegale`
7   # i 25 more variables: prenom4UniteLegale <chr>, prenomUsuelUniteLegale <chr>, pseudonymeUniteLegale <chr>, identifiantAssociationUniteLegale <...
8   #   trancheEffectifsUniteLegale <chr>, anneeEffectifsUniteLegale <dbl>, dateDernierTraitementUniteLegale <dttm>, nombrePeriodesUniteLegale <...
9   #   anneeCategorieEntreprise <dbl>, dateDebut <date>, etatAdministratifUniteLegale <chr>, nomUniteLegale <chr>, nomUsageUniteLegale <chr>, de...
10  #   denominationUsuelle1UniteLegale <chr>, denominationUsuelle2UniteLegale <chr>, denominationUsuelle3UniteLegale <chr>, categorieJuridiqueUn...
11  #   activitePrincipaleUniteLegale <chr>, nomenclatureActivitePrincipaleUniteLegale <chr>, nicSiegeUniteLegale <chr>, economieSocialeSolidaire <...
12  #   societeMissionUniteLegale <chr>, caractereEmployeurUniteLegale <chr>
```

4.27 Exercice 3

Exercice 3 (manipulation avec duckdb)

- Avec duckdb, se connecter au fichier du RA2020 au format Parquet (cf. exercice 1)
- Filtrer la table du RA2020 EXPLOITATIONS sur la région de votre choix (SIEGE_REG) puis calculer la moyenne de la SAU (SAU_TOT) en fonction de l'orientation nomenclature agrégée (OTEFDA_COEF17).
- Réaliser le même traitement que précédemment mais sur une région différente.
- Fusionner les 2 tables créées aux questions précédentes de manière à avoir une table qui se présente sous la forme suivante :

OTEFDA_COEF17	MOY_SAU_76	MOY_SAU_93
5074	65905.17	2586.15
3900	77613.40	53591.05
4800	573678.07	216192.42
9000	6308.55	554.31
4700	78358.02	8047.61
3500	315107.89	104019.71
6184	340337.60	54778.68
2829	17959.11	18192.31
4500	122406.67	7476.54
1516	900230.49	111232.37
4600	633306.48	51781.59

- Re-faites l'exercice avec SQL (ou la syntaxe dplyr)

4.28 Pour finir, une comparaison entre {arrow} et {duckdb} (1/2)

Tableau repris de la documentation [utilitr](#).

Je souhaite...	arrow	duckdb
Optimiser mes traitements pour des données volumineuses	✓	✓
Travailler sur un fichier .parquet ou .csv sans le charger entièrement en mémoire	✓	✓
Utiliser la syntaxe <code>dplyr</code> pour traiter mes données	✓	✓
Utiliser du langage SQL pour traiter mes données	✗	✓
Joindre des tables très volumineuses (plus de 4 Go)	✗	✓
Utiliser des fonctions fenêtres	✗	✓
Utiliser des fonctions statistiques qui n'existent pas dans arrow	✗	✓
Écrire un fichier .parquet	✓	✓

4.29 Pour finir, une comparaison entre {arrow} et {duckdb} (2/2)

{arrow} et {duckdb} partagent de nombreux concepts. Voici quelques différences :

- **{duckdb} comprend parfaitement SQL.** Si vous êtes à l'aise avec ce langage, vous ne serez pas dépayrés.
- Le projet duckdb est très récent. Il y a régulièrement des évolutions qui sont souvent des extensions ou des optimisations, et parfois la résolution de bugs. **arrow est un projet plus ancien et plus mature.**
- **La couverture fonctionnelle des fonctions standards de R est meilleure sur {duckdb} que sur {arrow}.**
Il est préférable d'utiliser {duckdb} pour les jointures de tables volumineuses.

De même, les fonctions `pivot_wider()`, `pivot_longer()` et les `windows_function` existent nativement dans duckdb mais pas dans arrow. Par exemple :

```
1 # arrow ne peut pas exécuter ceci
2 bpe_ens_2022_dataset |>
3   group_by(DEP) |>
4   mutate(NB_EQUIP_TOTAL_DEP = sum(NB_EQUIP)) |>
5   select(DEP, NB_EQUIP, NB_EQUIP_TOTAL_DEP)
```

```
1 # Source:   SQL [?? x 3]
2 # Database: DuckDB v0.9.2 [unknown@Linux 6.5.0-1024-azure:R 4.3.2/:n
3 # Groups:   DEP
4   DEP      NB_EQUIP  NB_EQUIP_TOTAL_DEP
5   <chr>    <dbl>          <dbl>
6   1 09        2            7316
7   2 09        3            7316
8   # i more rows
```

5 Pour en savoir plus



5.1 Conseil de lecture

Pour ceux qui veulent aller plus loin :

- Comment bien utiliser le package [arrow](#) ? C'est par [ici](#)
- Comment utiliser DuckDB sur des fichiers au format Parquet ? C'est par [là](#)
- Comment requêter des fichiers au format Parquet avec Python ? Lien vers la [documentation](#).
- Why Parquet Is the Go-To Format for Data Engineer. Article en anglais très pointu sur la constitution et le fonctionnement des fichiers au format Parquet. Un must-read à lire par [ici](#).

Note interne écrite par le DEMESIS : voir [ici](#)