Introduction Apache Spark

Systèmes et infrastructures pour les données massives
INF8200 cours 4

J-F Rajotte 2024-01

Contenu

- Retour sur Hadoop MapReduce
- Introduction à Spark
  - Survol des composantes
  - RDD
  - DataFrame
  - Spark SQL

Retour sur MapReduce

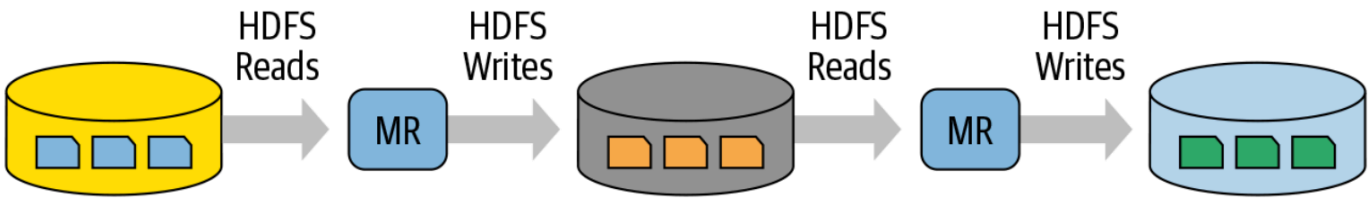
MapReduce : avantages

- Pour des applications relativement simples sur des données massives
- Pour de grandes grappes de calculs
- Tolérance aux pannes

Retour sur MapReduce

MapReduce : désavantages

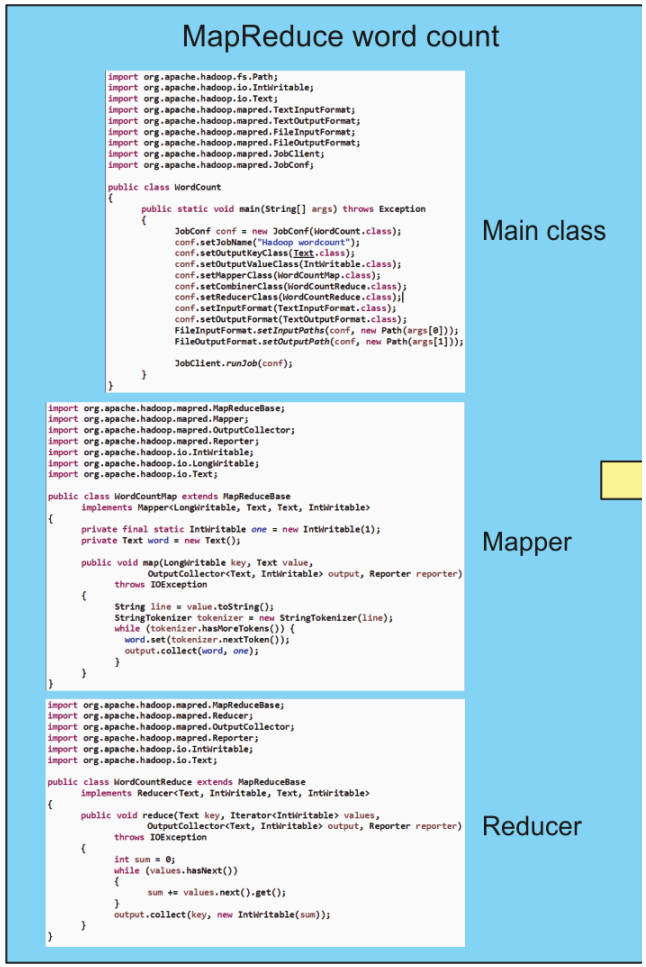
- Les résultats de chaque phase de MapReduce doivent être stocké sur HDFS pour être réutilisées immédiatement après.
- Généralement lent
- Relativement complex à programmer



Retour sur MapReduce

MapReduce : désavantages

- Relativement complex à programmer
- Résultats seulement à la fin (long avant d'avoir un retour)

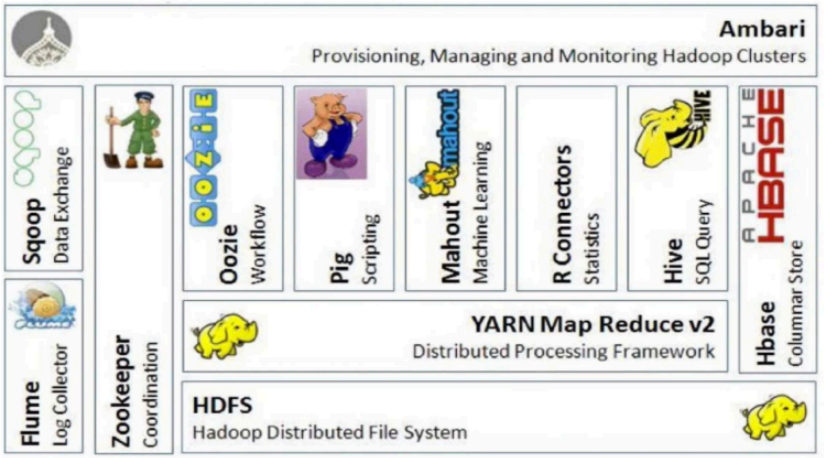


Retour sur MapReduce

MapReduce : désavantages

Les Applications au-delà du simple “batch processing” sont développées par des projets séparés avec leur propres API.

- Ajoute à la complexité
- Difficile à apprendre



Le projet Spark
Motivation

- Comme Hadoop, c'est un Framework de calcul distribué généraliste
- Buts :
  - Conserver les avantages de MapReduce
    - Résistance aux pannes
    - Parallèlisme
  - Améliorations
    - Stockage des résultats intermédiaires en mémoire
    - Plus d'applications avec APIs compatible
    - Plusieurs languages : Scala, Java, Python, SQL, R
    - Plus facile d'utilisation

Le projet Spark

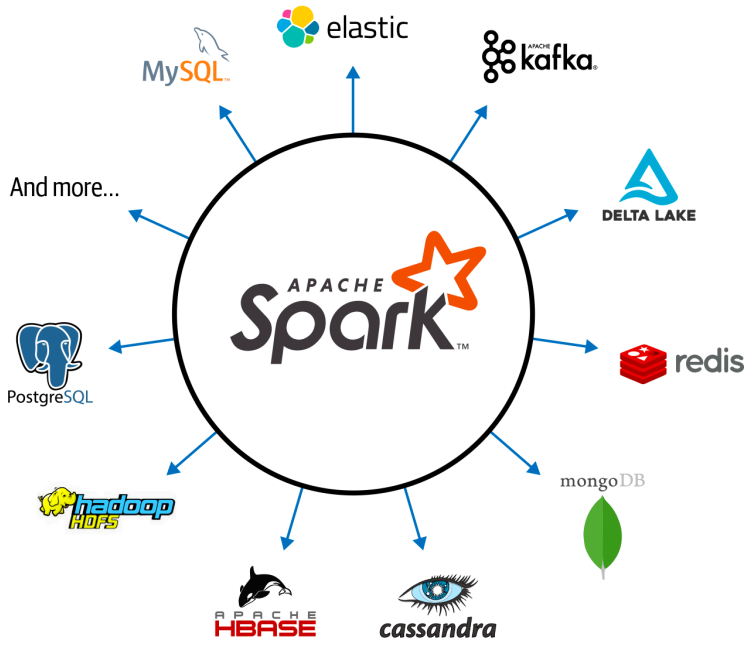
Historique

- 2009 Projet Spark @ UC Berkeley
- 2013 Spark est donné à Apache Software Foundation
  - Les créateurs démarrent la compagnie **Databricks**
- 2014 Apache Spark 1.0
- 2020 Apache Spark 3.0
  - 80% des fonctions de Pandas sont implémentées
  - Support pour GPU (Apprentissage profond)
- Présent dernière version (2023) Apache Spark 3.5

Le projet Spark

Se concentre seulement sur le traitement

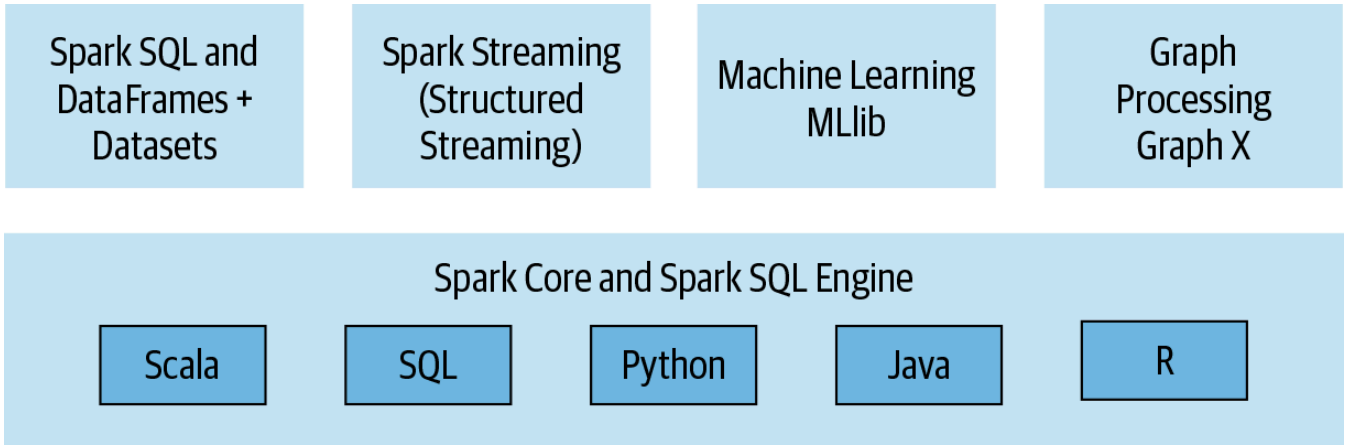
- Pas le stockage (contrairement à Hadoop)
- Adapté à plusieurs sources



Les composantes de Spark

Stack unifié

- Permets des librairies unifiées provenant de 5 modules



Spark SQL

Pour les données structurées

- Lectures de données de plusieurs format : csv, text, json, parquet…
- Creation de tables manipulables avec des requêtes de type SQL
  - Même avec les APIs des autres languages (Python, Scala, R…)

```
// In Scala
// Read data off Amazon S3 bucket into a Spark DataFrame
spark.read.json("s3://apache_spark/data/committers.json")
  .createOrReplaceTempView("committers")
// Issue a SQL query and return the result as a Spark DataFrame
val results = spark.sql("""SELECT name, org, module, release, num_commits
    FROM committers WHERE module = 'mllib' AND num_commits > 10
    ORDER BY num_commits DESC""")
```

Spark MLlib

Implémentation d'algorithmes d'apprentissage automatique

- Extraction et transformation d'attributs
- Opérations d'algèbre linéaire et statistiques

```
# In Python
from pyspark.ml.classification import LogisticRegression
...
training = spark.read.csv("s3://...")
test = spark.read.csv("s3://...")

# Load training data
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Predict
lrModel.transform(test)
```

Spark Structure Streaming

Combiner des données statiques et sources de flux

- Le flux de données est considéré comme une table augmentant continuellement avec de nouvelles rangées ajoutées
- L'utilisateur peux considérer qu'il envoie des requêtes à une table statique.

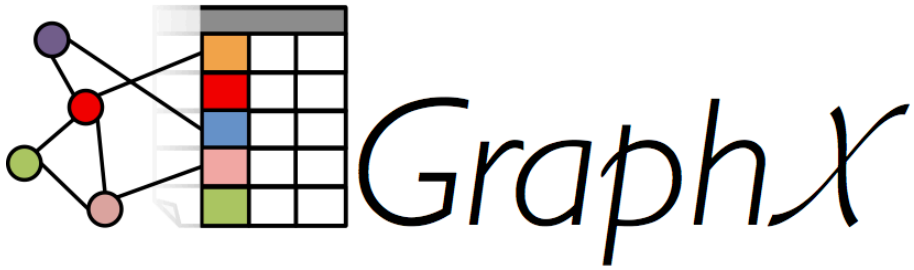
```
# In Python
# Read a stream from a local host
from pyspark.sql.functions import explode, split
lines = (spark
  .readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load())

# Perform transformation
# Split the lines into words
words = lines.select(explode(split(lines.value, " ")).alias("word"))
# Generate running word count
word_counts = words.groupBy("word").count()
```

Spark GraphX

Manipulation de graphes

- Implémentation d'algorithmes standards de graphe

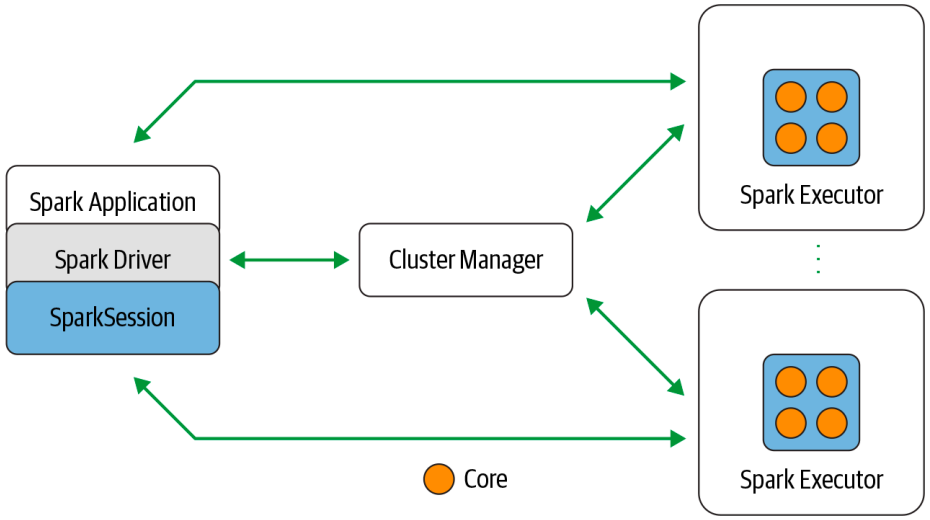



```
// In Scala
val graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
val graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Source image: https://spark.apache.org/docs/latest/graphx-programming-guide.html

L’exécution distribuée de Spark

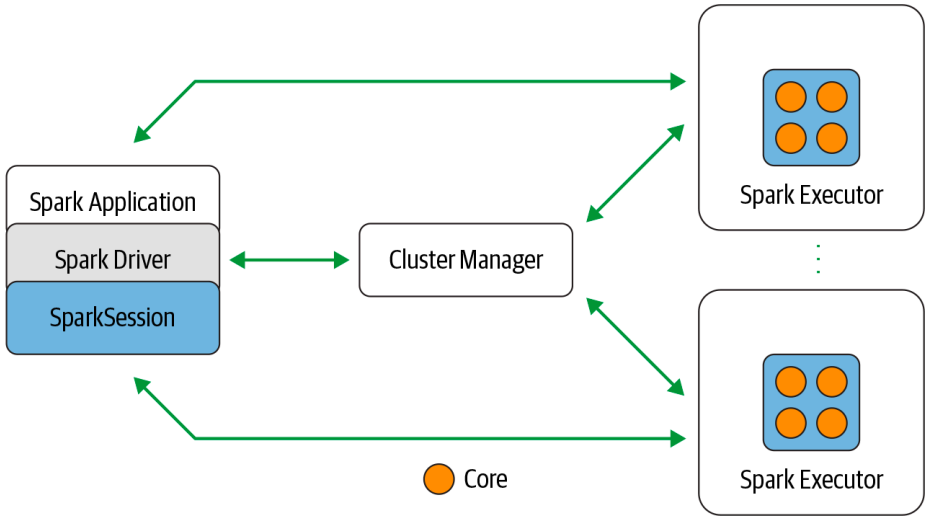
- Une application consiste à un **Driver** orchestrant les **Spark Executors** et le **Cluster Manager** en communicant à l’aide d’une **Spark Session.**



Spark Driver

- Responsable de démarrer la **SparkSession**
- Communiques avec le **Cluster Manager** pour demander des ressources (CPU, mémoire)
- Organize le calcul (DAG)
- Distribue les tâches aux Spark Executors

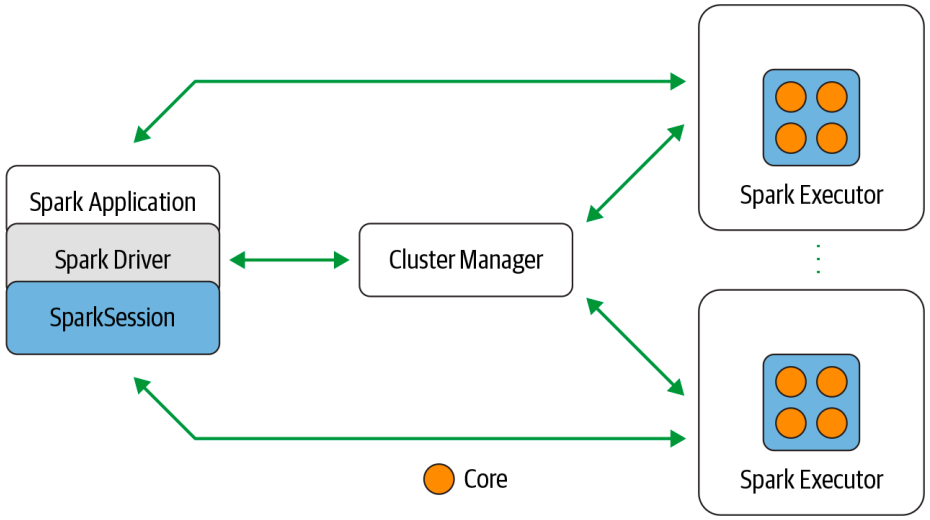
Note: une fois les ressources allouées, le driver communique directement avec les executors



SparkSession

Point d'entrée unifié pour toutes les fonctionnalités de Spark

- Création de Dataframes, Base de données, Tables…
- Lecture de sources de données
- Envoyer des requêtes SQL



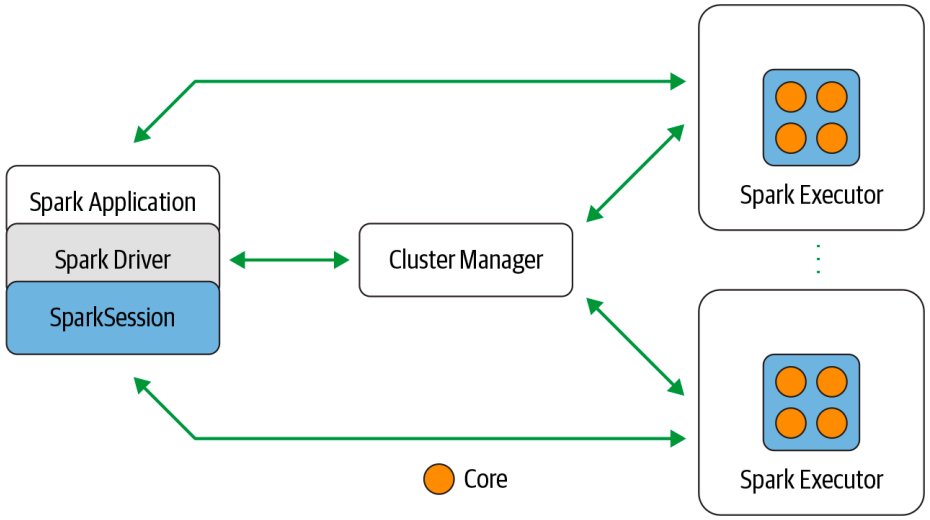
Cluster manager

Responsable de la gestion et l'allocation des ressources

Spark supporte les gestionnaires de ressources suivants

- YARN
- Apache Mesos
- Kubernetes

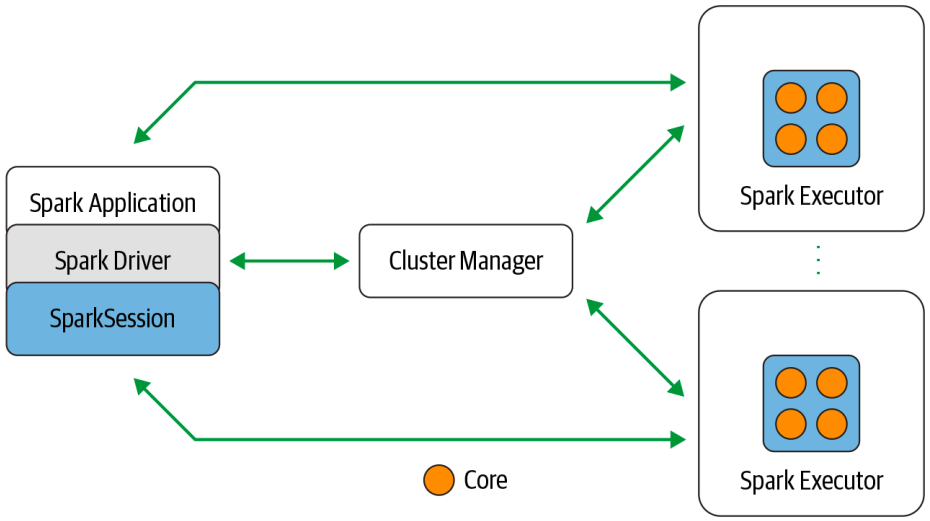
Note : Il y a aussi un "built-in stand-alone cluster manager"



Spark Executors

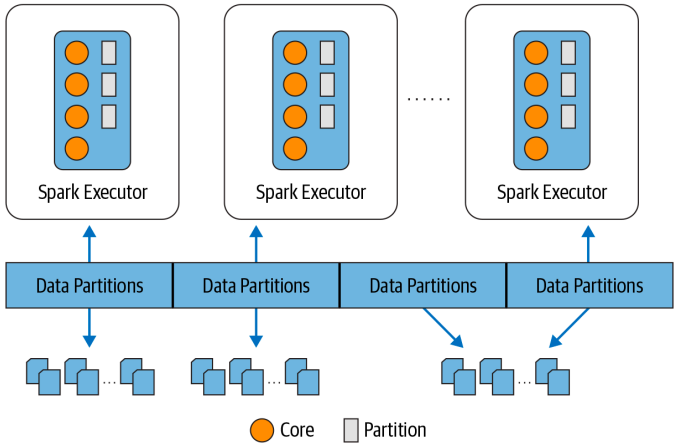
Programme sur noeuds de calcul

- Responsable pour exécuter les tâches sur les noeuds
- Communique avec le programme du driver



Distribution des données

Comme pour MapReduce, le traitement d'un Spark Executor se fait idéalement sur les chunks *près* de lui.




```
# In Python
log_df =
spark.read.text("path_to_large_tex
t_file").repartition(8)
print(log_df.rdd.getNumPartitions(
))
```

Note: la function "repartition" redéfini le nombre de chunks sur le cluster

Les structures de Spark

RDD

Resilient Distributed Dataset

L’abstraction fondamental de Spark pouvant être traitée de manière distribuée

- **Résiliant** : Si la donnée dans la mémoire est perdue, elle peut être recrée
  - Un RDD garde son information de lineage (processus de création), il peut donc être recréé s’il y a une panne
  - À noter la différence avec MapReduce qui doit reprendre du début sur une copie (réplica) des données
- **Distribué** sur le cluster
- **Dataset** : les données initiales peuvent provenir d’un fichier ou être créées

RDD

Les bénéfices de l'immuabilité

- **Tolérance aux pannes** : Créé une fois et peuvent être recréé en tout temps
- **Exactitude** (Correctness) : Évite les problèmes d'incohérences (e.g. pas d'update simultanés)
- **Performance** : Peut être mis en mémoire et partagé par plusieurs tâches

RDD

Modes de création

- À partir de la ligne de commande

```
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30), ("TD", 35), ("Brooke", 25)])
```

- À partir d'un ou plusieurs fichiers

```
dataRDD = sc.textFile("mon_fichier.txt")
```

- À partir de données en mémoire (résultat d'une tâche)
- À partir d'un autre RDD

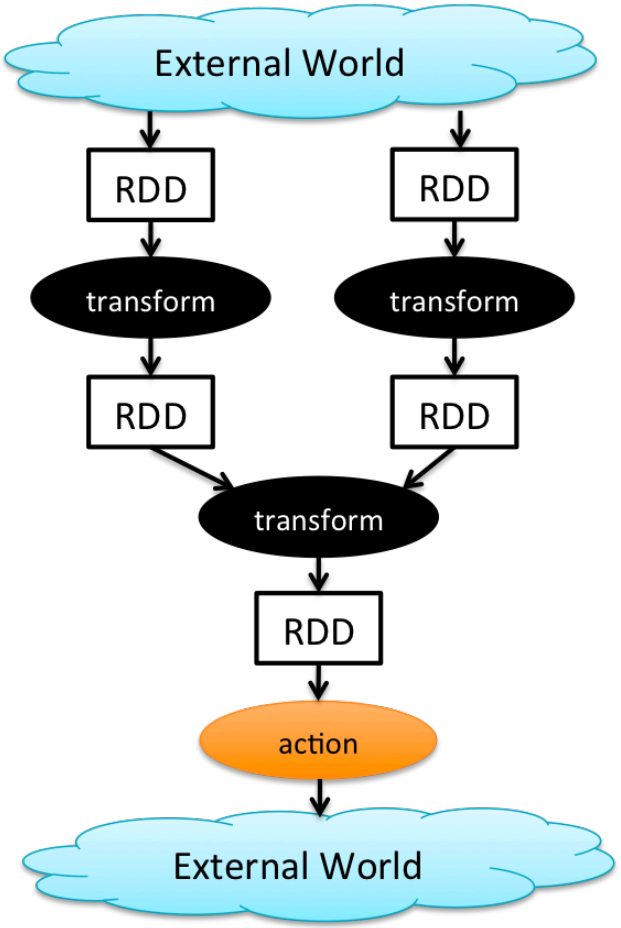
```
agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

Pas très évident à manipuler et ne correspond pas à la convivialité de l'esprit de Spark

RDD

Deux types d'opérations

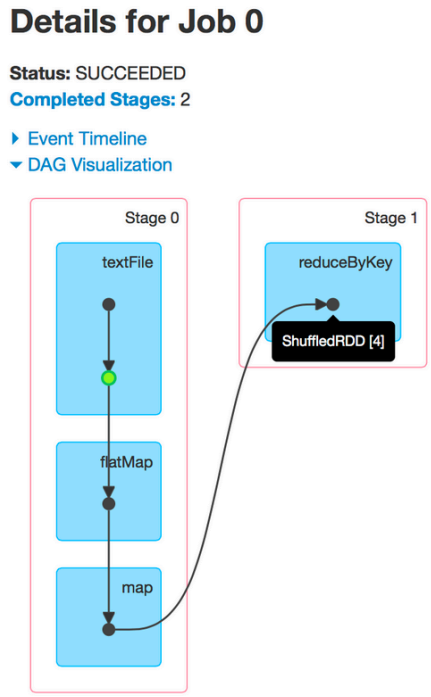
- Transformation
  - Retourne toujours un RDD
  - Exemple : filter
- Action
  - Mène à un traitement (computation)
  - Retourne un résultat
  - Exemple : count



RDD

DAG Directed Acyclic Graph (Graphe orienté acyclique)

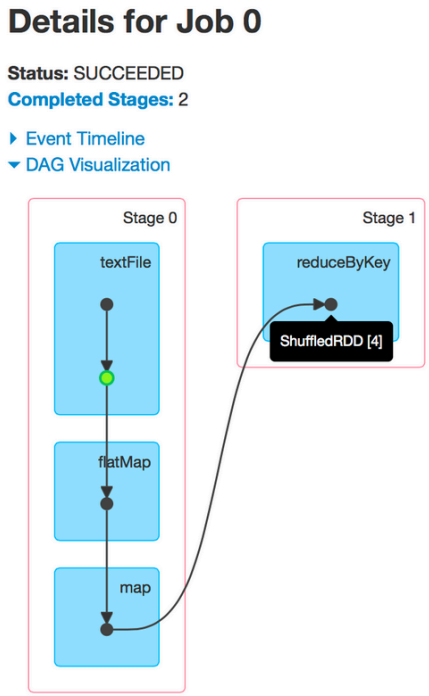
- Chaine de dépendance des RDD
- A droite : un simple comptage de mot (word count)
  - Lecture d'un fichier texte (e.g. sur HDFS)
  - Transform : FlatMap séparant chaque ligne en mot
  - Transform : Map créant les pairs clé-valeur (mot, 1)
  - Transform : "Reduce by key" somme la valeur pour chaque mot
  - Action : (pas visible dans le DAG) "collect" obtenir les résultats sur le driver.



RDD

DAG Directed Acyclic Graph (Graphe orienté acyclique)

- Éléments du DAG (word count)
  - Carré bleu = opération
  - Points = RDD créé par l'opération
  - Stage = Série d'opération complètement parallèle délimitée par les shuffle (i.e. quand les données doivent être redistribuées sur le cluster)
  - Point vert : RDD mis en mémoire (cache)



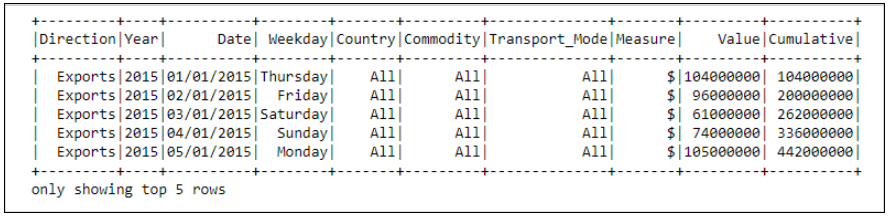
Un nouveau "Stage" chaque fois qu'il y a un shuffle

DataFrame

DataFrame

Inspiré des DataFrame Pandas

- Peut être considéré comme une table distribuée
- Chaque colonne a son propre nom et type: int, float, string…



Direction	Year	Date	Weekday	Country	Commodity	Transport_Mode	Measure	Value	Cumulative
Exports	2015	01/01/2015	Thursday	All	All	All	$	104000000	104000000
Exports	2015	02/01/2015	Friday	All	All	All	$	96000000	200000000
Exports	2015	03/01/2015	Saturday	All	All	All	$	61000000	262000000
Exports	2015	04/01/2015	Sunday	All	All	All	$	74000000	336000000
Exports	2015	05/01/2015	Monday	All	All	All	$	105000000	442000000

only showing top 5 rows

DataFrame

Plus lisible

API DataFrame

```
from pyspark.sql.functions import avg
# Create a DataFrame using SparkSession
data_df = spark.createDataFrame([("Brooke", 20),
("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their
ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

API RDD

```
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny",
31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their
lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] +
y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

Création de DataFrame
Schema

- **Inféré**
  - Doit passer sur les données
  - Spark doit créer un job pour lire et inférer le type de chaque colonne
  - Option d'inférer sur une fraction des données (samplingRatio)
- **Explicite**
  - Définir le type de chaque colonne
  - Plus rapide
  - Aide à la détection d'erreur
  - Souvent pour les sources de données externes

Création de DataFrame

Lecture de fichier

- Plusieurs formats possibles: csv, parquet, json, text, Avro…
- Plusieurs sources : NoSQL, RDBMS, streaming…
- Pour un fichier CSV

```
DF = spark
  .read
  .option("samplingRatio", 0.001)
  .option("header", true)
  .csv("""file.csv""")
```

Création de DataFrame

Écriture de fichier

- Même formats que la lecture
  - Pour parquet, le schema est préservé dans les métadonnées
- Pour un fichier parquet

```
DF.write.format("parquet").save(parquetPath)
```

Création de DataFrame

Schema

- Programatiquement

```
from pyspark.sql.types import *
schema = StructType([StructField("author", StringType(), False),
  StructField("title", StringType(), False),
  StructField("pages", IntegerType(), False)])
```

- Avec un string DDL (Data Definition Language)

```
schema = "author STRING, title STRING, pages INT"
```

DataFrame
Colonne

- Défini par le type *Column*
- Voir les colonnes (la commande suivante retourne une liste de strings)

```
DF.columns
```

- Sélectionner une colonne (la commande suivante retourne un object *Column*)

```
DF.col(“Id”)
```

- Manipuler une colonne

```
DF.select(expr("Hits * 2”))
#ou
DF.select(col("Hits") * 2)
```

- Créer une colonne à partir de plusieurs colonnes

```
DF.withColumn("AuthorsId", (concat(expr("First"), expr("Last"), expr("Id"))))
```

DataFrame
Rangée (Row)

- Une rangée est un object Row dans Spark.
- Chaque élément peut avoir un type différent
- Chaque élément peut être accédé par un indice
- Une rangée peut être créée à partir d'une liste

```
from pyspark.sql import Row
blog_row = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568, "3/2/2015",
  ["twitter", "LinkedIn"])
# access using index for individual items
blog_row[1]
```

- Un DataFrame peut être créé avec des Rows

```
rows = [Row("Matei Zaharia", "CA"), Row("Reynold Xin", "CA")]
authors_df = spark.createDataFrame(rows, ["Authors", "State"])
authors_df.show()
```

DataFrame

Renommer, ajouter et éliminer des colonnes

- Motivation : convention, lisibilité, nettoyage…
  - e.g. les espaces ou les virgules dans les noms de colonnes pourrait mener à des problèmes si on sauvegarde dans certains formats de fichiers
- Deux manières :
  - 1) avec le schema
  - 2) avec la méthode withColumnRenamed()

```
df_renamed = df.withColumnRenamed("Delay", “ResponseDelayedinMins")
```

Note : Puisque les DataFrame sont immuable, la méthode retourne un nouveau DataFrame avec la nouvelle colonne

DataFrame

Manipulation et modifications de colonnes

- Transformer les string de temps en format approprié avec les fonctions comme to_timestamp() et to_date()

```
ts_df = (df
  .withColumn("IncidentDate", to_timestamp(col("CallDate"), "MM/dd/yyyy"))
  .drop("CallDate")
  .withColumn("OnWatchDate", to_timestamp(col("WatchDate"), "MM/dd/yyyy"))
  .drop("WatchDate")
  .withColumn("AvailableDtTS", to_timestamp(col("AvailableDtTm"),
  "MM/dd/yyyy hh:mm:ss a"))
  .drop("AvailableDtTm"))
```

DataFrame
Aggregation

- Les DataFrames permettent beaucoup de transformations permettant l'agrégation avec groupBy

```
(fire_df
  .select("CallType")
  .where(col("CallType").isNotNull())
  .groupBy("CallType")
  .count()
  .orderBy("count", ascending=False)
  .show(n=10, truncate=False))
```

- Autres méthodes d'agrégation :
  - De base: min(), max(), sum(), avg()…
  - Avancés: correlation(), covariance()…

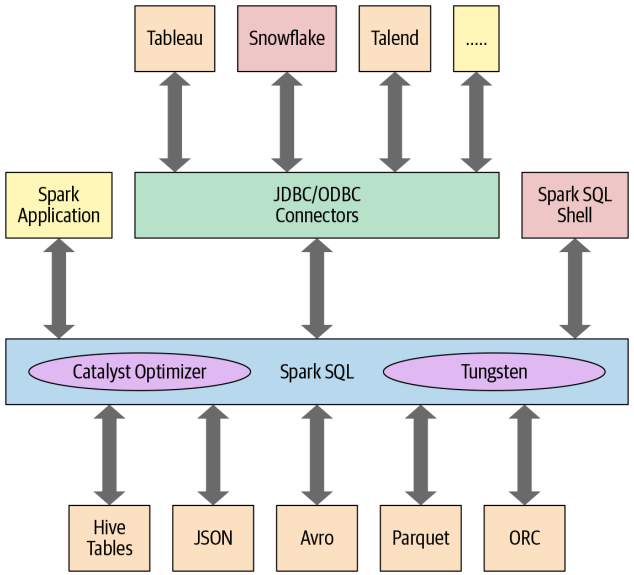
DataFrame
Accéder au RDD

- Motivation
  - Connection avec une application externe basée sur les RDD
  - Forcer *comment* faire une requête plutôt que l'*objectif* de la requête
    - Comment (rdd) : sum(elements) / nombre(éléments)
    - Objectif (dataframe): elements.mean()
- Df.rdd()

Spark SQL

Spark SQL

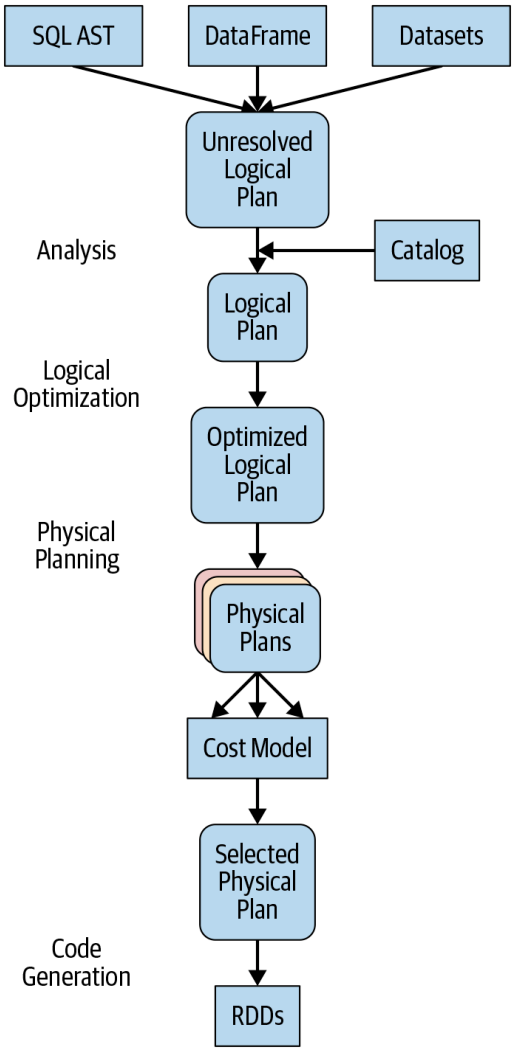
- Permet d'envoyer des requêtes SQL sur des données structurées (avec un schéma), exemple : dataframe
- Accès aux tables
- Lecture et écriture de plusieurs formats (json, csv, text, Avro, Parquet…)
- Permet une exploration SQL interactive
- Fait le pont avec des base de données externes



Spark SQL

Optimizer Catalyst : un compilateur

- Traduit une requête en plan de traitement (execution plan) en 4 phases:
  - Analyse
    - Relier les colonnes (ou tables) au catalogue (interface donnant accès au nom de colonne, leur type, fonctions…).
  - Optimization logique
    - Choisi le plan d'exécution au moindre coût
  - Planification physique
    - Expliciter les opérations à faire sur chacune des machine physique
  - Génération de code
    - Java bytecode pour le traitement sur chaque machine



Spark SQL
Méthode sql()

- SparkSession : point d'entrée unifié pour programmer avec Spark
- Les requêtes sql se font à partir de la méthode sql() d'une instance de SparkSession, habituellement appelé spark:

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
spark.sql("SELECT * FROM myTableName")
```

- Toutes les requêtes sql retournent un dataframe

Tables
Deux types

- Tables gérées
  - Spark gère les données et les métadonnées
  - Stockages possible : Système de fichier local, HDFS, Amazone S3…
- Non gérées
  - Spark ne gère que les métadonnées
  - Gestion des données est externe, exemple fichier local, Cassandra…

Tables

Base de données

- Toutes les tables résident dans une base de données
- Par défaut, les tables sont créés dans la base de données “default”
- On peut créer sa propre base de données

```
spark.sql("CREATE DATABASE maBD")
spark.sql("USE maBD")
```

  - Après ces commandes, toutes les tables créées seront dans la BD nommé “maBD”

Création de tables

- Table gérée
  - Table vide

```
spark.sql("CREATE TABLE managed_tbl (date STRING, qty INT)”)
```

  - À partir d’un DataFrame

```
df.write.saveAsTable("managed_tbl")
```

- Table non-gérée
  - À partir d’un fichier csv

```
spark.sql("""CREATE TABLE us_delay_flights_tbl(date STRING, delay INT,
  distance INT, origin STRING, destination STRING)
  USING csv OPTIONS (PATH
  '/databricks-datasets/learning-spark-v2/flights/departuredelays.csv')""")
```

Tables temporaires (Views)

- Ne contiennent pas les données
- Même requêtes que pour les tables normales
- Exemples d'utilité :
  - Exploration interactive
  - un sous-ensemble d'une table

Tables temporaires (Views)

Les tables temporaires disparaissent avec SparkSession

- À partir d'un DataFrame:
```
df.createOrReplaceTempView("mytable")
```
- Verifier
```
spark.sql("show tables").show()
```
- Lecture:
```
df2 = spark.sql("select * from mytable")
```
- Effacer
```
spark.catalog.dropTempView("mytable")
```

Table temporaires

Globales ou limité à la session (session-scoped)

- Globale
  - Visible par toutes les SparkSession

```
spark.sql("CREATE OR REPLACE GLOBAL TEMP VIEW my_tmp_view AS SELECT * from full_table WHERE country = 'Canada')
```

  - Acceder par la base de données globale

```
spark.sql("SELECT * FROM global_temp.my_tmp_view")
```

- Session-scoped
  - Créer

```
spark.sql("CREATE OR REPLACE TEMP VIEW my_tmp_view AS SELECT * from full_table WHERE country = 'Canada')
```

  - Accéder

```
spark.sql("SELECT * FROM my_tmp_view")
```

Tables

Catalog

- Les métadonnées des tables sont accessibles dans le catalog
- Exemple d'informations

```
spark.catalog.listDatabases()
spark.catalog.listTables()
spark.catalog.listColumns("us_delay_flights_tbl")
```

Caching

- Les tables peuvent être mises, et fixées, en mémoire pour un accès rapide
- Pratique si on prévoit faire des requêtes fréquentes, exemple: pour des explorations en session interactives
- Il est possible de faire le “caching” de manière LAZY, c’est-à-dire seulement après la première requête

```
CACHE [LAZY] TABLE <table-name>
UNCACHE TABLE <table-name>
```

Sources externes

Interaction avec sources externes
JDBC et BD SQL

- Spark a un API qui peut lire des bases de données extérieurs en utilisant JDBC
  - Java DataBase Connectivity
  - Il faut spécifier le Driver
- Les requêtes retournent un DataFrame
- Les tables peuvent être transférées comme DataFrame ou View
- Attention: Toutes les données passeront par une connection

Interaction avec sources externes
JDBC et BD SQL

- PostgreSQL, MySQL, CosmosDB, MS SQL Server
  - On doit télécharger le “jar” approprié
- Exemple pour MySQL

```
jdbcDF = (spark
  .read
  .format("jdbc")
  .option("url", "jdbc:mysql://[DBSERVER]:3306/[DATABASE]")
  .option("driver", "com.mysql.jdbc.Driver")
  .option("dbtable", "[TABLENAME]")
  .option("user", "[USERNAME]")
  .option("password", "[PASSWORD]")
  .load())
```

Interaction avec sources externes

Beaucoup d'autres options

- Beeline
- Tableau
- Azure Cosmos DB
- Cassandra
- Snowflake
- MongoDB
- …

Configuration et performance

Configuration de Spark SQL

Modification

- On peut y accéder avec la command “SET” et le paramètre “-v” retournant la configuration de Spark SQL en pairs clé,valeur

```
spark.sql("SET -v").select("key", "value").show(n=5, truncate=False)
```

```
+-----------------------------------------------------------+-----------+
|key                                                        |value      |
+-----------------------------------------------------------+-----------+
|spark.sql.adaptive.enabled                                 |false      |
|spark.sql.adaptive.nonEmptyPartitionRatioForBroadcastJoin  |0.2        |
|spark.sql.adaptive.shuffle.fetchShuffleBlocksInBatch.enabled|true       |
|spark.sql.adaptive.shuffle.localShuffleReader.enabled      |true       |
|spark.sql.adaptive.shuffle.maxNumPostShufflePartitions     |<undefined>|
+-----------------------------------------------------------+-----------+
```

- Ou, pour accéder à une configuration

```
spark.conf.get("spark.sql.shuffle.partitions")
```

Configuration de Spark SQL

Modification de paramètres

- On peut voir si un paramètre est modifiable avec la command

```
spark.conf.isModifiable("<config_name>")
```

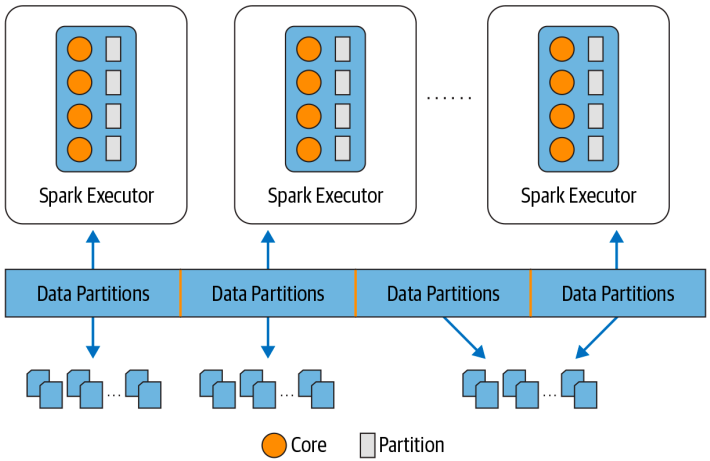
- Modification

```
spark.conf.get("spark.sql.shuffle.partitions")
'200'
spark.conf.set("spark.sql.shuffle.partitions", 5)
spark.conf.get("spark.sql.shuffle.partitions")
'5'
```

Configuration de Spark SQL

Partition

- Une des source de l'efficacité de Spark est sa capacité de traiter plusieurs tâches en parallèle
- Ce parallélisme est lié à la partition des données: au mieux un tâche par coeur (core) et une tâche par partition
- Idéalement, autant de partition que de coeurs par exécuteur



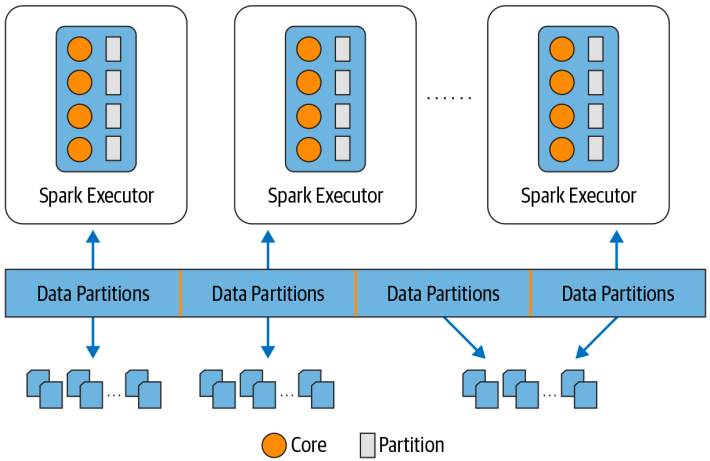
Configuration de Spark SQL

Partition

Configuration de la partition

- Par le paramètre spark.sql.files.maxPartitionBytes
  - 128Mb par défaut
- Aussi contrôlable par certains APIs, ex: DataFrame

```
ds = spark.read.textFile("../
README.md").repartition(16)
```



Configuration de Spark SQL
Caching

DataFrame.cache()

- Mettra en mémoire le plus de partitions en mémoire sur les exécuteurs
- Une partition **ne peut pas** être partiellement mis en mémoire
- Un DataFrame **peut** être partiellement mis en mémoire
  - Les partitions “non-cache” devront être re-calculées

- Lorsqu’on utilise la fonction cache, le DataFrame n’est pas en mémoire tant qu’une action n’a pas été appelée
  - Attention! Seule les données accedées par l’action seront mis en mémoire

Configuration de Spark SQL
Caching

Note sur le cache

- Lorsqu'on utilise la fonction cache, le DataFrame n'est pas en mémoire tant qu'une action n'a pas été appelée, exemple df.count()
- **Attention!** Seule les partitions accédées par l'action seront mis en mémoire
  - Exemple:
    - df.take(1) retournant la première ligne (row) du DataFrame
    - Seulement une partition (celle contenant la ligne) sera mise en mémoire

Configuration de Spark SQL
Persist

DataFrame.persist()

- persist une comme cache mais avec un contrôle sur le niveau de stockage grace au paramètre StorageLevel.*LEVEL*
- Quelques options:
  `MEMORY_ONLY`
  `DISK_ONLY`
- Possibilité de réplication avec l'argument LEVEL_NAME_2, ex: MEMORY_ONLY_2
  - Permet à Spark de rouler les jobs sur la copie
  - Meilleurs tolérance aux pannes
  - Traitement supplémentaire nécessaire

Configuration de Spark SQL

Tables et Views

Similairement, les tables et Views peuvent être *cache*

```
df.createOrReplaceTempView("dfTable")
spark.sql("CACHE TABLE dfTable")
spark.sql("SELECT count(*) FROM dfTable").show()
```

Enfin on peut libérer la mémoire ou le disque

- DataFrame.unpersist() (fonctionne aussi pour cache)

```
spark.sql("UNCACHE TABLE dfTable")
```

Configuration de Spark SQL

Cacher ou ne pas cacher (ou persister)

- Quand *cache* ou *persist*:
  - Dataframe utilisé à répétition, exemple entraînement d'apprentissage automatique
  - DataFrame utilisé fréquemment dans un ETL ou un pipeline
- Quand **ne pas** *cache* ou *persist*:
  - DataFrame trop grand pour la mémoire
  - Utilisation peu fréquente