sparklyr

RAPPEL

Comme vu dans notre document sur Haddop et Sparks, Apache Spark est un framework open source de calcul distribué dédié au Big Data. Sa particularité est qu'il est capable de travailler en mémoire vive. Il est très performant pour les opérations nécessitant plusieurs itérations sur les mêmes données, exactement ce dont ont besoin les algorithmes de machine learning.

Spark peut fonctionner sans Hadoop, mais il a besoin d'un gestionnaire de clusters (qu'il a en interne) et d'un système de fichiers distribués (qu'il n'a pas), ce que peut lui fournir Hadoop avec respectivement Hadoop Yarn et HDFS (Hadoop Distributed File System). De fait, les faire fonctionner ensemble est très avantageux :Hadoop pour le stockage et Spark pour les calculs.

Au-delà des API (modules de classes et fonctions) standards, Spark intègre des librairies additionnelles : Streaming, traitement des données en flux ; SQL, accès aux données Spark avec des requêtes SQL ; GraphX, traitement des graphes ; MLlib, types de données et algorithmes pour le machine learning.

I- INTRODUCTION AU PACKAGE SPARLYR

- a. Présentation

Sparklyr est un package développé par les équipes de R studio. Il a été sparklyr, un package développé par les équipes de RStudio. Il a été pensé pour fonctionner parfaitement avec dplyr (une extension facilitant le traitement et la manipulation de données contenues dans une ou plusieurs tables), et directement dans RStudio. c'est est un package R qui vous permet d'accéder à Spark à partir de R. Cela signifie que vous pourrez bénéficier de la puissance de la syntaxe facile à écrire de R et de la puissance de la gestion illimitée des données de Spark. La cerise sur le gâteau est que sparklyr utilise la syntaxe de dplyr, donc une fois que vous connaissez dplyr, vous êtes à mi-chemin de la connaissance de sparklyr.

Le seul problème potentiel est que Spark est nouveau et sparklyr est encore plus récent. Cela signifie que certaines fonctionnalités sont manquantes ou difficiles à utiliser et que de nombreux messages d'erreur ne sont pas aussi clairs qu'ils devraient l'être.

- b. Installation

L'installation de sparklyr est possible depuis CRAN (Comprehensive R Archive Network). Elle de fait de la fait de la façon suivante:

```
#install.packages("sparklyr")
```

Une fois sparklyr installé, vous devrez également installer une version locale de Spark à des fins de développement.

```
#install.packages("devtools")
#library(devtools)
```

```
#library(tidyr)
#library(sparklyr)
#spark_install(version = "2.1.0")
```

II- APPLICATIONS ELEMENTAIRE DE SPARKLYR: Utilisation de spark avec les syntaxes de dplyr

Travailler avec sparklyr est très similaire à travailler avec dplyr lorsque vous avez des données dans une base de données. Sparklyr convertit votre code R en code SQL avant de le transmettre à Spark. De façon simple, Les étapes de résument comme suit:

-Connectez-vous à Spark en utilisant spark_connect (). -Travaillez. -Fermez la connexion à Spark en utilisant spark_disconnect ()

- a. Connexion à Spark

La connexion à spark se fait en utilisant la commande spark_connect. "Spark_connect" prend une URL qui donne l'emplacement à Spark. Pour un cluster local (lors de l'exécution), l'URL doit prendre la valeur "local". Pour un cluster distant (sur une autre machine, généralement un serveur hautes performances), la chaîne de connexion sera une URL et un port sur lesquels se connecter. N'oubliez pas cependant de charger le package sparklyr

```
#library(sparklyr)

#spark_conn<- spark_connect(master="local")
```

On peut ensuite par exemple regarder la version de spark en utilisant "spark_version" qui prendra en paramètre le spark connect

```
#spark_version(sc=spark_conn)
```

Pour se déconnecter il suffit d'utiliser l'instruction spark_disconnect.

```
#spark_disconnect(sc=spark_conn)
```

Vous savez maintenant comment vous connectez, et vous déconnecter de spark à partir de sparklyr.

- b. Recopier des données dans Spark

Avant de pouvoir effectuer un travail réel avec Spark, il faudrait pouvoir y insérer des données. Sparklyr a des fonctions telles que "spark_read_csv ()" qui permettent de lire un fichier CSV dans Spark. Plus généralement, il est utile de pouvoir copier des données de R vers Spark. Ceci est fait avec la fonction "copy_to ()" de dplyr.

Néanmoins je précise que, la copie de données est un processus fondamentalement lent. En fait, une grande partie de la stratégie d'optimisation des performances lors de l'utilisation de grands ensembles de données consiste à trouver des moyens d'éviter de copier les données d'un emplacement à un autre.

"copy_to ()" prend deux arguments: une connexion Spark (dest) et une trame de données (df) à copier vers Spark.

Une fois que vous avez copié vos données dans Spark pour vous assurer que votre instruction à vraiment fonctionné, vous pouvez voir une liste de toutes les trames de données stockées dans Spark en utilisant src_tbls (), qui prend simplement un argument de connexion Spark (x)

Le long de ce tutoriel j'explorerais les métadonnées de "Million Song Dataset" . Bien que Spark évolue bien au-delà d'un million de lignes de données, pour que les choses restent simples et réactives, j'utiliserais un sous-ensemble de mille pistes.

Pour clarifier la terminologie: une piste fait référence à une ligne dans le dataset. Pour notre dataset de mille pistes, c'est la même chose qu'une chanson. Pour celà, commencons dans un premier temps à chargerla bibliothèque dplyr et a nous connecter a spark.

```
#library(dplyr)
```

Nous pouvons ensuite explorer la structure de nos données en utilisant "str" Mais avant ça nous allons importer les données dont nous avons besoin.

```
#track_metadata <- readRDS("track_metadata.rds")
```

```
#str(track_metadata)
```

ensuite il faut se connecter au cluster spark

```
#spark_conn <- spark_connect(master = "local")
```

puis pour copier les données dans spark on utilise copy_to comme suit

```
#track_metadata_tbl <- copy_to(spark_conn, track_metadata)
```

Vous pouvez maintenant vérifier la liste des dataset disponible en utilisant src_tbls

```
#src_tbls(spark_conn)
```

Il vous renvoie bien Track_metadata

- c. Tibble

Après avoir copié lorsque vous avez copié les données dans Spark, copy_to () a renvoyé une valeur. Cette valeur de retour est un type spécial de tibble () qui ne contient aucune donnée qui lui est propre. Pour expliquer cela, vous devez en savoir un peu plus sur la manière dont les packages tidyverse stockent les données. Les tibbles ne sont généralement qu'une variante des dataframes qui ont une meilleure méthode d'impression. Cependant, dplyr leur permet également de stocker des données à partir d'une source de données distante, comme des bases de données, et comme c'est le cas ici, Spark.

Du côté Spark, les données sont stockées dans un Dataframe. C'est un équivalent plus ou moins direct du type de variable data.frame de R. L'appel de tbl () avec une connexion Spark et une chaîne nommant les données Spark, renverra le même objet tibble qui a été renvoyé lorsque vous avez utilisé copy_to (). Une fonction utile ici est la fonction object_size() du paquet pryr Un outil utile que vous verrez dans cet exercice est la fonction object_size () du paquet pryr. Cela vous montre combien de mémoire un objet prend.

Premièrement nous allons nous connecter aux données dans Spark

```
#track_metadata_tbl <- tbl(spark_conn, "track_metadata")
```

Puis nous regarderons la dimension de nos données.

```
#dim(track_metadata_tbl)
```

- d. sélection de colonnes

Le moyen le plus simple de manipuler les données stockées dans Spark est d'utiliser la syntaxe dplyr.

dplyr permet 5 actions principales utilisable sur les dataframe: Sélectionner des colonnes, filtrer les lignes, organiser l'ordre des lignes, modifier les colonnes ou ajouter de nouvelles colonnes et calculer des statistiques récapitulatives.

Commençons par sélectionner les colonnes. Cela se fait par la fonction select () avec un tibble, suivi des noms sans guillemets des colonnes que vous souhaitez sélectionner. Pour sélectionner les colonnes vous devez procedez comme suit.

```
#track_metadata_tbl %>%
  # Selection des colonnes
#  select(artist_name, release, title, year)
```

Il faut noter Notez que l'indexation entre crochets n'est actuellement pas prise en charge dans sparklyr. Donc vous ne pouvez pas l'utiliser. Pour preuve: vous pourrez essayer le code suivant

```
#tryCatch({
#    track_metadata_tbl[, c("artist_name", "release", "title","year")]
#  },
 # error = print
#)
```

- .e Filtrer des lignes

Outre la sélection de colonnes, l'autre moyen d'extraire des parties importantes de votre ensemble de données consiste à filtrer les lignes. Ceci peut etre réalisé en utilisant la fonction " filter ()". Pour utiliser"filter ()", il faut lui passez un tibble et quelques conditions logiques Nous allons afficher les colonnes précedement selectionnées en filtrant que l'année.

```
#glimpse(track_metadata_tbl)

#track_metadata_tbl %>%
  # Selection des colonnes
 # select(artist_name, release, title, year) %>%
  # Filtrage
#  filter(year >= 1960, year < 1970)
```

- .f ordonner le dataset

La fonction arrange () vous permet de réorganiser les lignes d'un tibble. Il faut un tibble, suivi des noms de colonnes sans guillemets. Par exemple, pour trier nom_artiste, puis organiser dans l'ordre décroissant par année puis par titre, vous écrirez ce qui suit.

```
#track_metadata_tbl %>%
#  arrange(artist_name, desc(year),title)
```

- g. mutation de colonnes

Tous les ensembles de données ne sont pas parfaitement propre. Souvent, vous devez corriger des valeurs ou créer de nouvelles colonnes dérivées de vos données existantes. Le processus de changement ou d'ajout de colonnes est appelé mutation dans dplyr, et est effectué en utilisant " mutate ()". Cette fonction prend un tibble et des arguments nommés pour mettre à jour les colonnes. Le nom de chacun de ces arguments est le nom des colonnes à modifier ou à ajouter, et la valeur est une expression expliquant comment la mettre à jour. Par exemple nous allons utiliser cette fonction pour créer une nouvelle colonnes contenant la durée de chaque chansons en min, sachant qu'elles sont données en secondes.

```
#track_metadata_tbl %>%
  # Selection des colonnes
 # select(title,duration)%>%
  # utilisation de la mutation
  #mutate(duration_minutes=duration / 60)
```

.h obtenir les statistiques descripives

Si vous souhaitez avoir les statistiques descriptives telles que la moyenne, le maximum ou l'écart type il vous faudra utiliser la fonction "summary ()" comme suit:

```
#track_metadata_tbl


#track_metadata_tbl %>%

  # Selection des colonnes

  #select(title, duration)%>%

  # Mutation précédente

  #mutate(duration_minutes= duration / 60)%>%

  # Summarize column
  #summarise(mean_duration_minute= mean(duration_minutes))
```

Il faut Noter que dplyr fonctionne de sorte de toujours conserver les données dans des tibbles. La valeur de retour ici est donc un tibble avec une ligne et une colonne pour chaque élément calculé.

III- UTILISATION AVANCEE DE DPLYR

Je tiens a rapeller que c'est parce que Sparklyr fonctionne avec dplyr que la majorité des syntaxes utilisées font référence à dplyr sous une connnexion soark. Arès avoir pris connaissance des méthodes de bases, il est temps de parler de quelques fonctions qui vous permetrons d'aller plus loin.

- a. la selection avancée

Si votre dataset contient des milliers de colonnes et que vous souhaitez en sélectionner un grand nombre, il peut être très fastidieux de saisir le nom de chaque C'est là qu'entrent en jeu certaines méthodes d'assistance pour faciliter la sélection de plusieurs colonnes. Ces méthodes incluent "starts_with ()" et "ends_with ()", qui aux colonnes qui commencent ou se terminent par un certain préfixe ou suffixe respectivement.
Ces fonctions ne peuvent etre appelées qu'avec select comme suit:

```
#track_metadata_tbl

#track_metadata_tbl %>%
  # Selection des colonnes qui commencent par artist
 # select(
  #  starts_with("artist")
  #)

  # selection des colonnes se terminant par id
#track_metadata_tbl %>%
 # select(
  #  ends_with("id")
  #)
```

On peut également utiliser la méthode “contain” afin de sélectionner toutes les colonnes contenant une expression précise. Afin de faire une sélection encore plus affinée, ma méthode “macthes()”, peut etre utilisée avec les regex.

```
#track_metadata_tbl

#track_metadata_tbl %>%
  # Selection des colonnes contenant "ti"
 # select(contains("ti"))

#track_metadata_tbl %>%
  # Selection des colonnes macthant avec le regex"ti.?t"
 # select(matches("ti.?t"))
```

On peut également chercher à savoir quelles sont les catégories d’un dataset en utilisant la méthode “distinct()” comme suit:

```
#track_metadata_tbl

#track_metadata_tbl %>%
  # renvoie les lignes avec des noms d'artistes distincs
   #distinct(artist_name)
```

Pour compter le nombre d’éléments distincts dans une colonne vous pouver utiliser la commande suivante:

```
#track_metadata_tbl %>%
  # pour compter le nombre de chaque élément distinct dans la colonne "artist_name", o, rajoutera le pa
 # count(artist_name, sort= TRUE)%>%
# top_n permet d'avoir les 20 premiers éléments. c'est un peu l'équivalent de 'head()' dans R.
  #top_n(20)
```

- b. collecter des données depuis spark

Nous avons précedemment vu que les tibbles ne stockent pas une copie des données. Au lieu de cela, les données restent dans Spark et le tibble stocke simplement les détails de ce qu’il souhaite récupérer à partir de Spark.

Pour une raison une une autre vous voudrez être déplacer vos données de Spark vers R.

Pour collecter nos données: c'est-à-dire pour les déplacer de Spark vers R, il faudra apeller la méthode "collect ()". Après celà on pourra vérifier la classe de l'élément collecté. Celà ce fait comme suit

```
#track_metadata_tbl

#results <- track_metadata_tbl %>%

 # filter(artist_familiarity > 0.9)

# Ici on peut examiner la classe de l'objet result
#class(results)

# ensuite on collecte nos données et on vérifie la classe de l'objet collecté
#collected <- results %>%
 # collect(collected)
#class(collected)
```