



Formation PySpark (PySpark Datasources : Lire un fichier CSV dans un DataFrame)

-  Mohammed ATTIK (Copyright)
-  mohammed.attik@gmail.com

[link](#)

PySpark : Lire un fichier CSV dans un DataFrame

PySpark propose la méthode `csv("chemin")` sur `DataFrameReader` pour lire un fichier CSV dans un DataFrame PySpark, et `dataframeObj.write.csv("chemin")` pour enregistrer ou écrire dans le fichier CSV.

Dans cette section, vous apprendrez comment lire un seul fichier, plusieurs fichiers, tous les fichiers d'un répertoire local dans un DataFrame, appliquer des transformations, et enfin écrire le DataFrame de nouveau dans un fichier CSV à l'aide d'un exemple PySpark.

PySpark prend en charge la lecture d'un fichier CSV avec un séparateur de pipe, de virgule, de tabulation, d'espace ou tout autre séparateur/délimiteur de fichiers.

Remarque : PySpark prend en charge nativement la lecture de fichiers aux formats CSV, JSON et de nombreux autres formats de fichiers dans un DataFrame PySpark.

Table des matières :

- Lire un fichier CSV dans un DataFrame PySpark
- Lire plusieurs fichiers CSV
- Lire tous les fichiers CSV dans un répertoire
- Options lors de la lecture d'un fichier CSV
- séparateur
- InferSchema (inférer le schéma)
- en-tête
- guillemets
- valeurs nulles
- format de date
- Lire des fichiers CSV avec un schéma spécifié par l'utilisateur
- Application de transformations sur le DataFrame
- Écrire un DataFrame dans un fichier CSV
- Utilisation d'options
- Mode de sauvegarde

Lire un fichier CSV dans un DataFrame PySpark

À l'aide de `csv("chemin")` ou `format("csv").load("chemin")` de `DataFrameReader`, vous pouvez lire un fichier CSV dans un `DataFrame` PySpark. Ces méthodes prennent comme argument un chemin de fichier à lire. Lorsque vous utilisez la méthode `format("csv")`, vous pouvez également spécifier les sources de données par leur nom entièrement qualifié, mais pour les sources intégrées, vous pouvez simplement utiliser leurs noms abrégés (csv, json, parquet, etc.).

```
!wget https://raw.githubusercontent.com/spark-examples/pyspark-examples/master/resources/zipco
```

```
# Importer les bibliothèques nécessaires
from pyspark.sql import SparkSession

# Créer une session Spark
spark = SparkSession.builder().master("local[1]") \
    .appName("SparkByExamples.com") \
    .getOrCreate()

# Lire le fichier CSV dans un DataFrame
df = spark.read.csv("/tmp/resources/zipcodes.csv")

# Afficher le schéma du DataFrame
df.printSchema()

# Utilisation du nom de la source de données entièrement qualifié, vous pouvez également faire

# Lire le fichier CSV en spécifiant le format
df = spark.read.format("csv") \
    .load("/tmp/resources/zipcodes.csv")

# ou
df = spark.read.format("org.apache.spark.sql.csv") \
    .load("/tmp/resources/zipcodes.csv")

# Afficher le schéma du DataFrame
df.printSchema()

# Cet exemple lit les données dans les colonnes du DataFrame avec "_c0" pour la première colon
# Et par défaut, le type de données de toutes ces colonnes est traité comme une chaîne (String
```

1.1 Utilisation de l'en-tête du fichier pour les noms de colonnes

Si vous avez un en-tête avec les noms de colonnes dans votre fichier d'entrée, vous devez spécifier explicitement `"True"` pour l'option `"header"` en utilisant l'option `(("header", True))`. Si vous ne le mentionnez pas, l'API considère l'en-tête comme un enregistrement de données.

```
df2 = spark.read.option("header", True) \
    .csv("/tmp/resources/zipcodes.csv")
```

Comme mentionné précédemment, PySpark lit toutes les colonnes comme des chaînes (StringType) par défaut. Je vais expliquer dans les sections suivantes comment lire le schéma (inferSchema) à partir de l'en-tête et dériver le type de colonne en fonction des données.

1.2 Lecture de plusieurs fichiers CSV

En utilisant la méthode `read.csv()`, vous pouvez également lire plusieurs fichiers CSV. Il suffit de passer tous les noms de fichiers en les séparant par des virgules en tant que chemin. Par exemple :

```
df = spark.read.csv("chemin1,chemin2,chemin3")
```

1.3 Lire tous les fichiers CSV dans un répertoire

Nous pouvons lire tous les fichiers CSV d'un répertoire dans un DataFrame en passant simplement le répertoire en tant que chemin à la méthode `csv()`.

```
df = spark.read.csv("Chemin du répertoire")
```

2. Options lors de la lecture d'un fichier CSV

Le jeu de données CSV de PySpark propose plusieurs options pour travailler avec les fichiers CSV. Voici quelques-unes des options les plus importantes expliquées avec des exemples.

Vous pouvez utiliser soit la méthode d'enchaînement `option(self, key, value)` pour utiliser plusieurs options, soit la méthode d'options alternatives `option(self, **options)`.

2.1 Délimiteur

L'option de délimiteur est utilisée pour spécifier le délimiteur de colonne du fichier CSV. Par défaut, il s'agit de la virgule (,), mais il peut être défini sur n'importe quel caractère tel que le pipe (|), la tabulation (\t), l'espace en utilisant cette option.

```
df3 = spark.read.options(delimiter=',') \
    .csv("Chemin du fichier/zipcodes.csv")
```

2.2 inferSchema

La valeur par défaut de cette option est `False`. Lorsqu'elle est définie sur `True`, elle infère automatiquement les types de colonnes en fonction des données. Notez que cela nécessite de lire les données une fois de plus pour inférer le schéma.

```
df4 = spark.read.options(inferSchema='True', delimiter=',') \
    .csv("Chemin du fichier/zipcodes.csv")
```

Alternativement, vous pouvez également écrire ceci en enchaînant la méthode `option()`.

```
df4 = spark.read.option("inferSchema", True) \
    .option("delimiter", ",") \
    .csv("Chemin du fichier/zipcodes.csv")
```

2.3 En-tête

Cette option est utilisée pour lire la première ligne du fichier CSV en tant que noms de colonnes. Par défaut, la valeur de cette option est `False`, et tous les types de colonnes sont supposés être des chaînes.

```
df3 = spark.read.options(header='True', inferSchema='True', delimiter=',') \
    .csv("/tmp/resources/zipcodes.csv")
```

2.4 Guillemets

Lorsque vous avez une colonne avec un délimiteur utilisé pour séparer les colonnes, utilisez l'option `quotes` pour spécifier le caractère de guillemet, par défaut c'est `"` et les délimiteurs à l'intérieur des guillemets sont ignorés. Mais en utilisant cette option, vous pouvez définir n'importe quel caractère.

```
df_with_quotes = spark.read.options(quote='\"', header='True', inferSchema='True', delimiter=',') \
    .csv("/chemin/vers/votre/fichier.csv")
```

2.5 Valeurs nulles

En utilisant l'option `nullValues`, vous pouvez spécifier la chaîne dans un fichier CSV à considérer comme nulle. Par exemple, si vous souhaitez considérer une colonne de date avec une valeur `"1900-01-01"` comme nulle dans le `DataFrame`.

```
df_with_nulls = spark.read.options(nullValue='1900-01-01', header='True', inferSchema='True', \
    .csv("/chemin/vers/votre/fichier.csv"))
```

2.6 Format de date

L'option `dateFormat` est utilisée pour définir le format des colonnes `DateType` et `TimestampType` en entrée. Prend en charge tous les formats `java.text.SimpleDateFormat`.

```
df_with_date_format = spark.read.options(dateFormat='yyyy-MM-dd', header='True', inferSchema='True') \
    .csv("/chemin/vers/votre/fichier.csv")
```

3. Lecture de fichiers CSV avec un schéma personnalisé spécifié par l'utilisateur

Si vous connaissez le schéma du fichier à l'avance et ne souhaitez pas utiliser l'option `inferSchema` pour les noms et types de colonnes, utilisez des noms et types de colonnes personnalisés définis par l'utilisateur en utilisant l'option `schema`.

```
schema = StructType() \
    .add("RecordNumber", IntegerType(), True) \
    .add("Zipcode", IntegerType(), True) \
    .add("ZipCodeType", StringType(), True) \
    .add("City", StringType(), True) \
    .add("State", StringType(), True) \
    .add("LocationType", StringType(), True) \
    .add("Lat", DoubleType(), True) \
    .add("Long", DoubleType(), True) \
    .add("Xaxis", IntegerType(), True) \
    .add("Yaxis", DoubleType(), True) \
    .add("Zaxis", DoubleType(), True) \
    .add("WorldRegion", StringType(), True) \
    .add("Country", StringType(), True) \
    .add("LocationText", StringType(), True) \
    .add("Location", StringType(), True) \
    .add("Decommisioned", BooleanType(), True) \
    .add("TaxReturnsFiled", StringType(), True) \
    .add("EstimatedPopulation", IntegerType(), True) \
    .add("TotalWages", IntegerType(), True) \
    .add("Notes", StringType(), True)

df_with_schema = spark.read.format("csv") \
    .option("header", True) \
    .schema(schema) \
    .load("/tmp/resources/zipcodes.csv")
```

4. Application de transformations sur le DataFrame

Une fois que vous avez créé un DataFrame à partir du fichier CSV, vous pouvez appliquer toutes les transformations et actions prises en charge par le DataFrame. Veuillez vous référer au lien pour plus de détails.

5. Écrire le DataFrame PySpark dans un fichier CSV

Utilisez la méthode `write()` de l'objet PySpark `DataFrameWriter` pour écrire un DataFrame PySpark dans un fichier CSV.

```
df.write.option("header", True) \
    .csv("/tmp/spark_output/zipcodes")
```

5.1 Options

Lors de l'écriture d'un fichier CSV, vous pouvez utiliser plusieurs options. Par exemple, "header" pour afficher les noms de colonnes du DataFrame en tant qu'enregistrement d'en-tête et "delimiter" pour spécifier le délimiteur dans le fichier de sortie CSV.

```
df2.write.options(header='True', delimiter=',') \
    .csv("/tmp/spark_output/zipcodes")
```

D'autres options disponibles sont "quote", "escape", "nullValue", "dateFormat", "quoteMode".

5.2 Modes de sauvegarde

Le DataFrameWriter de PySpark dispose également d'une méthode mode() pour spécifier le mode de sauvegarde.

- "overwrite" : le mode est utilisé pour écraser le fichier existant.
- "append" : pour ajouter les données au fichier existant.
- "ignore" : ignore l'opération d'écriture si le fichier existe déjà.
- "error" : c'est une option par défaut lorsqu'un fichier existe déjà, cela retourne une erreur.

```
df2.write.mode('overwrite').csv("/tmp/spark_output/zipcodes")
# Vous pouvez également utiliser ceci
df2.write.format("csv").mode('overwrite').save("/tmp/spark_output/zipcodes")
```

6. Exemple complet de lecture de CSV avec PySpark

```
##### Begin head Google-colab
#
!wget -q http://archive.apache.org/dist/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz
!tar xf spark-3.5.0-bin-hadoop3.tgz
#
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!pip install -q findspark
#
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.0-bin-hadoop3"
#
import findspark
findspark.init()
#### end head Google-colab

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
from pyspark.sql.types import ArrayType, DoubleType, BooleanType
from pyspark.sql.functions import col, array_contains
```

```

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

!wget https://raw.githubusercontent.com/spark-examples/pyspark-examples/master/resources/zipco

df = spark.read.csv("/content/zipcodes.csv")

df.printSchema()

df2 = spark.read.option("header", True) \
    .csv("/content/zipcodes.csv")
df2.printSchema()

df3 = spark.read.options(header='True', delimiter=',') \
    .csv("/content/zipcodes.csv")
df3.printSchema()

schema = StructType() \
    .add("RecordNumber", IntegerType(), True) \
    .add("Zipcode", IntegerType(), True) \
    .add("ZipCodeType", StringType(), True) \
    .add("City", StringType(), True) \
    .add("State", StringType(), True) \
    .add("LocationType", StringType(), True) \
    .add("Lat", DoubleType(), True) \
    .add("Long", DoubleType(), True) \
    .add("Xaxis", IntegerType(), True) \
    .add("Yaxis", DoubleType(), True) \
    .add("Zaxis", DoubleType(), True) \
    .add("WorldRegion", StringType(), True) \
    .add("Country", StringType(), True) \
    .add("LocationText", StringType(), True) \
    .add("Location", StringType(), True) \
    .add("Decommisioned", BooleanType(), True) \
    .add("TaxReturnsFiled", StringType(), True) \
    .add("EstimatedPopulation", IntegerType(), True) \
    .add("TotalWages", IntegerType(), True) \
    .add("Notes", StringType(), True)

df_with_schema = spark.read.format("csv") \
    .option("header", True) \
    .schema(schema) \
    .load("/content/zipcodes.csv")
df_with_schema.printSchema()

df2.write.mode("overwrite").option("header", True) \
    .csv("/content/zipcodes123.csv")

df2.createOrReplaceTempView("ParquetTable")
parkSQL = spark.sql("select * from ParquetTable")
parkSQL.show()



```

7. Conclusion :

Dans ce tutoriel, vous avez appris comment lire un fichier CSV, plusieurs fichiers CSV et tous les fichiers d'un dossier local dans un DataFrame PySpark, en utilisant plusieurs options pour modifier le comportement par défaut, et comment écrire des fichiers CSV dans un DataFrame en utilisant différentes options d'enregistrement.

Remarque : En plus des options mentionnées ci-dessus, l'API CSV de PySpark prend en charge de nombreuses autres options.

Formation PySpark (PySpark Datasources : Lecture et écriture de fichiers Parquet avec PySpark)

-  Mohammed ATTIK (Copyright)
-  mohammed.attik@gmail.com

[link](#)

Lecture et écriture de fichiers Parquet avec PySpark

PySpark SQL propose des méthodes pour lire un fichier Parquet dans un DataFrame et écrire un DataFrame dans des fichiers Parquet. Les fonctions `parquet()` de `DataFrameReader` et `DataFrameWriter` sont utilisées respectivement pour la lecture et l'écriture/création d'un fichier Parquet. Les fichiers Parquet conservent le schéma ainsi que les données, c'est pourquoi ils sont utilisés pour traiter des fichiers structurés.

Dans cette section, je vais expliquer comment lire et écrire un fichier Parquet, ainsi que comment partitionner les données et récupérer les données partitionnées à l'aide de SQL.

Voici quelques déclarations simples sur la manière d'écrire et de lire des fichiers Parquet en PySpark, que je vais expliquer en détail dans les sections suivantes.

```
# Lire et écrire un fichier Parquet en utilisant parquet()  
df.write.parquet("/tmp/out/people.parquet")  
parDF1 = spark.read.parquet("/tmp/out/people.parquet")
```

Avant d'expliquer en détail, comprenons ce qu'est un fichier Parquet et ses avantages par rapport aux formats de fichiers CSV, JSON et autres fichiers texte.

Qu'est-ce qu'un fichier Parquet ?

Le fichier Apache Parquet est un format de stockage orienté colonne disponible pour tout projet de l'écosystème Hadoop, indépendamment du choix du framework de traitement des données, du modèle de données ou du langage de programmation.

Avantages :

- Lors de la requête d'un stockage orienté colonne, il saute très rapidement les données non pertinentes, ce qui accélère l'exécution des requêtes. Ainsi, les requêtes d'agrégation prennent moins de temps par rapport aux bases de données orientées lignes.
- Il est capable de prendre en charge des structures de données imbriquées avancées.
- Parquet prend en charge des options de compression efficaces et des schémas d'encodage.

PySpark SQL prend en charge à la fois la lecture et l'écriture de fichiers Parquet qui captent automatiquement le schéma des données d'origine. Cela réduit également le stockage des données en moyenne de 75%. PySpark prend en charge Parquet par défaut dans sa bibliothèque, donc aucune bibliothèque externe n'est nécessaire.

Exemple PySpark Apache Parquet

Afin de comprendre les concepts du fichier Parquet, travaillons sur l'écriture d'un fichier Parquet à partir d'un DataFrame. Tout d'abord, créez un DataFrame PySpark à partir d'une liste de données en utilisant la méthode `spark.createDataFrame()`.

```
# Créer des données d'exemple
# Imports
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("parquetFile").getOrCreate()
data = [("James", "", "Smith", "36636", "M", 3000),
        ("Michael", "Rose", "", "40288", "M", 4000),
        ("Robert", "", "Williams", "42114", "M", 4000),
        ("Maria", "Anne", "Jones", "39192", "F", 4000),
        ("Jen", "Mary", "Brown", "", "F", -1)]
columns = ["firstname", "middlename", "lastname", "dob", "gender", "salary"]
df = spark.createDataFrame(data, columns)
```

Dans cet exemple, un DataFrame est créé avec les colonnes `firstname`, `middlename`, `lastname`, `dob`, `gender`, `salary`.

Écriture d'un DataFrame PySpark dans un fichier au format Parquet

Créons maintenant un fichier Parquet à partir du DataFrame PySpark en appelant la fonction `parquet()` de la classe `DataFrameWriter`. Lorsque vous écrivez un DataFrame dans un fichier Parquet, il préserve automatiquement les noms de colonnes et leurs types de données. Chaque fichier partiel créé par PySpark a l'extension de fichier `.parquet`. Voici l'exemple :

```
# Écrire un DataFrame dans un fichier Parquet en utilisant write.parquet()
df.write.parquet("/tmp/output/people.parquet")
```

Lecture d'un fichier Parquet dans un DataFrame PySpark

PySpark fournit une méthode `parquet()` dans la classe `DataFrameReader` pour lire le fichier Parquet dans un dataframe. Voici un exemple de lecture d'un fichier Parquet vers un DataFrame :

```
# Lire un fichier Parquet en utilisant read.parquet()
parDF = spark.read.parquet("/tmp/output/people.parquet")
```

Types de modes d'enregistrement des fichiers Parquet

En PySpark, après avoir écrit le DataFrame dans le fichier Parquet, nous pouvons spécifier le mode d'enregistrement à l'aide de la méthode `.mode()`. Voici les types de modes d'enregistrement disponibles dans PySpark à partir du module `pyspark.sql.DataFrameWriter.mode`.

Syntaxe :

`DataFrameWriter.mode(saveMode: Optional[str])`

Types de modes d'enregistrement des fichiers Parquet : Options

- **append** : Ce mode ajoute les données du DataFrame au fichier existant, si les fichiers de destination existent déjà. Dans le cas où les fichiers de destination n'existent pas, il créera un nouveau fichier Parquet à l'emplacement spécifié.

Exemple :

```
df.write.mode("append").parquet("chemin/vers/le/fichier/parquet")
```

- **overwrite** : Ce mode écrase le fichier Parquet de destination avec les données du DataFrame. S'il n'existe pas, il crée un nouveau fichier Parquet.

Exemple :

```
df.write.mode("overwrite").parquet("chemin/vers/le/fichier/parquet")
```

- **ignore** : Si le fichier Parquet de destination existe déjà, ce mode ne fait rien et n'écrit pas le DataFrame dans le fichier. S'il n'existe pas, il crée un nouveau fichier Parquet.

Exemple :

```
df.write.mode("ignore").parquet("chemin/vers/le/fichier/parquet")
```

- **error ou errorIfExists** : Ce mode génère une erreur si le fichier Parquet de destination existe déjà. Il n'écrit pas le DataFrame dans le fichier. S'il n'existe pas, il crée un nouveau fichier Parquet.

Exemple :

```
df.write.mode("error").parquet("chemin/vers/le/fichier/parquet")
```

Ajouter ou écraser un fichier Parquet existant

En utilisant le mode d'enregistrement append, vous pouvez ajouter un DataFrame à un fichier Parquet existant. Pour écraser, utilisez le mode d'enregistrement overwrite.

```
# Utiliser les modes append et overwrite pour enregistrer un fichier Parquet
df.write.mode('append').parquet("chemin/vers/le/fichier/parquet")
df.write.mode('overwrite').parquet("chemin/vers/le/fichier/parquet")
```

Exécution de requêtes SQL sur un DataFrame

PySpark SQL permet de créer des vues temporaires sur des fichiers Parquet pour exécuter des requêtes SQL. Ces vues sont disponibles jusqu'à la fin de votre programme.

```
# Utilisation de spark.sql
parDF.createOrReplaceTempView("ParquetTable")
parkSQL = spark.sql("select * from ParquetTable where salary >= 4000 ")
```

Création d'une table sur un fichier Parquet

Examinons maintenant l'exécution de requêtes SQL sur un fichier Parquet. Pour exécuter des requêtes SQL, créez une vue temporaire ou une table directement sur le fichier Parquet au lieu de la créer à partir du DataFrame.

```
# Créer une vue temporaire sur le fichier Parquet
spark.sql("CREATE TEMPORARY VIEW PERSON USING parquet OPTIONS (path \"/tmp/output/people.parqu
spark.sql("SELECT * FROM PERSON").show()
```



Ici, nous avons créé une vue temporaire PERSON à partir du fichier "people.parquet". Cela donne les résultats suivants.

```
# Output
+-----+-----+-----+-----+-----+-----+
|firstname|middlename|lastname|  dob|gender|salary|
+-----+-----+-----+-----+-----+-----+
|  Robert |         |Williams|42114|    M|  4000|
|   Maria |      Anne|   Jones|39192|    F|  4000|
|Michael |      Rose|         |40288|    M|  4000|
|   James |         |   Smith|36636|    M|  3000|
|     Jen |     Mary|   Brown|     |    F|    -1|
+-----+-----+-----+-----+-----+-----+
```

Créer un fichier Parquet partitionné

Lorsque nous exécutons une requête particulière sur la table PERSON, elle parcourt toutes les lignes et renvoie les résultats. Cela est similaire à l'exécution traditionnelle de requêtes de base de données. Dans PySpark, nous pouvons améliorer l'exécution des requêtes de manière optimisée en partitionnant les données à l'aide de la méthode pyspark partitionBy(). Voici un exemple de partitionBy().

```
df.write.partitionBy("gender", "salary").mode("overwrite").parquet("/tmp/output/people2.parque
```

Récupération à partir d'un fichier Parquet partitionné

L'exemple ci-dessous explique comment lire un fichier Parquet partitionné dans un DataFrame avec gender=M.

```
parDF2 = spark.read.parquet("/tmp/output/people2.parquet/gender=M")
parDF2.show(truncate=False)
```

La sortie pour l'exemple ci-dessus est la suivante.

```
# Sortie
+-----+-----+-----+-----+-----+
|firstname|middlename|lastname|dob  |salary|
+-----+-----+-----+-----+-----+
|Robert   |           |Williams|42114|4000  |
|Michael  |Rose      |         |40288|4000  |
|James    |           |Smith   |36636|3000  |
+-----+-----+-----+-----+-----+
```

Création d'une table sur un fichier Parquet partitionné

Ici, je crée une table sur un fichier Parquet partitionné et j'exécute une requête qui s'exécute plus rapidement que la table sans partition, améliorant ainsi les performances.

```
# Créer une vue temporaire sur un fichier Parquet partitionné
spark.sql("CREATE TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path \"/tmp/output/people2.par
spark.sql("SELECT * FROM PERSON2").show()
```

Voici la sortie.

```
# Sortie
+-----+-----+-----+-----+-----+
|firstname|middlename|lastname|dob  |salary|
```

Maria	Anne	Jones	39192	4000
Jen	Mary	Brown		-1

Exemple complet de lecture et écriture de fichiers Parquet avec PySpark

```
##### Begin head Google-colab
#
!wget -q http://archive.apache.org/dist/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz
!tar xf spark-3.5.0-bin-hadoop3.tgz
#
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!pip install -q findspark
#
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.0-bin-hadoop3"
#
import findspark
findspark.init()
##### end head Google-colab

# Imports
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("parquetFile").getOrCreate()
data = [("James ", "", "Smith", "36636", "M", 3000),
        ("Michael ", "Rose", "", "40288", "M", 4000),
        ("Robert ", "", "Williams", "42114", "M", 4000),
        ("Maria ", "Anne", "Jones", "39192", "F", 4000),
        ("Jen", "Mary", "Brown", "", "F", -1)]
columns = ["firstname", "middlename", "lastname", "dob", "gender", "salary"]
df = spark.createDataFrame(data, columns)
df.write.mode("overwrite").parquet("/content/people.parquet")
parDF1 = spark.read.parquet("/content/people.parquet")
parDF1.createOrReplaceTempView("parquetTable")
parDF1.printSchema()
parDF1.show(truncate=False)

parkSQL = spark.sql("select * from ParquetTable where salary >= 4000 ")
parkSQL.show(truncate=False)

spark.sql("CREATE or replace TEMPORARY VIEW PERSON USING parquet OPTIONS (path \"/content/peop
spark.sql("SELECT * FROM PERSON").show()

df.write.partitionBy("gender", "salary").mode("overwrite").parquet("/content/people2.parquet")



parDF2 = spark.read.parquet("/content/people2.parquet/gender=M")
parDF2.show(truncate=False)
```

```
spark.sql("CREATE or replace TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path \"/content/peo  
spark.sql("SELECT * FROM PERSON2").show()
```

Conclusion :

Nous avons appris comment écrire un fichier Parquet à partir d'un DataFrame PySpark, lire un fichier Parquet dans un DataFrame et créer des vues/tables pour exécuter des requêtes SQL. Nous avons également expliqué comment partitionner les fichiers Parquet pour améliorer les performances.

Formation PySpark (PySpark SQL Functions : Fonctions d'agrégation PySpark avec des exemples)

-  Mohammed ATTIK (Copyright)
-  mohammed.attik@gmail.com

[link](#)

Fonctions d'agrégation PySpark avec des exemples

PySpark fournit des fonctions d'agrégation standard intégrées définies dans l'API DataFrame. Elles sont utiles lorsque nous devons effectuer des opérations d'agrégation sur les colonnes du DataFrame. Les fonctions d'agrégation opèrent sur un groupe de lignes et calculent une seule valeur de retour pour chaque groupe.

Toutes ces fonctions d'agrégation acceptent en entrée un type de colonne (Column) ou le nom d'une colonne sous forme de chaîne, ainsi que plusieurs autres arguments en fonction de la fonction, et elles retournent un type de colonne.

Dans la mesure du possible, essayez de tirer parti de la bibliothèque standard car elles offrent une sécurité accrue lors de la compilation, gèrent les valeurs nulles et sont plus performantes par rapport aux UDF (User-Defined Functions). Si les performances sont cruciales pour votre application, évitez d'utiliser des UDF personnalisées autant que possible, car celles-ci ne garantissent pas les performances.

Fonctions d'agrégation PySpark

Les fonctions d'agrégation PySpark SQL sont regroupées sous le nom de "agg_funcs" dans PySpark. Ci-dessous se trouve une liste de fonctions définies sous ce groupe. Cliquez sur chaque lien pour en savoir plus avec des exemples.

- [approx_count_distinct](#)
- [avg](#)
- [collect_list](#)
- [collect_set](#)
- [countDistinct](#)

- `count`
- `grouping`
- `first`
- `last`
- `kurtosis`
- `max`
- `min`
- `mean`
- `skewness`
- `stddev`
- `stddev_samp`
- `stddev_pop`
- `sum`
- `sumDistinct`
- `variance, var_samp, var_pop`

Exemples de fonctions d'agrégation PySpark

Tout d'abord, créons un DataFrame pour travailler avec les fonctions d'agrégation PySpark.

```
simpleData = [("James", "Sales", 3000),
              ("Michael", "Sales", 4600),
              ("Robert", "Sales", 4100),
              ("Maria", "Finance", 3000),
              ("James", "Sales", 3000),
              ("Scott", "Finance", 3300),
              ("Jen", "Finance", 3900),
              ("Jeff", "Marketing", 3000),
              ("Kumar", "Marketing", 2000),
              ("Saif", "Sales", 4100)
            ]
schema = ["employee_name", "department", "salary"]
df = spark.createDataFrame(data=simpleData, schema=schema)
df.printSchema()
df.show(truncate=False)
```

Cela donne la sortie suivante.

```
# Output
+-----+-----+-----+
|employee_name|department|salary|
+-----+-----+-----+
|      James|      Sales|   3000|
|    Michael|      Sales|   4600|
```

	Robert	Sales	4100
	Maria	Finance	3000
	James	Sales	3000
	Scott	Finance	3300
	Jen	Finance	3900
	Jeff	Marketing	3000
	Kumar	Marketing	2000
	Saif	Sales	4100
+-----+-----+-----+			

Voyons maintenant comment agréger des données avec PySpark.

Fonction d'agrégation `approx_count_distinct`

Dans PySpark, la fonction `approx_count_distinct()` renvoie le nombre d'éléments distincts dans un groupe.

```
# approx_count_distinct()
print("approx_count_distinct: " + \
      str(df.select(approx_count_distinct("salary")).collect()[0][0]))

# Affiche approx_count_distinct: 6
```

Fonction d'agrégation `avg` (moyenne)

La fonction `avg()` retourne la moyenne des valeurs de la colonne d'entrée.

```
# avg
print("avg: " + str(df.select(avg("salary")).collect()[0][0]))

# Affiche avg: 3400.0
```

Fonction d'agrégation `collect_list`

La fonction `collect_list()` retourne toutes les valeurs d'une colonne d'entrée avec les doublons.

```
# collect_list
df.select(collect_list("salary")).show(truncate=False)

+-----+
|collect_list(salary)|
+-----+
|[3000, 4600, 4100, 3000, 3000, 3300, 3900, 3000, 2000, 4100]|
+-----+
```

Fonction d'agrégation `collect_set`

La fonction `collect_set()` retourne toutes les valeurs d'une colonne d'entrée sans doublons.

```
# collect_set
df.select(collect_set("salary")).show(truncate=False)
```

```
+-----+
|collect_set(salary)|
+-----+
|[4600, 3000, 3900, 4100, 3300, 2000]|
+-----+
```

Fonction d'agrégation `countDistinct`

La fonction `countDistinct()` retourne le nombre d'éléments distincts dans une colonne.

```
# countDistinct
df2 = df.select(countDistinct("department", "salary"))
df2.show(truncate=False)
print("Distinct Count of Department & Salary: "+str(df2.collect()[0][0]))
```

Fonction d'agrégation `count`

La fonction `count()` retourne le nombre d'éléments dans une colonne.

```
print("count: "+str(df.select(count("salary")).collect()[0]))

# Affiche count: 10
```

Fonction d'agrégation `grouping`

La fonction `grouping()` est utilisée pour indiquer si une colonne donnée en entrée est agrégée ou non dans le résultat de la requête. Elle retourne 1 si la colonne est agrégée et 0 si elle ne l'est pas.

Cependant, il est important de noter que la fonction `grouping()` ne peut être utilisée qu'avec les opérations de regroupement avancées telles que `GroupingSets`, `Cube` ou `Rollup`. Vous ne pouvez pas l'utiliser directement sur une colonne dans une requête de regroupement normale.

Voici un exemple d'utilisation de la fonction `grouping()` avec `GroupingSets` :

```
from pyspark.sql.functions import grouping

# Création de données de test
data = [("Alice", 1000), ("Bob", 2000), ("Alice", 3000), ("Bob", 4000)]
df = spark.createDataFrame(data, ["name", "salary"])
```

```
# Exemple d'utilisation de GroupingSets avec grouping()
df.groupBy("name").agg(grouping("name")).show()
```

Dans cet exemple, la fonction `grouping()` est utilisée avec `GroupingSets` pour déterminer si la colonne "name" est agrégée dans le résultat du regroupement.

Fonction d'agrégation first

La fonction `first()` retourne le premier élément d'une colonne. Lorsque `ignoreNulls` est défini sur `true`, elle retourne le premier élément non nul.

```
# first
df.select(first("salary")).show(truncate=False)

+-----+
|first(salary, false)|
+-----+
|3000                |
+-----+

# df.select(first("salary", ignoreNulls=True)).show(truncate=False)
```

Fonction d'agrégation last

La fonction `last()` retourne le dernier élément d'une colonne. Lorsque `ignoreNulls` est défini sur `true`, elle retourne le dernier élément non nul.

```
# last
df.select(last("salary")).show(truncate=False)

+-----+
|last(salary, false)|
+-----+
|4100                |
+-----+
```

Fonction d'agrégation kurtosis

La fonction `kurtosis()` retourne le kurtosis des valeurs dans un groupe.

```
df.select(kurtosis("salary")).show(truncate=False)

+-----+
|kurtosis(salary)    |
+-----+
```

```
| -0.6467803030303032 |  
+-----+
```

Le kurtosis mesure l'aplatissement ou l'étalement d'une distribution par rapport à une distribution normale.

Fonction d'agrégation max

La fonction `max()` retourne la valeur maximale dans une colonne.

```
df.select(max("salary")).show(truncate=False)
```

```
+-----+  
| max(salary) |  
+-----+  
| 4600        |  
+-----+
```

Fonction d'agrégation min

La fonction `min()` retourne la valeur minimale dans une colonne.

```
df.select(min("salary")).show(truncate=False)
```

Fonction d'agrégation mean

La fonction `mean()` retourne la moyenne des valeurs dans une colonne. C'est un alias pour `avg`.

```
df.select(mean("salary")).show(truncate=False)
```

```
+-----+  
| avg(salary) |  
+-----+  
| 3400.0       |  
+-----+
```

Fonction d'agrégation skewness

La fonction `skewness()` retourne l'asymétrie des valeurs dans un groupe.

```
df.select(skewness("salary")).show(truncate=False)
```

```
+-----+  
| skewness(salary) |  
+-----+
```

```
| -0.12041791181069571 |  
+-----+
```

Fonctions `stddev()` , `stddev_samp()` et `stddev_pop()`

- `stddev()` : Alias de `stddev_samp` .
- `stddev_samp()` : Retourne l'écart-type de l'échantillon des valeurs dans une colonne.
- `stddev_pop()` : Retourne l'écart-type de la population des valeurs dans une colonne.

```
df.select(stddev("salary"), stddev_samp("salary"), \  
          stddev_pop("salary")).show(truncate=False)
```

Fonction d'agrégation `sum`

La fonction `sum()` retourne la somme de toutes les valeurs dans une colonne.

```
df.select(sum("salary")).show(truncate=False)
```

Fonction d'agrégation `sumDistinct`

La fonction `sumDistinct()` retourne la somme de toutes les valeurs distinctes dans une colonne.

```
df.select(sumDistinct("salary")).show(truncate=False)
```

Fonctions `variance()` , `var_samp()` et `var_pop()`

- `variance()` : Alias de `var_samp` .
- `var_samp()` : Retourne la variance non biaisée des valeurs dans une colonne.
- `var_pop()` : Retourne la variance de la population des valeurs dans une colonne.

```
df.select(variance("salary"), var_samp("salary"), var_pop("salary")) \  
          .show(truncate=False)
```

Code complet

Le code source des exemples d'agrégation PySpark est fourni en bas du message.

Le code source fourni utilise PySpark pour démontrer diverses fonctions d'agrégation sur un DataFrame. Voici une traduction commentée du code :

```
##### Begin head Google-colab  
#  
!wget -q http://archive.apache.org/dist/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz
```

```

!tar xf spark-3.5.0-bin-hadoop3.tgz
#
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!pip install -q findspark
#
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.0-bin-hadoop3"
#
import findspark
findspark.init()
#### end head Google-colab

# Importation de PySpark et des fonctions d'agrégation nécessaires
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import approx_count_distinct, collect_list
from pyspark.sql.functions import collect_set, sum, avg, max, countDistinct, count
from pyspark.sql.functions import first, last, kurtosis, min, mean, skewness
from pyspark.sql.functions import stddev, stddev_samp, stddev_pop, sumDistinct
from pyspark.sql.functions import variance, var_samp, var_pop

# Création d'une session Spark
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

# Création d'un DataFrame avec des données simples
simpleData = [
    ("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)
]
schema = ["employee_name", "department", "salary"]
df = spark.createDataFrame(data=simpleData, schema=schema)

# Affichage du schéma du DataFrame et de ses données
df.printSchema()
df.show(truncate=False)

# Utilisation des différentes fonctions d'agrégation
print("approx_count_distinct: " + str(df.select(approx_count_distinct("salary")).collect()[0][0]))
print("avg: " + str(df.select(avg("salary")).collect()[0][0]))

df.select(collect_list("salary")).show(truncate=False)
df.select(collect_set("salary")).show(truncate=False)

df2 = df.select(countDistinct("department", "salary"))

```

```

df2.show(truncate=False)
print("Distinct Count of Department & Salary: " + str(df2.collect()[0][0]))

print("count: " + str(df.select(count("salary")).collect()[0]))
df.select(first("salary")).show(truncate=False)
df.select(last("salary")).show(truncate=False)
df.select(kurtosis("salary")).show(truncate=False)
df.select(max("salary")).show(truncate=False)
df.select(min("salary")).show(truncate=False)
df.select(mean("salary")).show(truncate=False)
df.select(skewness("salary")).show(truncate=False)
df.select(stddev("salary"), stddev_samp("salary"), stddev_pop("salary")).show(truncate=False)
df.select(sum("salary")).show(truncate=False)
df.select(sumDistinct("salary")).show(truncate=False)
df.select(variance("salary"), var_samp("salary"), var_pop("salary")).show(truncate=False)

```

Conclusion

Ces fonctions permettent d'effectuer divers calculs statistiques et de synthèse sur les données du DataFrame.