



L'apparition du Big Data

Programme



- Pourquoi le Big Data ?
- Les systèmes distribués
- Map Reduce
- Hadoop



La problématique



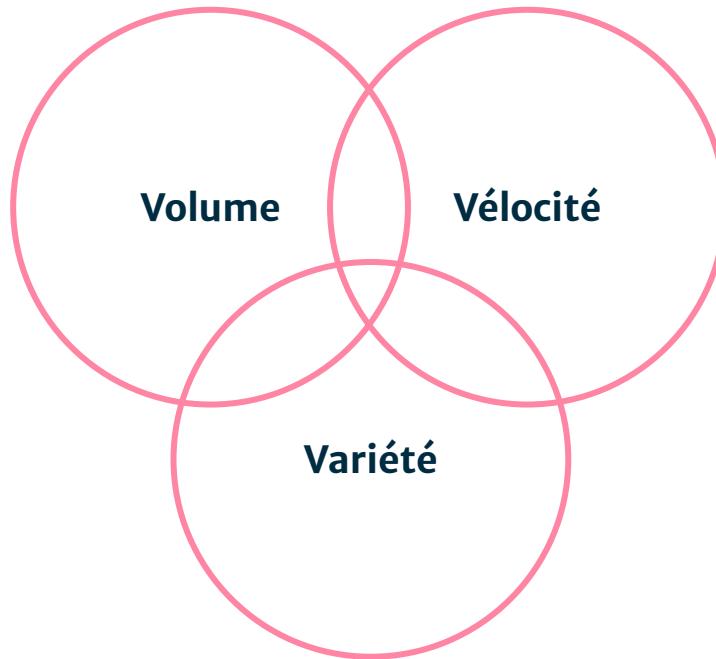
Le déluge de données

Nous générerons de plus en plus de données:

- Transactions financières
- Événement d'équipement réseaux
- IOT
- Log serveur
- Click Stream (Navigation Web)
- E-mail et formulaire web
- Données issues des réseaux sociaux



Le problème





Le problème : Volume

Finances

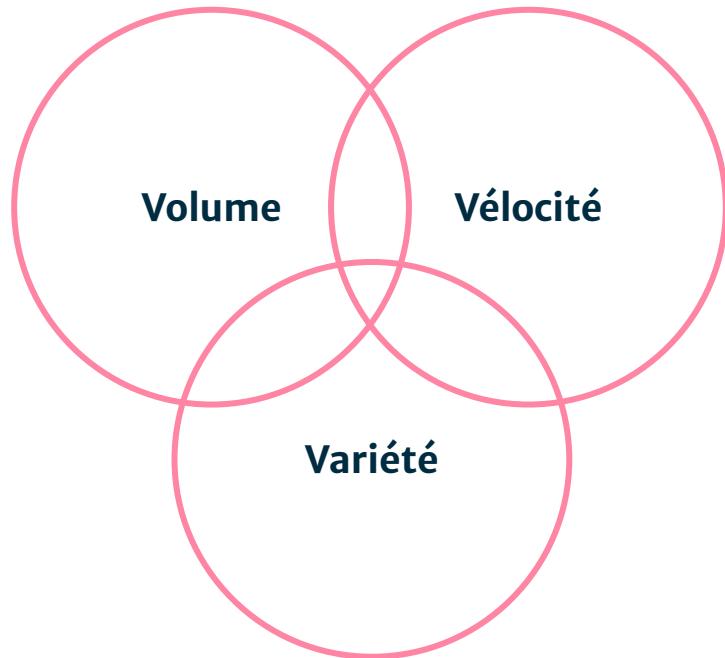
- Presque 4 milliards d'actions échangées par jour à la bourse de New York

Facebook

- 2013 = 10 To /Jour

Twitter

- 400 000 tweets écrits par minute IoT
- Tesla a capturé des informations issues de plus d'1 milliard de km





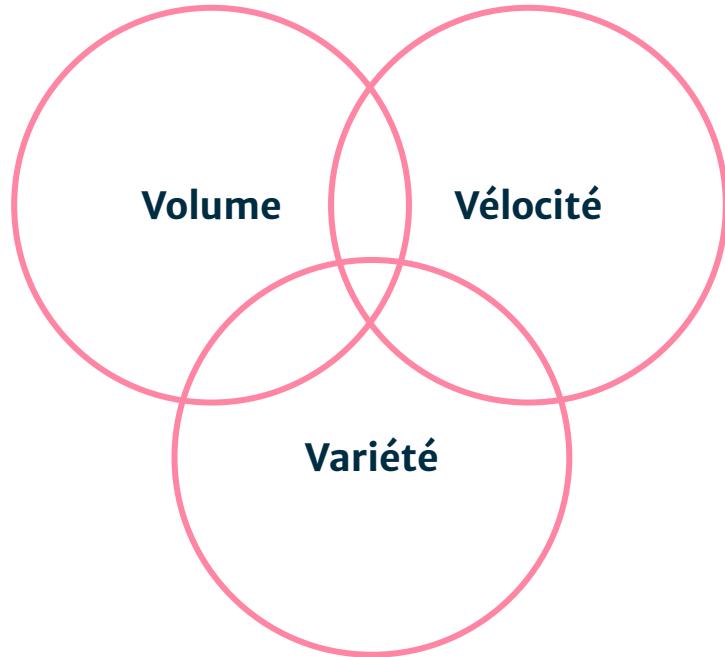
Le problème : Variété

Données Structurées

- BDD

Données Non Structurées

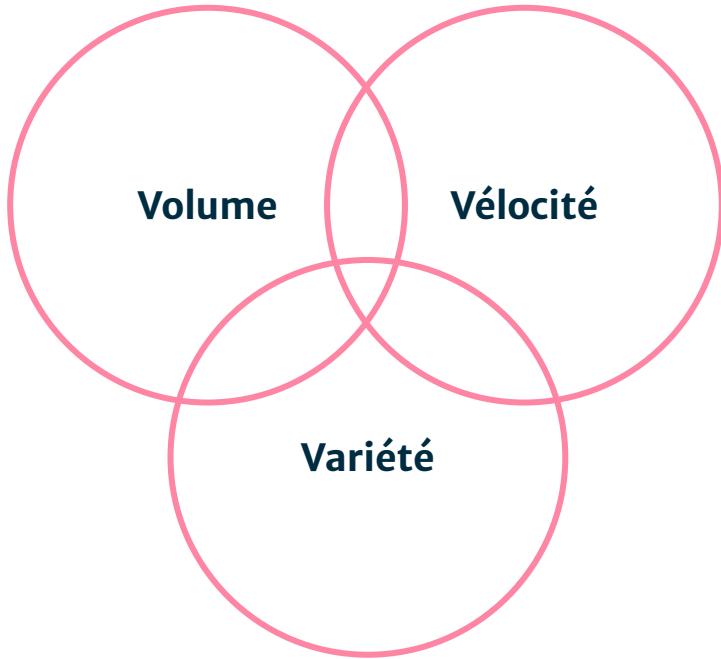
- Page Web
- Log
- Image
- Vidéo





Le problème : Vélocité

- Fréquence de mise à jour
- Temps Réel





Les systèmes distribués

Croissance verticale & horizontale



Croissance verticale

- CPU plus rapide
- Plus de mémoire
- Programmation simple
- Limité par le matériel
- Faible volume

Croissance horizontale

- Gros volume
- Plusieurs machines
- Programmation complexe
 - Gestion des crashes
 - Distribution des calculs



Problème de la donnée

- Traditionnellement centralisé
- Transfert de donnée pour les traitements
- Bande passante réseau limité

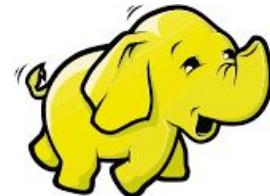
Besoin

- Facilement scalable
- Tolérant à la panne
- Rentable (hardware peu coûteux)



Naissance de Hadoop

- GFS (2003)
- MapReduce (2004)
- Hadoop (2006)
 - HDFS
 - Hadoop MapReduce



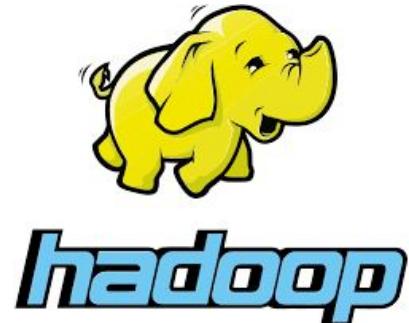
hadoop

Qu'est-ce que Hadoop ?



- **Plateforme de Stockage et de Calcul Distribué**
 - Grand volume de donnée de manière résiliente
 - Permet de se concentrer sur les problèmes business et non infra

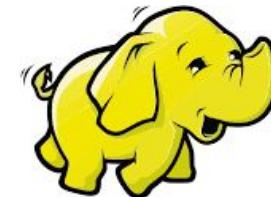
- **Different cas d'usage possible**
 - Extract Transform Load
 - Business Intelligence
 - Stockage
 - Machine Learning
 - ...





Hadoop est scalable

- Ajout facile de noeud
- Augmentation de ressource = augmentation de performance
- Gestion des échecs
 - Le système continue de fonctionner
 - La tâche est attribuée à un autre noeud
 - Pas de perte de donnée car réPLICATION



hadoop

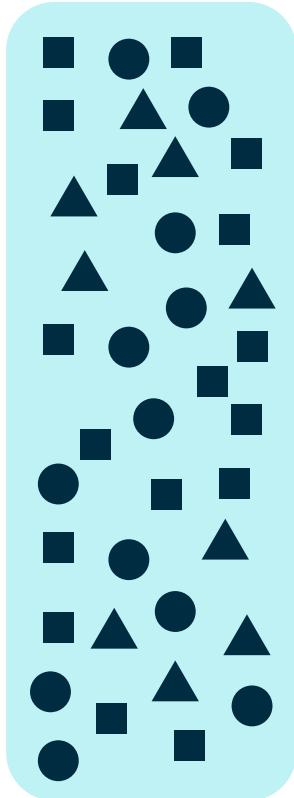


Map Reduce

Map Reduce

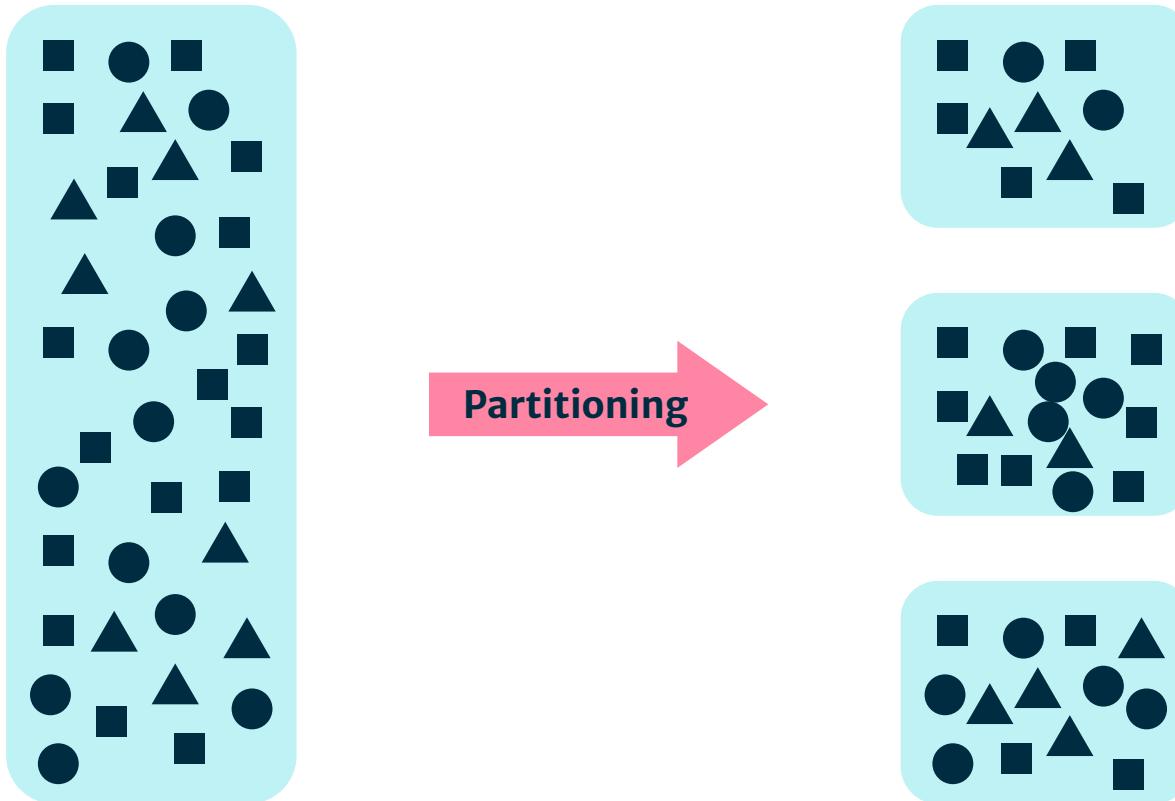


- Modèle de programmation inventé par Google en 2004
 - Utilisé pour paralléliser les calculs sur des systèmes de fichiers distribués
-
- **Map** : Projection (Select), Filter (Where)
 - **Reduce** : Aggregation (Group By), Join



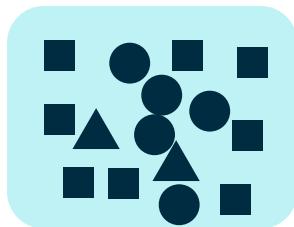
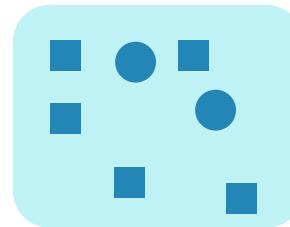
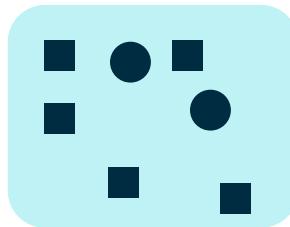
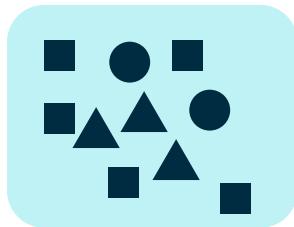
**Combien de cercles
et de carrés ?**

Map Reduce

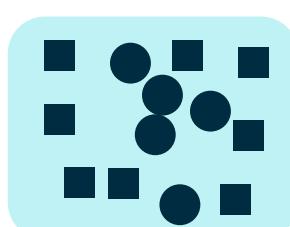




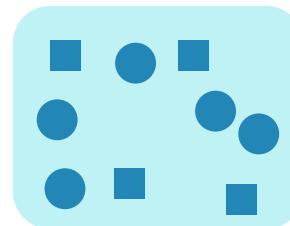
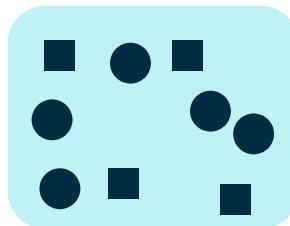
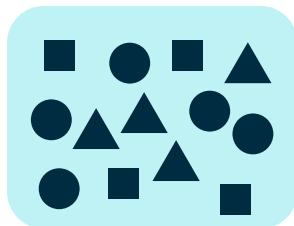
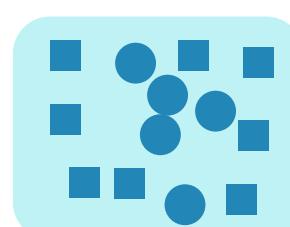
Map Reduce



Filter

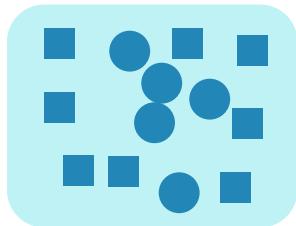
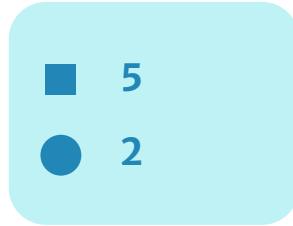
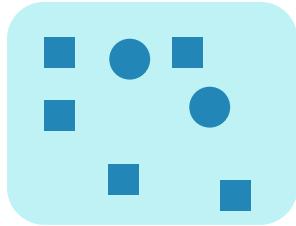


Map

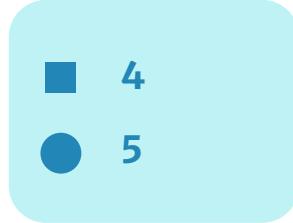
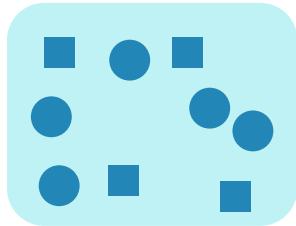
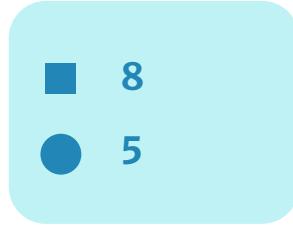




Map Reduce

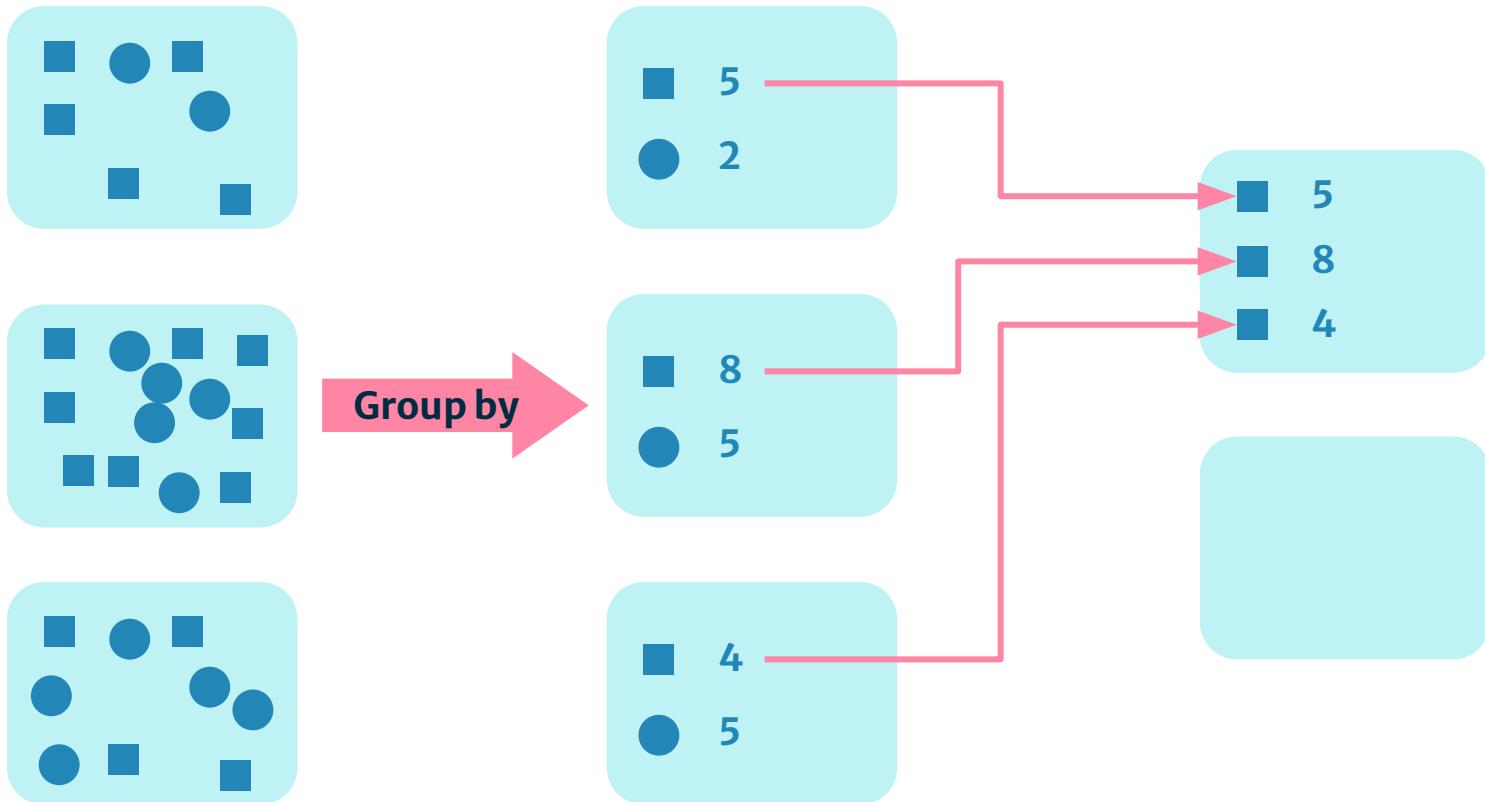


Group by



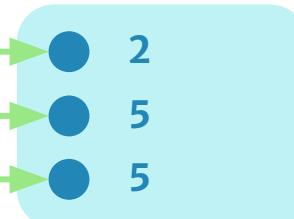
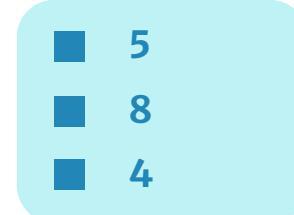
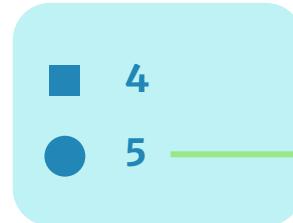
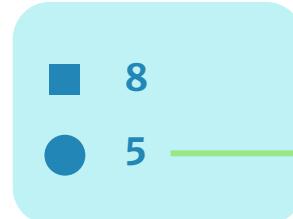
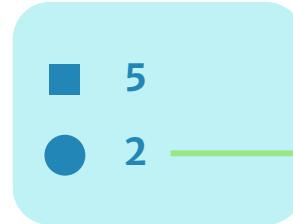
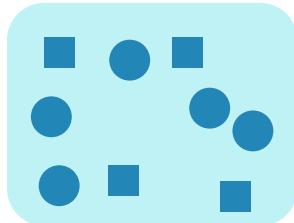
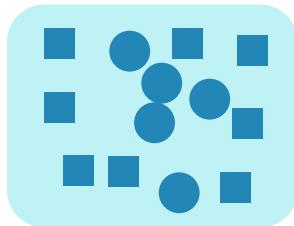
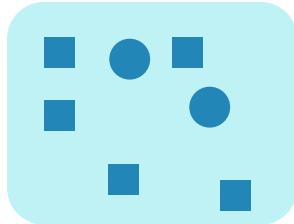


Map Reduce



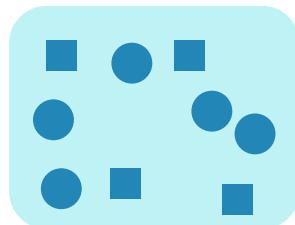
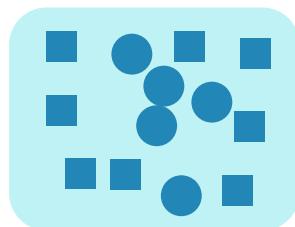
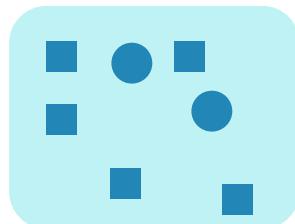


Map Reduce

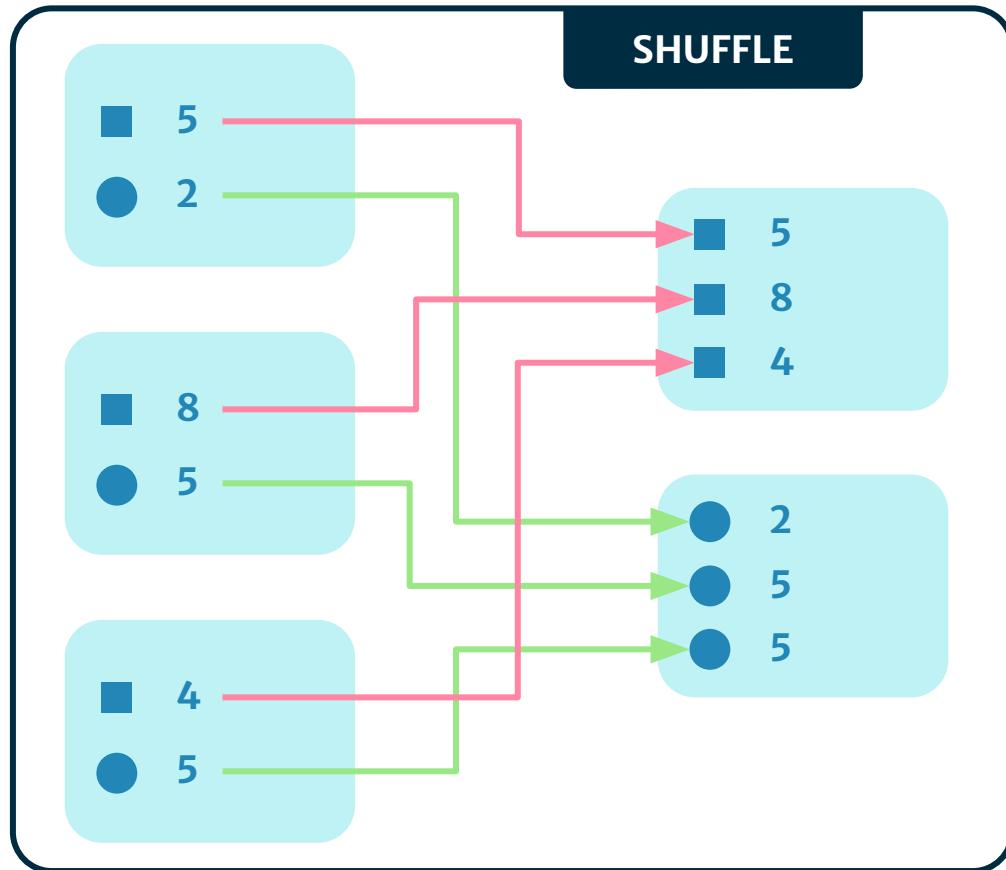




Map Reduce

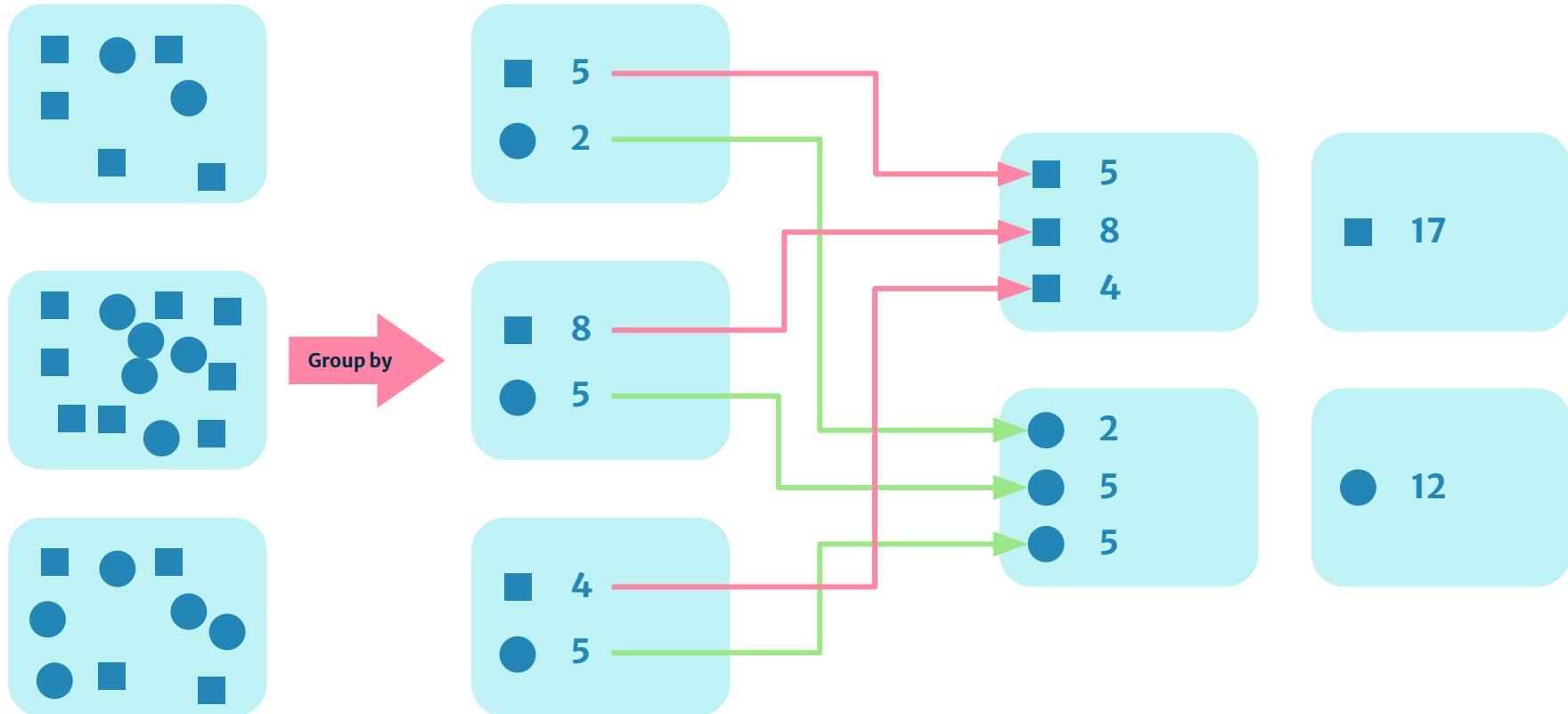


Group by





Map Reduce





L'écosystème Hadoop

HDFS

HDFS = Hadoop Distributed File System



- Comme un système de fichier classique
- Distribué
- Haute disponibilité
- Résilience

Utilisation & Limitation

- Gros fichier (bloc de 128 Mo)
- Pas de modification des fichiers (append accepté)
- Optimisé pour la lecture séquentielle de fichier

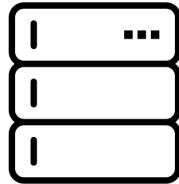




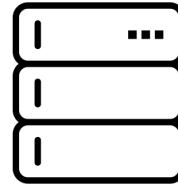
HDFS : Exemple



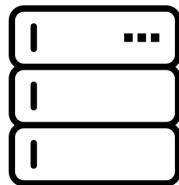
Datanode



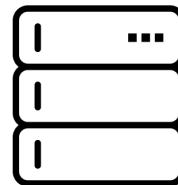
Datanode



Datanode

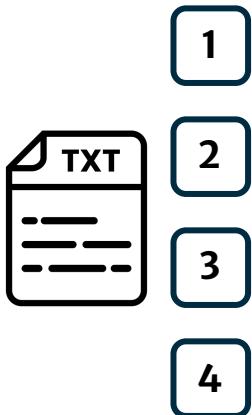


Datanode

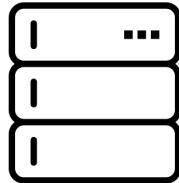




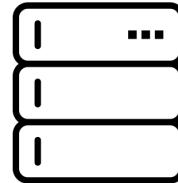
HDFS : Exemple



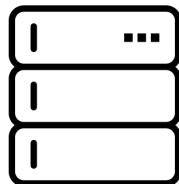
Datanode



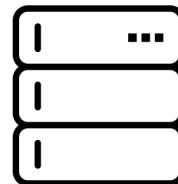
Datanode



Datanode

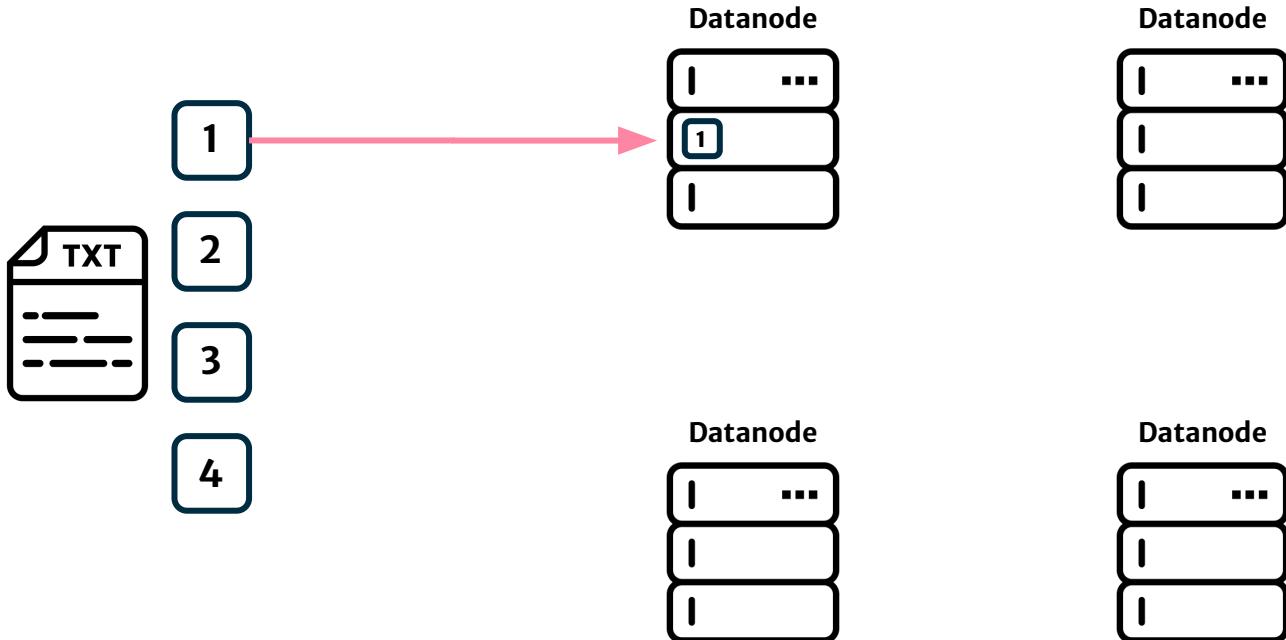


Datanode



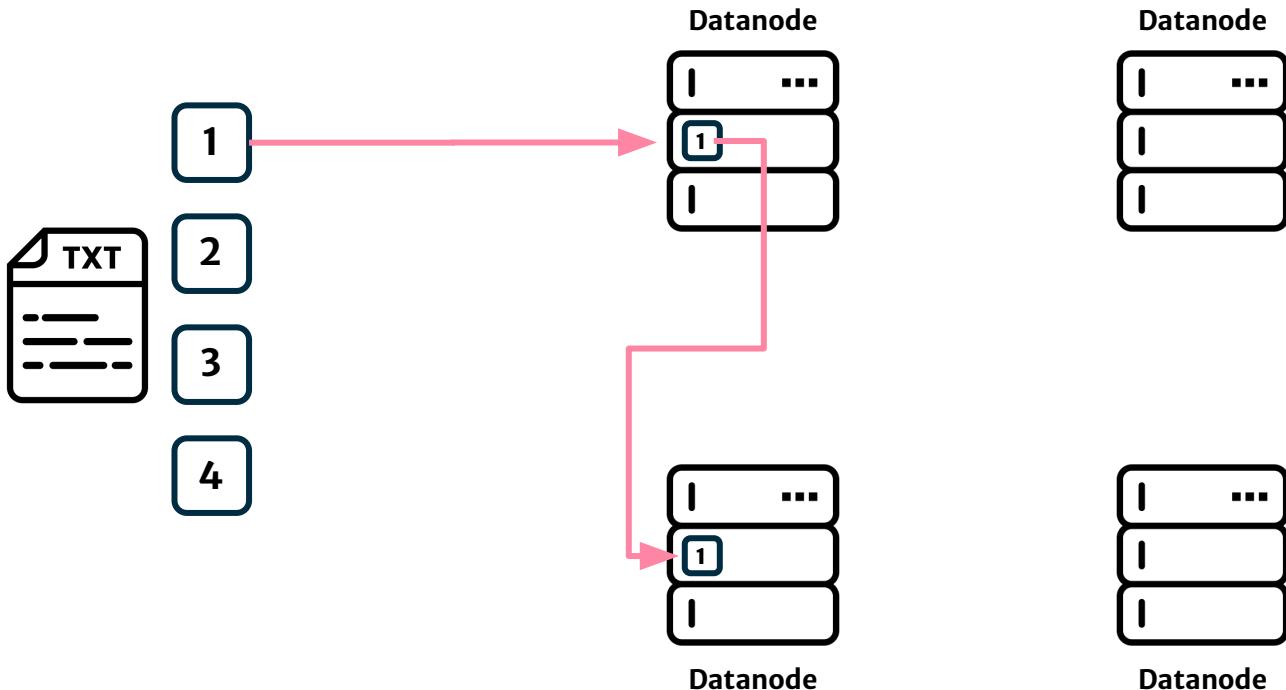


HDFS : Exemple

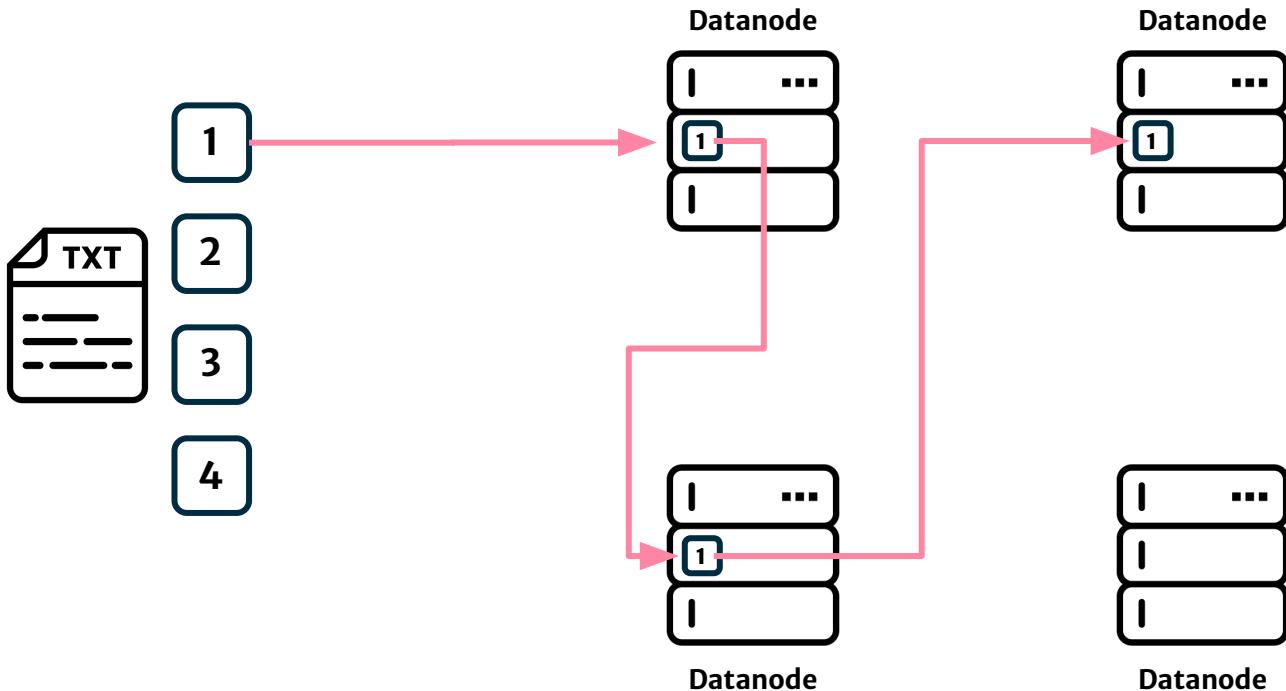




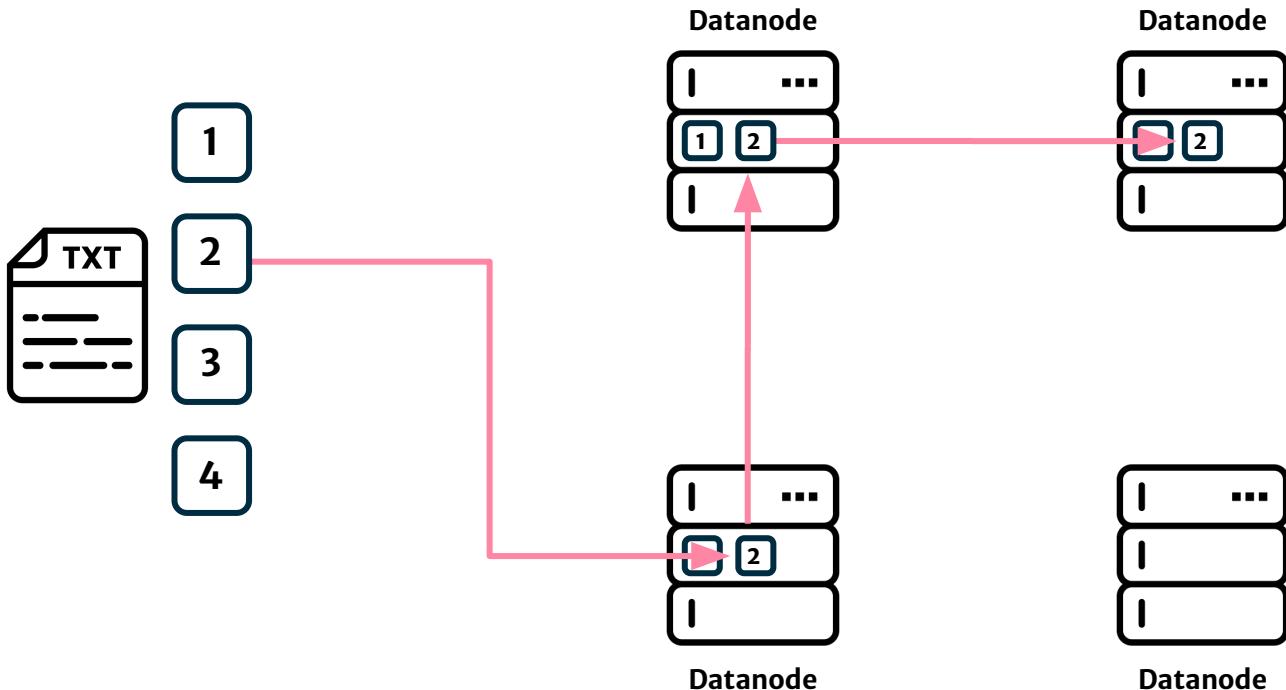
HDFS : Exemple



HDFS : Exemple

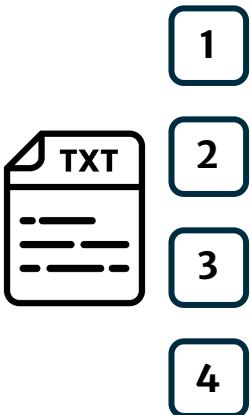


HDFS : Exemple

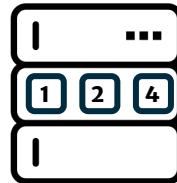




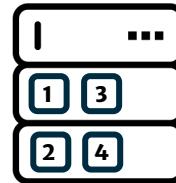
HDFS : Exemple



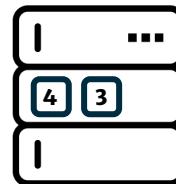
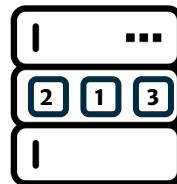
Datanode



Datanode

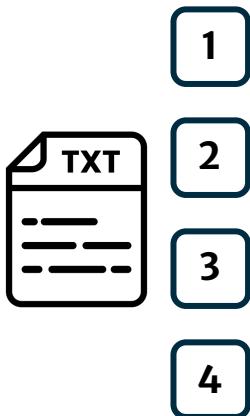


Datanode

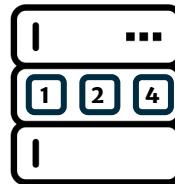


Datanode

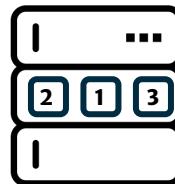
HDFS : Exemple



Datanode

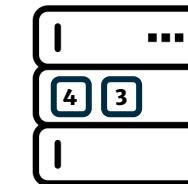


Datanode



Datanode

Datanode



Datanode

Formats de fichiers



- **Texte (csv, json)**
 - Lent
 - Compatible
- **Parquet**
 - Binaire
 - Schéma
 - Compressé
- **Avro**
 - Binaire
 - Schéma
 - Streaming



L'écosystème Hadoop

Hive



- Vue SQL sur les fichiers du HDFS
 - Data Warehouse
 - HiveQL ~ = SQL
-
- Génère du code Spark ou mapreduce
 - Données structurées (Parquet, Avro, ...)
 - Pas besoin de programmer (SQL)





L'écosystème Hadoop

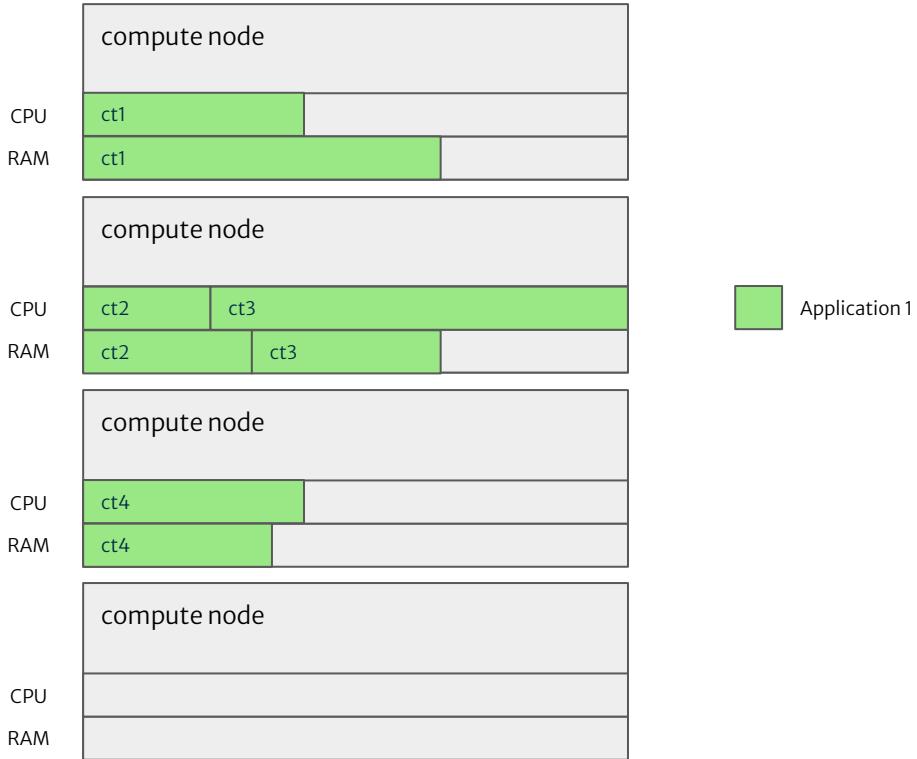
YARN

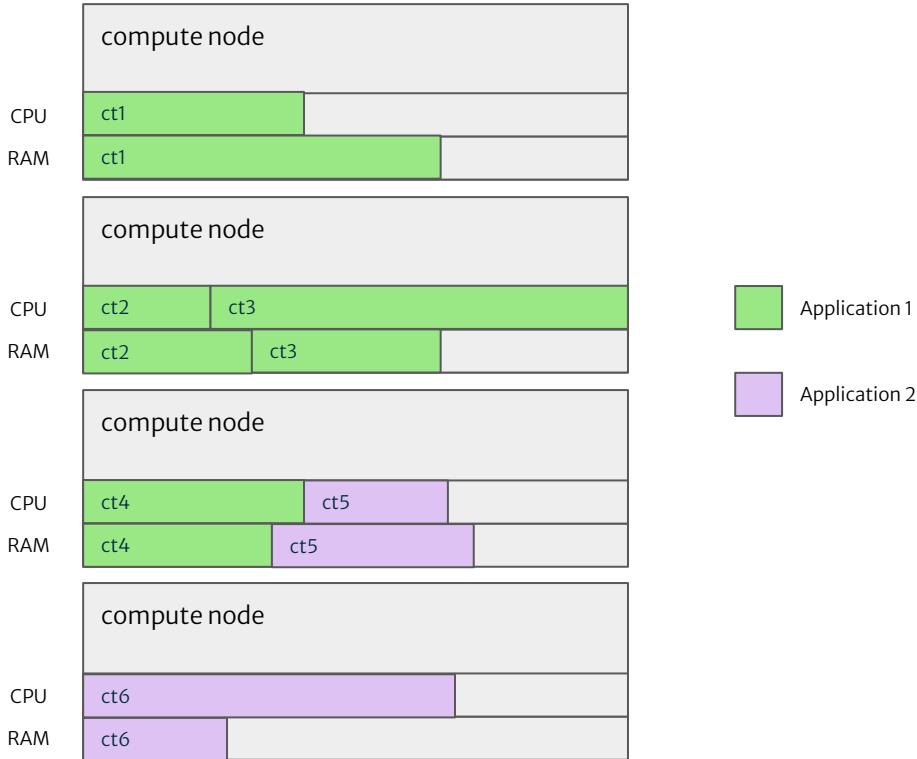
YARN = Yet Another Resource Negotiator



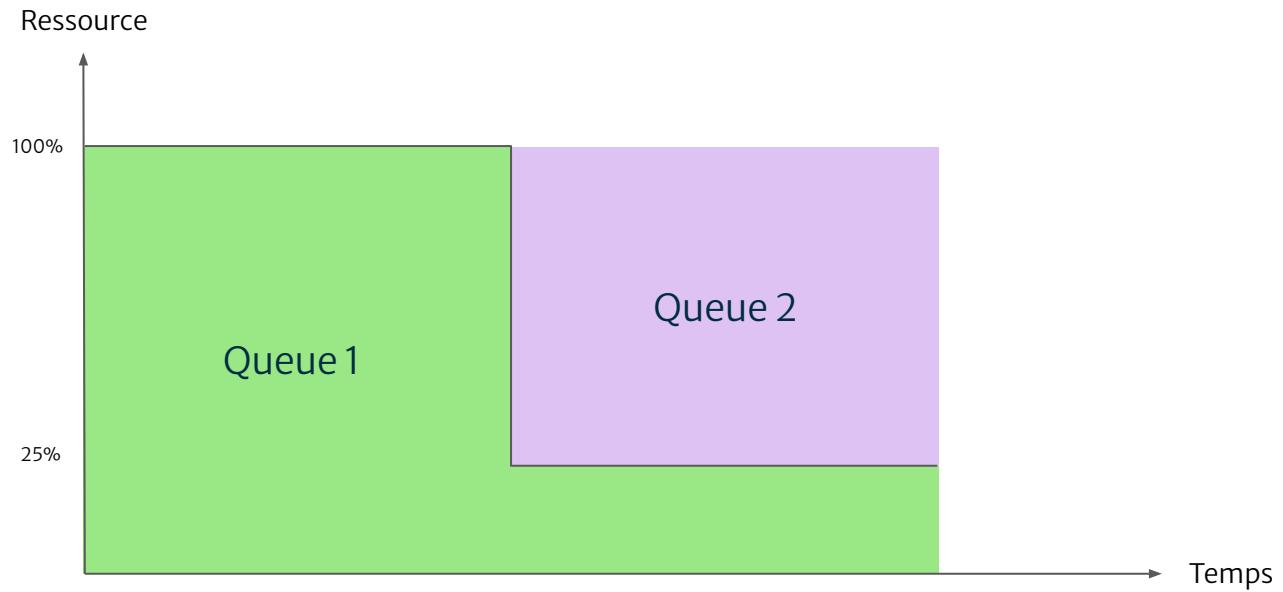
- Gère la répartition des ressources de calcul
- Conteneur = CPU + RAM
- Queue pour la répartition équitable

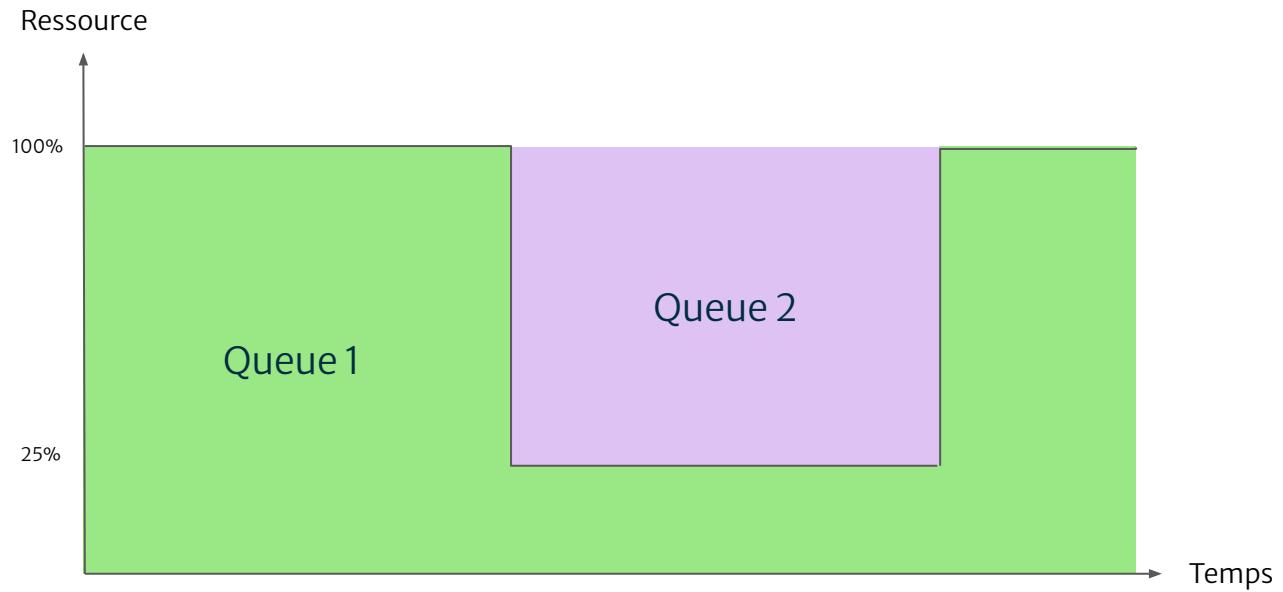














L'écosystème Hadoop

SPARK



- Distribution des calculs (mapreduce)
- Moteur de calcul générique
 - Spark SQL
 - Spark Streaming
 - Spark ML
- Garde les résultats intermédiaires en mémoire
- Gestion des pannes
- Scala, Python
- Gère la colocalisation, données et calcul





1. HDFS stocke les données

compute node + data node
A1 - A2
CPU
RAM

compute node + data node
A3
CPU
RAM

compute node + data node
CPU
RAM

Spark + YARN + HDFS



1. HDFS stocke les données
2. YARN alloue les ressources à spark (où se trouve la donnée)

compute node + data node		
HDFS	A1 - A2	
CPU	ct1	ct2
RAM	ct1	ct2

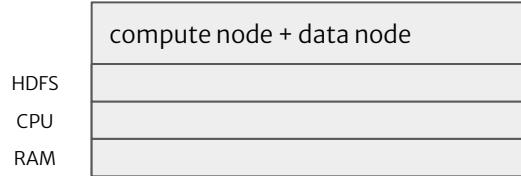
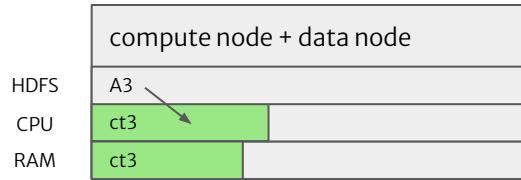
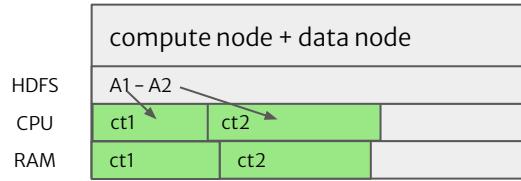
compute node + data node		
HDFS	A3	
CPU	ct3	
RAM	ct3	

compute node + data node		
HDFS		
CPU		
RAM		

Spark + YARN + HDFS



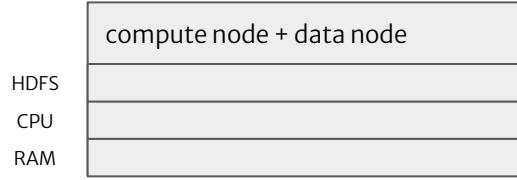
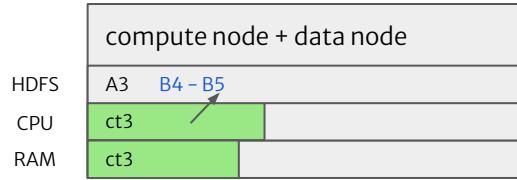
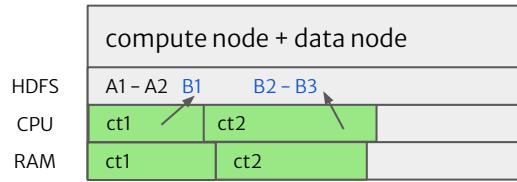
1. HDFS stocke les données
2. YARN alloue les ressources à spark (où se trouve la donnée)
3. Spark effectue des calculs



Spark + YARN + HDFS



1. HDFS stocke les données
2. YARN alloue les ressources à spark (où se trouve la donnée)
3. Spark effectue des calculs
4. Spark stocke les résultats sur HDFS



Spark + YARN + HDFS



1. HDFS stocke les données
2. YARN alloue les ressources à spark (où se trouve la donnée)
3. Spark effectue des calculs
4. Spark stocke les résultats sur HDFS
5. Une fois terminé les ressources sont libérées dans YARN

HDFS	compute node + data node
CPU	A1 - A2 B1 - B2 - B3
RAM	

HDFS	compute node + data node
CPU	A3 B4 - B5
RAM	

HDFS	compute node + data node
CPU	
RAM	



Take away

- Croissance exponentielle des volumes de données
- Scaling verticale insuffisant
- Nécessité de créer une nouvelle programmation distribuée
 - Stockage
 - Traitement
- Hadoop
- MapReduce
- Spark



Les concepts de base de **Spark**





Programme de la section

- Partitioning
- Pipelining
- Lineage
- Parallélisme
- Task, Stage et Job
- Optimisation



Problématiques

- Traiter des grosses quantités de données
- Le plus rapidement possible
- Sans perdre de données
- Gérer l'augmentation de la charge



Le **partitioning** à la lecture

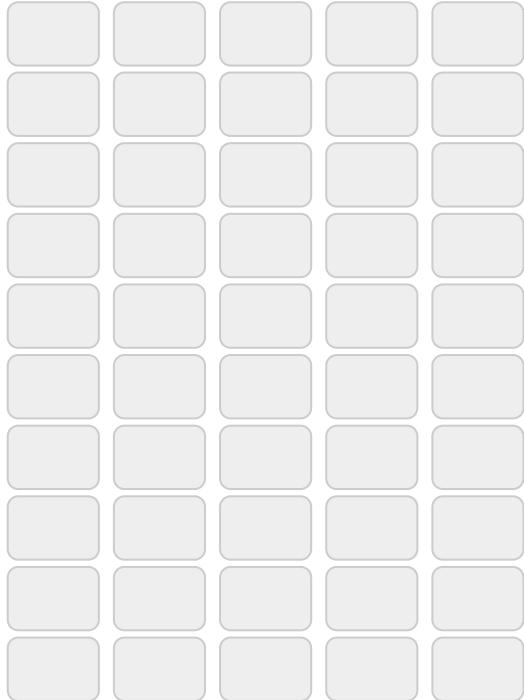


Le partitioning

1 gros fichier



Plein de partitions



Comment Spark crée ses partitions ?



- Dépend du parallélisme
- Dépend de la taille du fichier
 - < 4 mo : 1 partition tous les 4 mo
 - > 128 mo : 1 partition tous les 128 mo
 - Sinon : autant de partition que de parallélisme

```
spark.sql.files.maxPartitionBytes = 128 mo  
  
spark.sql.files.openCostInBytes = 4 mo  
  
spark.sql.files.minPartitionNum = spark.default.parallelism
```

 Lien



Pipelining

Pipelining



```
: Kelly Mr. James,male,34  
: Connolly Miss. Kate,female,30  
: Wirz Mr. Albert,male,27
```

--	--	--

--	--	--	--

--	--	--	--

```
spark.read.csv("data.csv") \  
    .withColumn("title", \  
    split(col("name"), " ")[1]) \  
    .filter(col("title") == "Mr.") \  
    .show()
```

Pipelining



```
: Kelly Mr. James,male,34  
: Connolly Miss. Kate,female,30  
: Wirz Mr. Albert,male,27
```

Kelly Mr. James	male	34
-----------------	------	----

--	--	--	--

--	--	--	--

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ") [1]) \  
.filter(col("title") == "Mr.") \  
.show()
```

Pipelining



```
: Kelly Mr. James,male,34  
: Connolly Miss. Kate,female,30  
: Wirz Mr. Albert,male,27
```

--	--	--

Kelly Mr. James	male	34	Mr.
-----------------	------	----	-----

--	--	--	--

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ")[1]) \  
.filter(col("title") == "Mr.") \  
.show()
```

Pipelining



```
• Kelly Mr. James,male,34  
• Connolly Miss. Kate,female,30  
• Wirz Mr. Albert,male,27
```

--	--	--

--	--	--	--

Kelly Mr. James	male	34	Mr.
-----------------	------	----	-----

```
spark.read.csv("data.csv") \  
    .withColumn("title", \  
    split(col("name"), " ") [1]) \  
    .filter(col("title") == "Mr.") \  
    .show()
```

Pipelining



```
: Kelly Mr. James,male,34  
: Connolly Miss. Kate,female,30  
: Wirz Mr. Albert,male,27
```

Conno Miss. Kate	female	34
------------------	--------	----

--	--	--	--

Kelly Mr. James	male	34	Mr.
-----------------	------	----	-----

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ") [1]) \  
.filter(col("title") == "Mr.") \  
.show()
```

Pipelining



```
: Kelly Mr. James,male,34  
: Connolly Miss. Kate,female,30  
: Wirz Mr. Albert,male,27
```

--	--	--

Conno Miss. Kate	female	34	Miss.
------------------	--------	----	-------

Kelly Mr. James	male	34	Mr.
-----------------	------	----	-----

```
spark.read.csv("data.csv") \  
    .withColumn("title", \  
    split(col("name"), " ") [1]) \  
    .filter(col("title") == "Mr.") \  
    .show()
```

Pipelining



```
• Kelly Mr. James,male,34  
• Connolly Miss. Kate,female,30  
• Wirz Mr. Albert,male,27
```

--	--	--

--	--	--	--

Kelly Mr. James	male	34	Mr.
-----------------	------	----	-----

```
spark.read.csv("data.csv") \  
    .withColumn("title", \  
    split(col("name"), " ") [1]) \  
    .filter(col("title") == "Mr.") \  
    .show()
```

Pipelining



```
: Kelly Mr. James,male,34  
: Connolly Miss. Kate,female,30  
: Wirz Mr. Albert,male,27
```

Wirz Mr. Albert	male	27
-----------------	------	----

--	--	--	--

Kelly Mr. James	male	34	Mr.
-----------------	------	----	-----

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ") [1]) \  
.filter(col("title") == "Mr.") \  
.show()
```

Pipelining



```
• Kelly Mr. James,male,34  
• Connolly Miss. Kate,female,30  
• Wirz Mr. Albert,male,27
```

--	--	--

--	--	--	--

Kelly Mr. James	male	34	Mr.
Wirz Mr. Albert	male	27	Mr.

```
spark.read.csv("data.csv") \  
    .withColumn("title", \  
    split(col("name"), " ") [1]) \  
    .filter(col("title") == "Mr.") \  
    .show()
```



Comment Spark arrive-t-il à “pipeliner” ses traitements ?

Programmation Eager et Lazy



- **Eager** : exécuté dès que l'instruction est atteinte
- **Lazy** : exécuté dès qu'un résultat est référé
- Dans Spark :
 - Le calcul des schémas est eager
 - Le calcul des transformations sur la données est lazy

Spark ne fait aucun traitement (*transformation*) sur la donnée tant qu'on ne lui demande pas un résultat (*action*).



Actions

- collect
- take
- show
- count
- write
- foreach
- ...

Transformations

- map
- select
- filter
- where
- group by
- join
- with column
- ...



Exemple

```
name,gender,age
Kelly Mr. James,male,34
Connolly Miss. Kate,female,30
Wirz Mr. Albert,male,27
```

```
spark.read.csv("data.csv") \
    .withColumn("title", \
    split(col("name"), " ")[1]) \
    .filter(col("title") == "Mr.") \
    .show()
```



Exemple

```
name,gender,age  
Kelly Mr. James,male,34  
Connolly Miss. Kate,female,30  
Wirz Mr. Albert,male,27
```



name	gender	age
-------------	---------------	------------

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ")[1]) \  
.filter(col("title") == "Mr.") \  
.show()
```



Exemple

```
name,gender,age  
Kelly Mr. James,male,34  
Connolly Miss. Kate,female,30  
Wirz Mr. Albert,male,27
```

name	gender	age
------	--------	-----

name	gender	age	title
------	--------	-----	-------

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ")[1]) \  
.filter(col("title") == "Mr.") \  
.show()
```



Exemple

```
name,gender,age  
Kelly Mr. James,male,34  
Connolly Miss. Kate,female,30  
Wirz Mr. Albert,male,27
```

name	gender	age
------	--------	-----

name	gender	age	title
------	--------	-----	-------

name	gender	age	title
------	--------	-----	-------

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ") [1]) \  
.filter(col("title") == "Mr.") \  
.show()
```



Exemple

```
name,gender,age  
Kelly Mr. James,male,34  
Connolly Miss. Kate,female,30  
Wirz Mr. Albert,male,27
```

name	gender	age
------	--------	-----

name	gender	age	title
------	--------	-----	-------

name	gender	age	title
------	--------	-----	-------

```
spark.read.csv("data.csv") \  
.withColumn("title", \  
split(col("name"), " ") [1]) \  
.filter(col("title") == "Mr.") \  
.show()
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
Wirz Mr. Albert	male	27	Mr.



Comment **Spark** se souvient-t-il
des transformations à appliquer ?



Data lineage



Définition

WIKIPÉDIA

Data Lineage en français "lignée des données" est un processus qui vise à fournir une cartographie du [système d'information](#). Il permet une visualisation du cycle de vie de la donnée en vue de répondre aux questions suivantes : de quelle source provient cette donnée, et quelles transformations a-t-elle subies.

Cette thématique prend de l'importance avec l'arrivée du [RGPD](#).



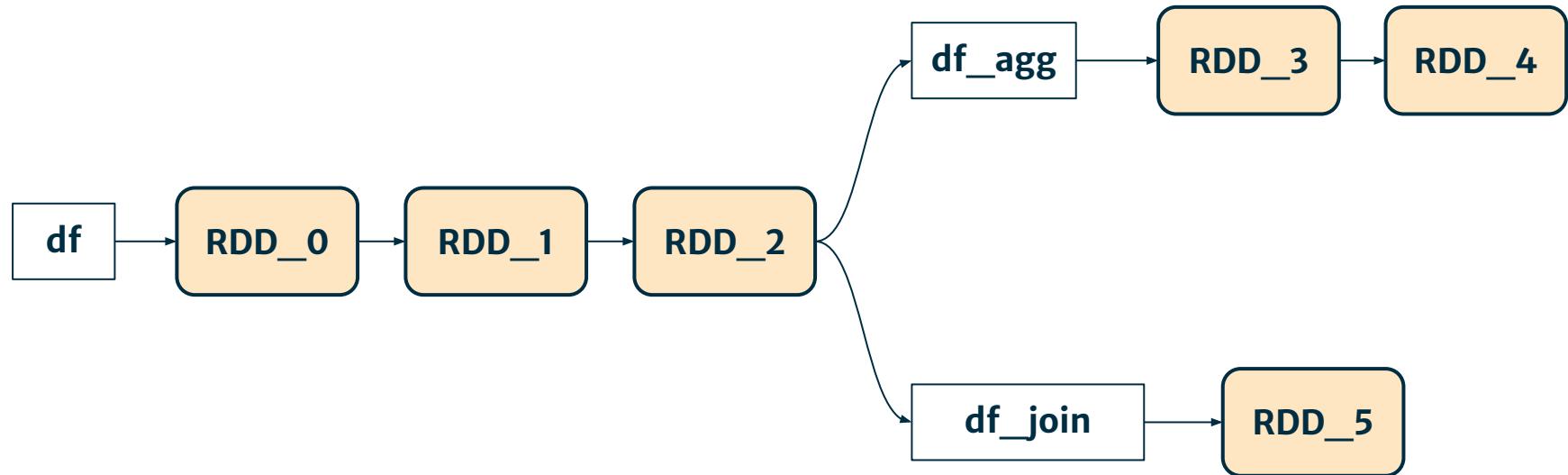
Le lineage dans Spark

```
df = spark.read.csv("data.csv")
    .withColumn("title", split(col("name"), " ")[1])
    .filter(col("title") == "Mr.")

df_agg = df.groupBy(col("title")).count()
df_join = df.join(spark.read.csv("referential"), col("title"))

df.count()
df_agg.count()
df_join.count()
```

Le lineage dans Spark





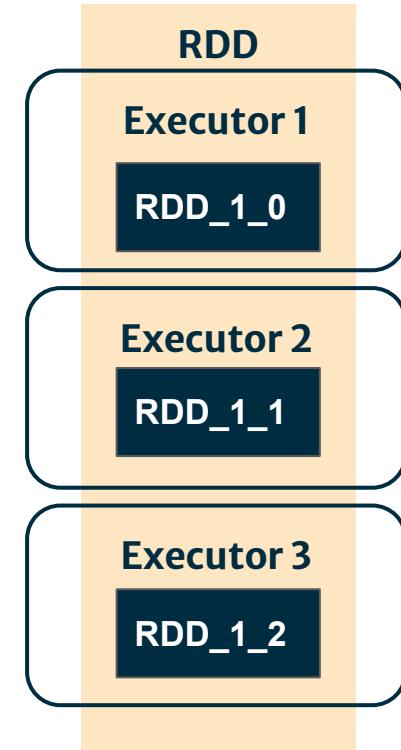
Comment **Spark** traite-t-il ses partitions ?

Le parallélisme



- Les exécuteurs traitent les partitions en parallèle
- Autant de partition en parallèle que de tasks

$$\text{nb_tasks} = \frac{\text{nb_executor} \times \text{nb_cpu_per_executor}}{\text{nb_cpu_per_task}}$$



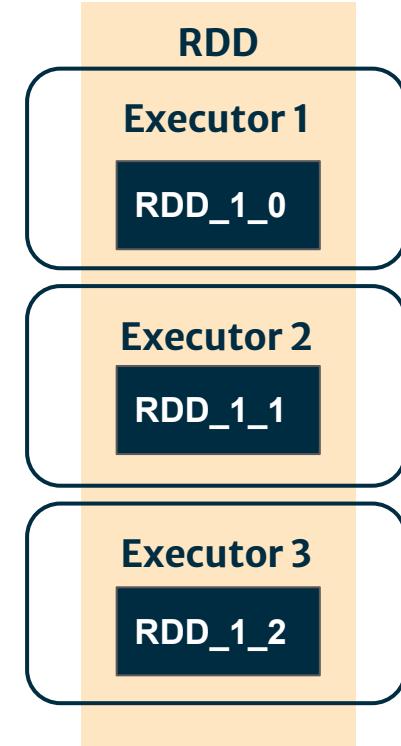
Le parallélisme



- Les exécuteurs traitent les partitions en parallèle
- Autant de partition en parallèle que de tasks

$$\text{nb_tasks} = \frac{\text{nb_executor} \times \text{nb_cpu_per_executor}}{\text{nb_cpu_per_task}}$$

- **3 exécuteurs**
- **2 CPUs par exécuteur**
- **1 CPU par task**
- ?



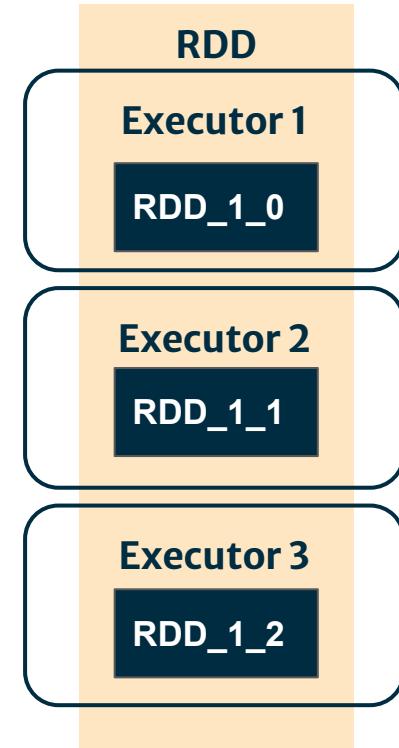
Le parallélisme



- Les exécuteurs traitent les partitions en parallèle
- Autant de partition en parallèle que de tasks

$$\text{nb_tasks} = \frac{\text{nb_executor} \times \text{nb_cpu_per_executor}}{\text{nb_cpu_per_task}}$$

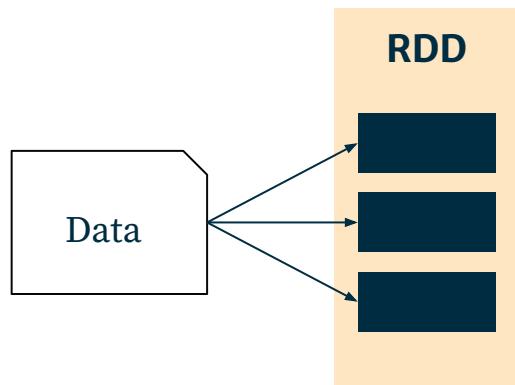
- 3 exécuteurs
- 2 CPUs par exécuteurs
- 1 CPU par task
- **6 tasks**





Task, Stage et Job

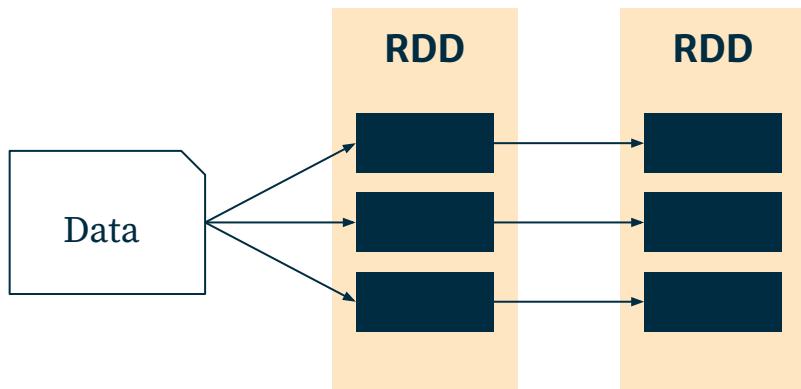
```
spark.read.csv("data.csv") \
```





Task, Stage et Job

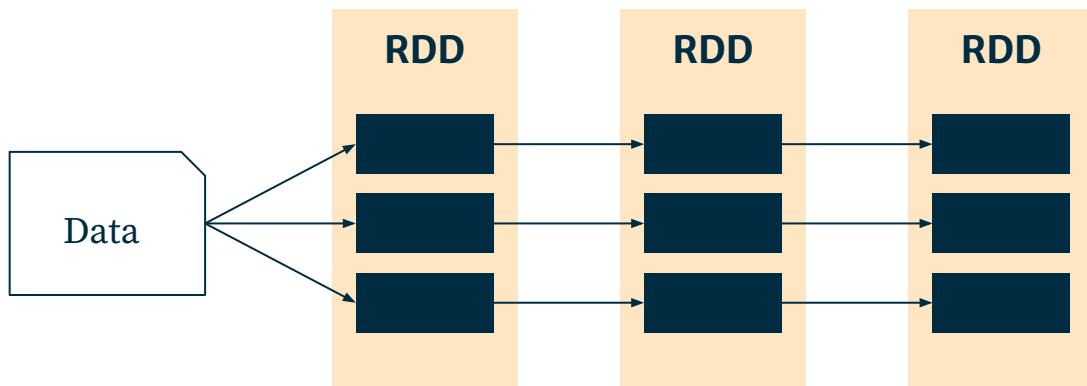
```
spark.read.csv("data.csv") \
    .withColumn("title", \
    split(col("name"), " ") [1]) \
```





Task, Stage et Job

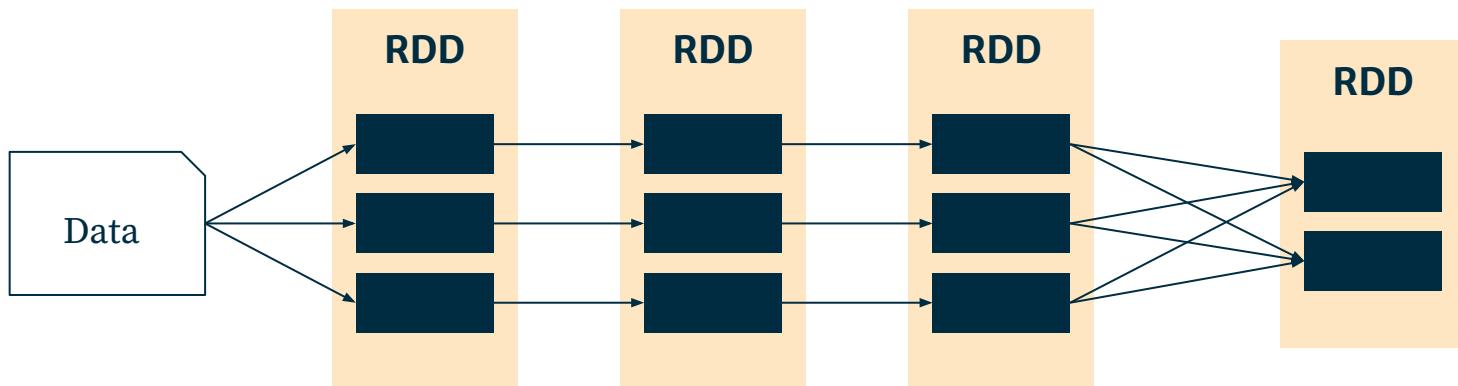
```
spark.read.csv("data.csv") \
    .withColumn("title", \
    split(col("name"), " ")[1]) \
    .filter(col("title") == "Mr.") \
```





Task, Stage et Job

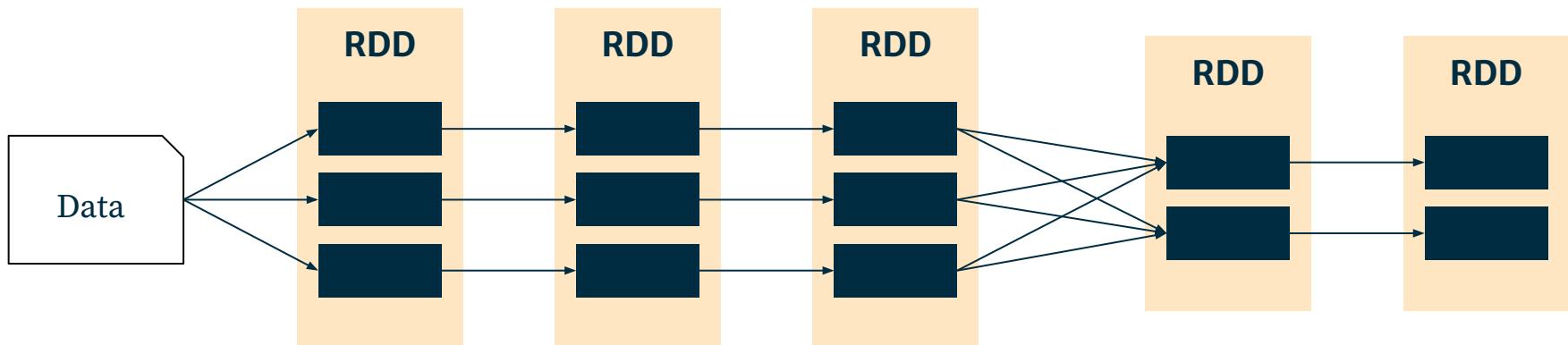
```
spark.read.csv("data.csv") \
    .withColumn("title", \
    split(col("name"), " ")[1]) \
    .filter(col("title") == "Mr.") \
    .groupBy(col("title"))
```





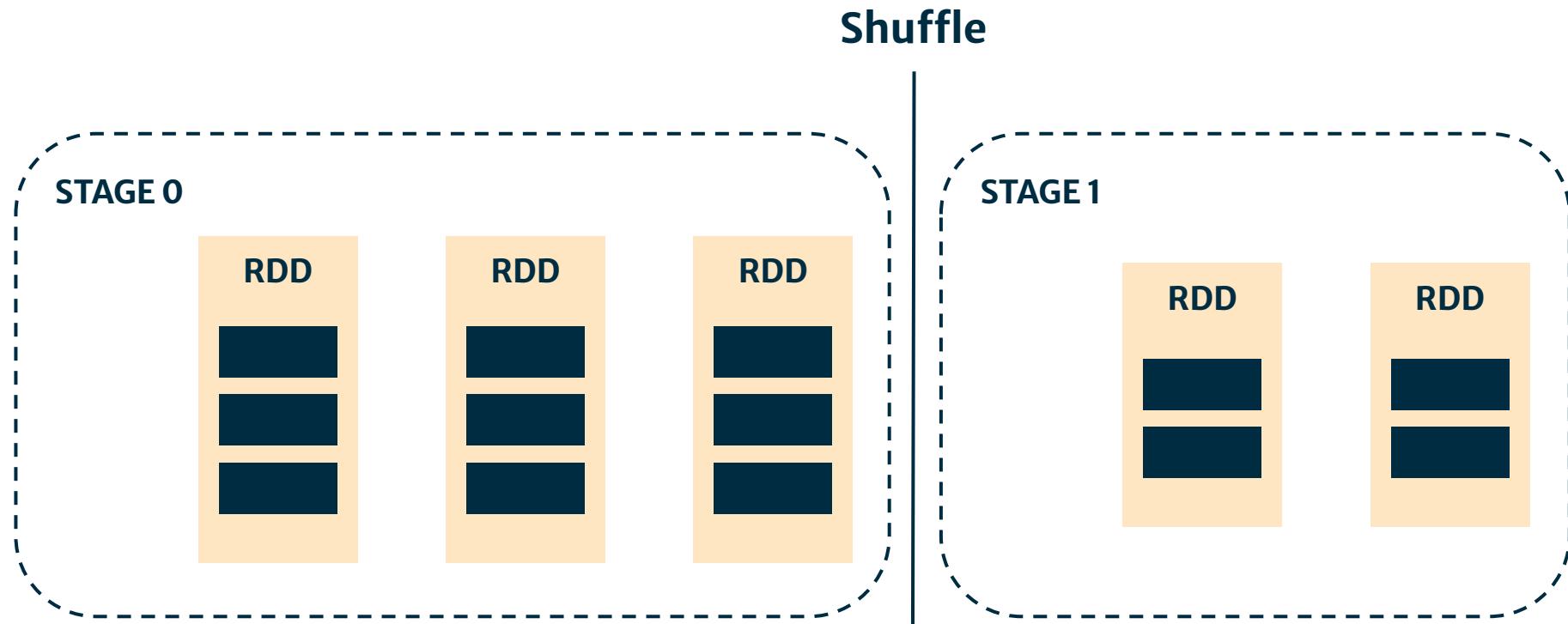
Task, Stage et Job

```
spark.read.csv( "data.csv" ) \
    .withColumn( "title", \
    split(col( "name"), " ") [1]) \
    .filter(col( "title") == "Mr.") \
    .groupByKey( col("title")) \
    .count()
```





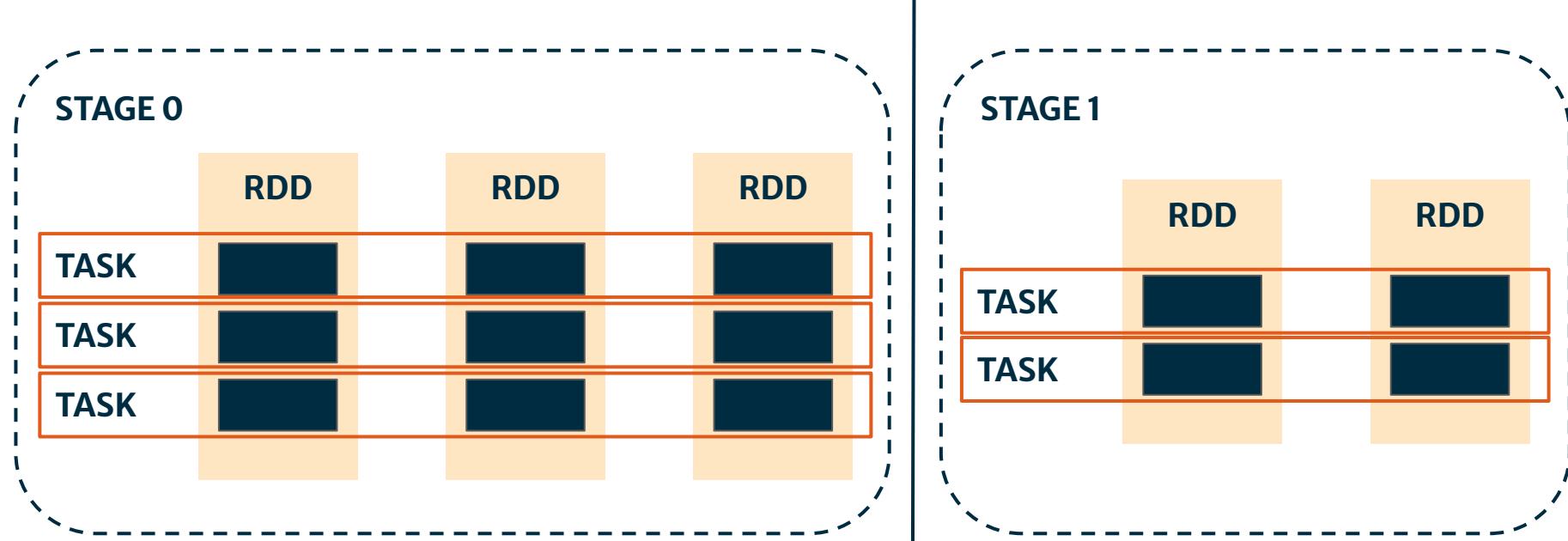
Task, Stage et Job





Task, Stage et Job

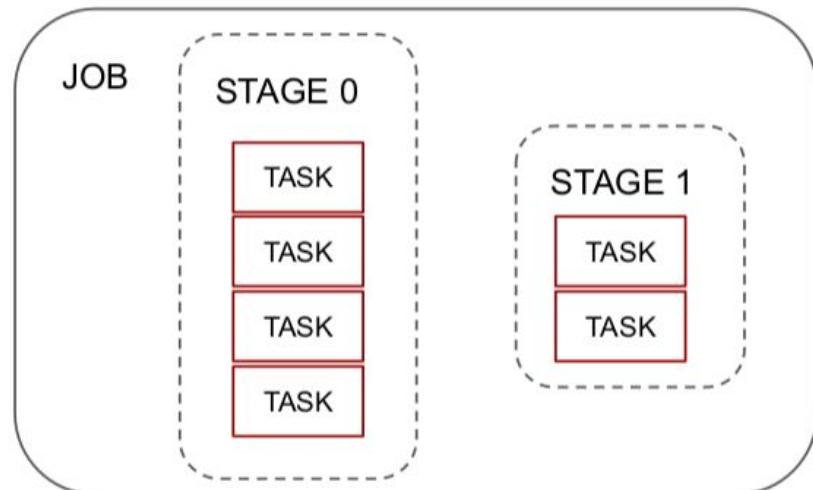
Shuffle



Task, Stage et Job



- **Task** : plus petit travail qu'on puisse donner à un exécuteur
- **Stage** : ensemble de Tasks exécutées en parallèle
- **Job** : ensemble de Stages résultant d'une action



Optimisations



- Une **task** est composée de plusieurs transformations
- Ces transformations sont **lazy**
- Spark optimise la séquence de transformation avant de l'exécuter
- Ce composant s'appelle **Catalyst**

Plus de détails au chapitre sur le fonctionnement interne de Spark



Take away

- **Spark découpe** la donnée en **partitions**
- Les **partitions** sont traitées en **parallèle**
- Les transformations sont **pipelinées**
- Une couche d'optimisation s'ajoute sur les **tasks**
- Tout est rendu possible parce que **Spark est Lazy**



Ma première application Spark



Programme

- Batch, Streaming
- Créer une application Spark
- Comment traiter la donnée
 - RDD
 - Dataset
 - DataFrame
- Le format parquet
- Les systèmes de stockage



Qu'est-ce que Spark ?

- Extract, Transform, Load (ETL) distribué
- Basé sur le paradigme Map Reduce
- Plusieurs API
 - Core
 - SQL
 - Streaming
 - ML
 - Graph





Qu'est-ce que Spark ?

- Extract, Transform, Load (ETL) distribué
- Basé sur le paradigme Map Reduce
- Plusieurs API
 - Core
 - SQL
 - Streaming
 - ML
 - Graph



Batch vs Streaming



Batch

- 1 gros fichier arrive chaque jour
- Je lance un job batch
- Toute la donnée est traitée d'un coup
- Je sauvegarde dans un fichier
- Fin du job

Streaming

- Un job stream tourne
- Des événements arrivent en continu
- Chaque événement est traité unitairement par le job
- Je sauvegarde au fur et à mesure mes résultats
- Le job ne s'arrête jamais



Créer un job Spark

Créer un job Spark



```
from pyspark.sql import SparkSession  
  
def main():  
    spark = SparkSession.builder.getOrCreate()
```

Créer un job Spark



```
from pyspark.sql import SparkSession\n\ndef main():\n    spark = SparkSession.builder \\ \n        .appName("first job") \\ \n        .master("local[*]") \\ \n        .getOrCreate()
```



Le SparkSession

- Nommer l'application
- Définir un master
 - Qui gère les ressources de mon application ?
- GetOrCreate
 - On ne peut créer qu'un seul SparkSession
 - Évite les doublons



Comment traiter la donnée ?



- RDD = Resilient Distributed Dataset
- Spark Core
- Fonctionne comme une liste...
- ...Sauf qu'un map dessus est exécuté en distribué
- Pas de schéma

L'API RDD n'est plus utilisée par les développeurs mais reste le composant de base de Spark.



Dataset

- RDD + un schéma
- Spark SQL
- N'existe qu'en Scala
- Prend un type parameter
 - Dataset[A]
- Est immutable
- Optimise les traitements

DataFrame



- Sous type de Dataset
 - En Scala : Dataset[Row]
 - En Python : seul type de Dataset utilisable
- Ré-implémente tous les mots clés du SQL
- Compatible SQL directement
- Est immutable

RDD vs DataFrame vs Dataset



	RDD	DataFrame	Dataset
Immutability	✓	✓	✓
Schéma	✗	✓	✓
Apache Spark 1	✓	✓	✓ (since 1.6 as experimental, but not in all the languages)
Apache Spark 2	✓	✓ (it does not exist in Java anymore)	✓ (not in the untyped languages such as Python)
Performance optimization	✗	✓	✓
Level	Low	High (built upon RDD)	High (DataFrame extension)
Typed	✓	✗	✓
Syntax Error	Compile time	Compile time	Compile time
Analysis Error	Compile time	Runtime	Compile time



Dataset vs DataFrame

Dataset en Scala :

```
spark.read.csv("data.csv").as[Person]
  .map(person => person.copy(title = person.name.split(" ") (1)))
  .filter(person => person.title == "Mr.")
  .write.csv("output")
```

DataFrame en Python ou Scala :

```
spark.read.csv("data.csv")
  .withColumn("title", split(col("name"), " ") [1]) \
  .where(col("title") == "Mr.") \
  .write.csv("output")
```



Lire de la donnée

- Depuis le SparkSession
- spark.read
 - CSV
 - json
 - parquet
 - textFile
 - jdbc
 - table
- Beaucoup d'options pour configurer
 - `spark.read.option("header", "true").option("delimiter", ";").csv("data.csv")`



Écrire de la donnée

- Depuis un objet Dataset
- df.write
 - CSV
 - json
 - parquet
 - jdbc
 - table
- Possibilité de partitionner par valeur de colonne
 - `df.write.partitionBy("title").csv("output")`
- Plusieurs modes d'écritures
 - Overwrite
 - Append
 - ...

Nom
title=Miss.
title=Mr.
_SUCCESS



Partitionner ses fichiers

- Extrêmement important
- Optimise la lecture d'une source
- À conceptualiser en fonction du besoin
 - Année / mois / jours / heure
 - Région / département / ville
 - ...



Quel **format** de données choisir ?

Fichier Parquet



- Open source
- Stockage colonne
- Sérialisé
- Contient des métadonnées
- Conçu pour la performance
 - Données compressés
 - Stockage binaire
 - Lecture rapide
 - “Predicate push-down”
 - Lire qu'une partie d'un fichier selon une condition
 - Que sur les données partitionnées

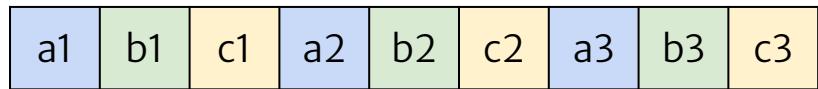


Orienté colonne

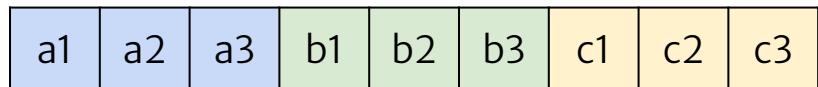
**Logical representation
of data**

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c3

Linear storage



Columnar storage





- Globalement similaire à Parquet
 - Très peu utilisé
- > Utilisez du Parquet (le format par défaut pour Apache Spark)



Où stocker sa donnée ?

Les systèmes de stockages



- HDFS
 - Cluster Hadoop
- Object Storage
 - AWS S3
 - Google Cloud Storage
 - Azure DataLake Storage Gen 2
 - Scaleway Object Storage
- Autre
 - Base SQL
 - DynamoDB
 - ...

À voir selon les besoins...



Choisir son connecteur

Store	Connector	Rename Performance
Amazon S3	s3a	$O(\text{data})$ (COPY+DELETE)
Scaleway Object Storage	s3a	$O(\text{data})$ (COPY+DELETE)
Azure Storage	wasb	$O(\text{files in directory})$
Azure Datalake Gen 2	abfs	$O(1)$
Google GCS	gs	$O(1)$
HDFS	hdfs	$O(1)$



Comment est implémenté l'écriture de fichier

Écriture des données : Comportement par défaut



- Algorithme commit output file v1
- Écriture dans des fichiers temporaires
 - Safe sur un échec de Task
 - Safe sur un échec de Job

Task Commit



Executor

Job

Task 1

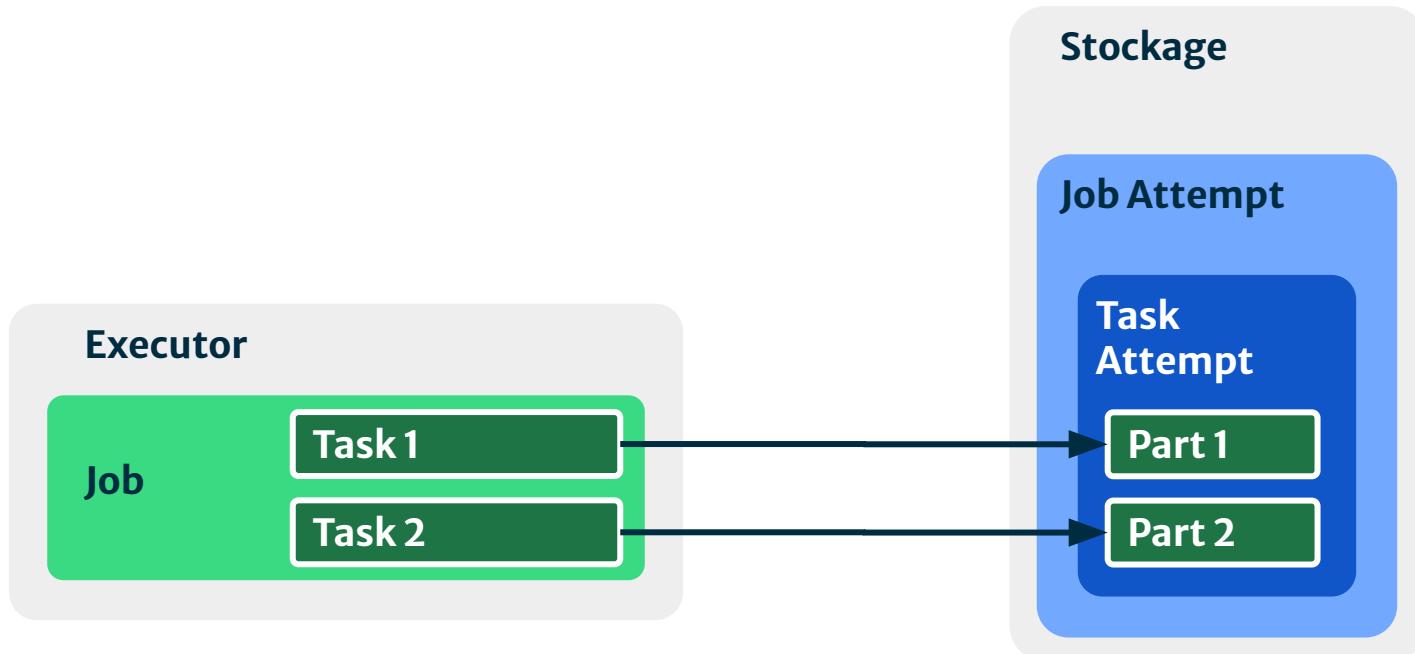
Task 2

Stockage

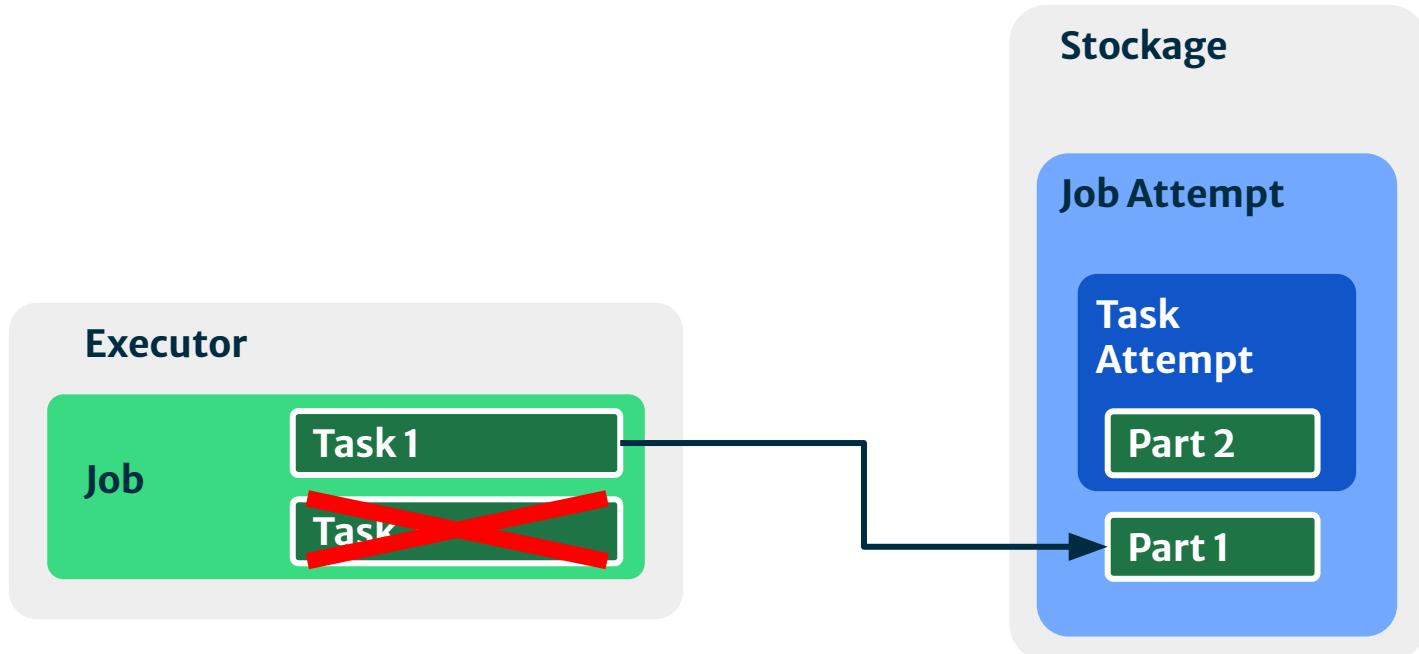
Job Attempt

**Task
Attempt**

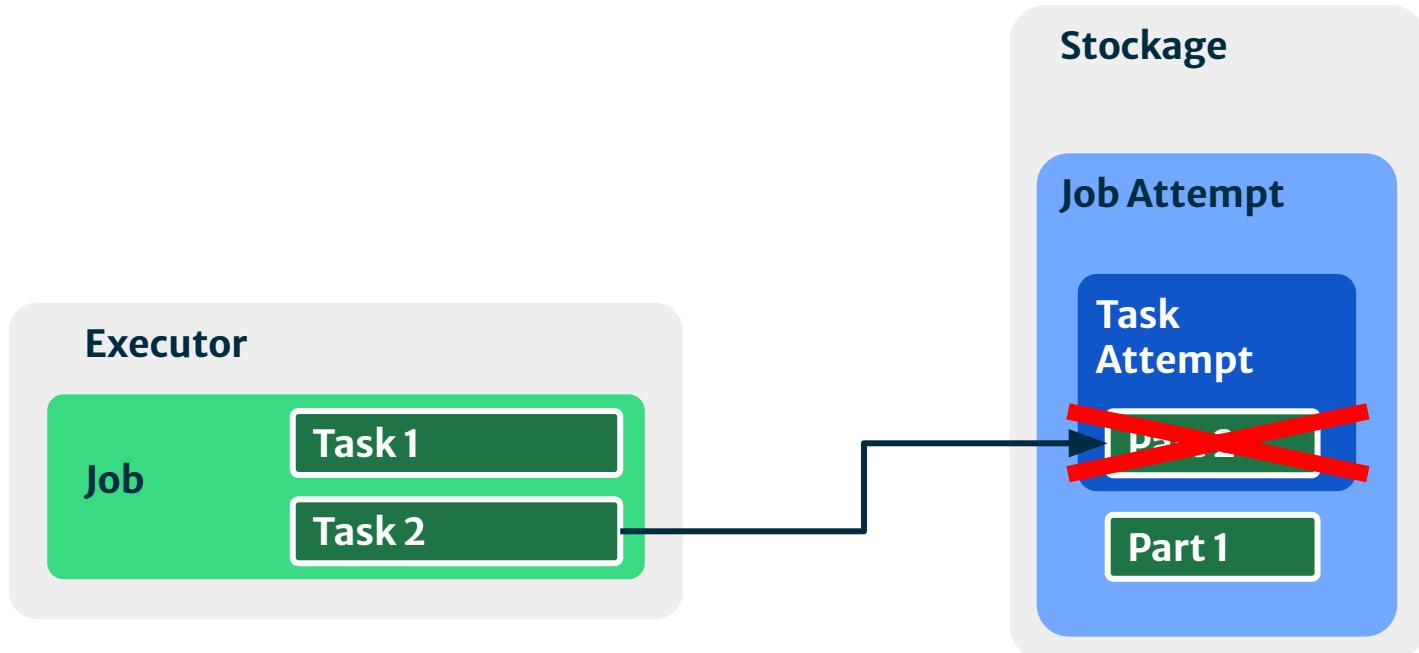
Task Commit



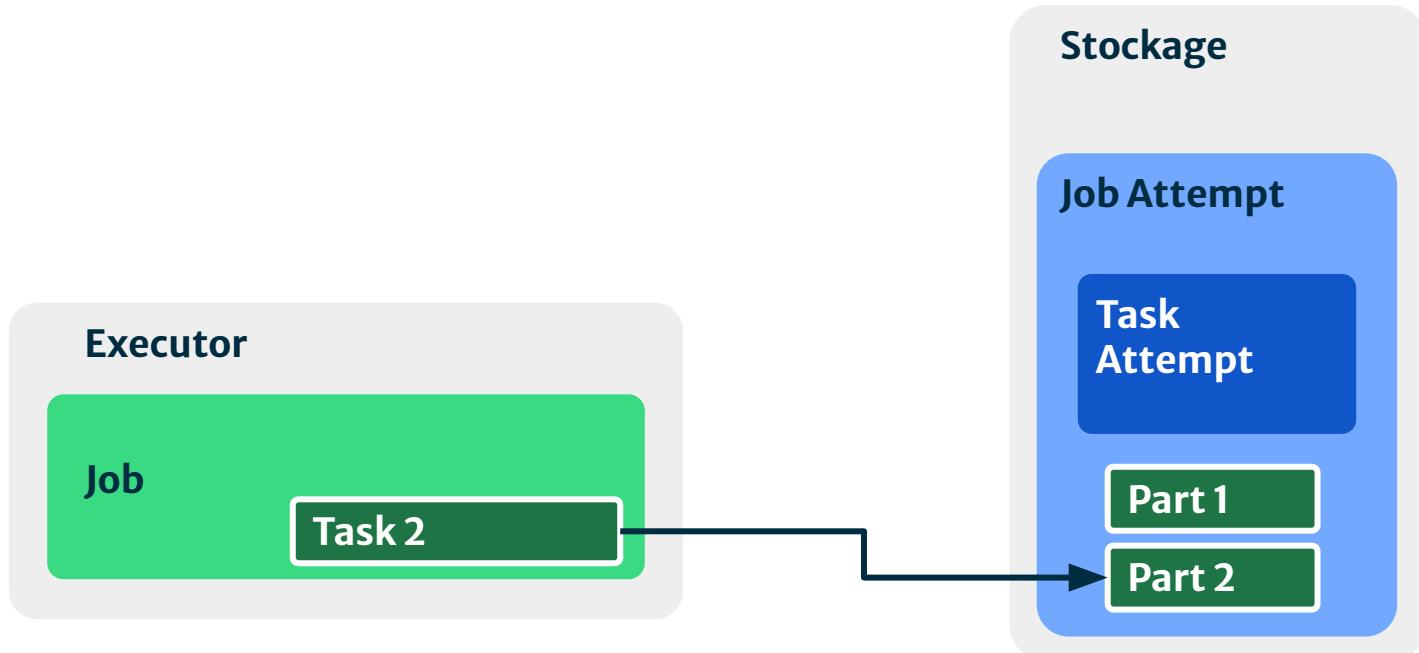
Task Commit



Task Commit



Task Commit



Job Commit



Executor

Job

Stockage

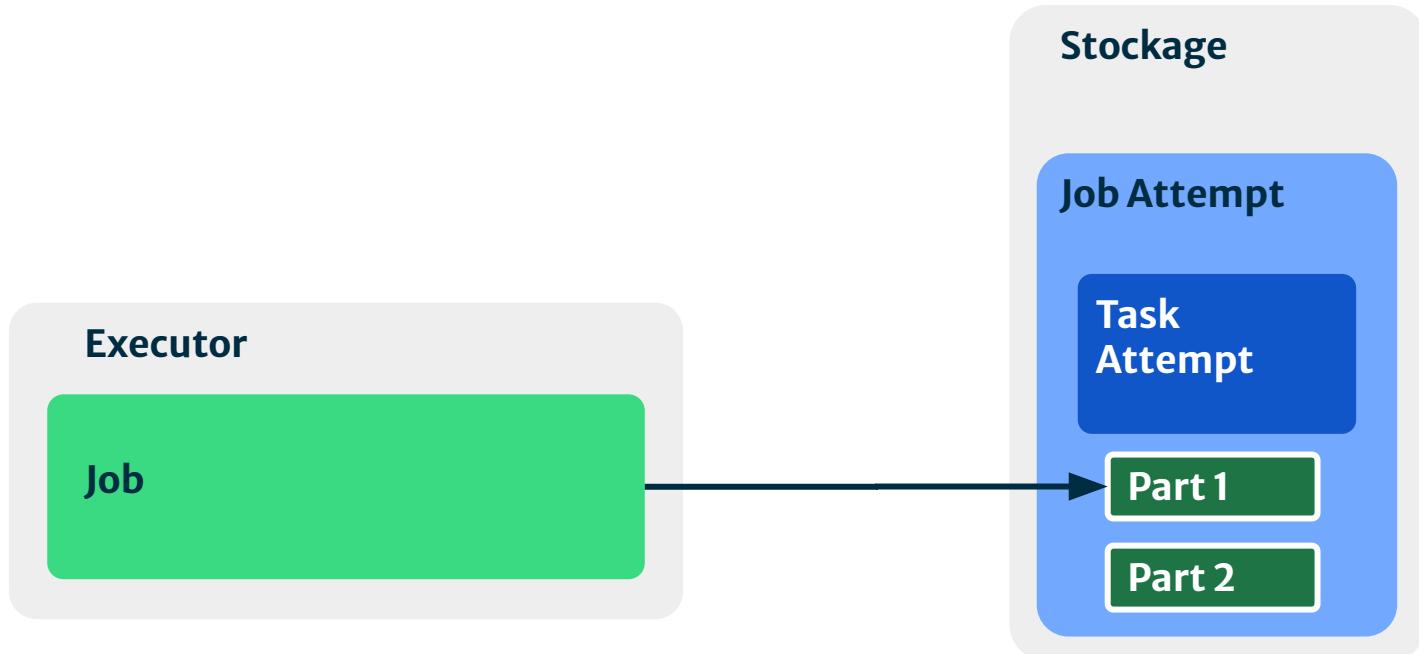
Job Attempt

Task
Attempt

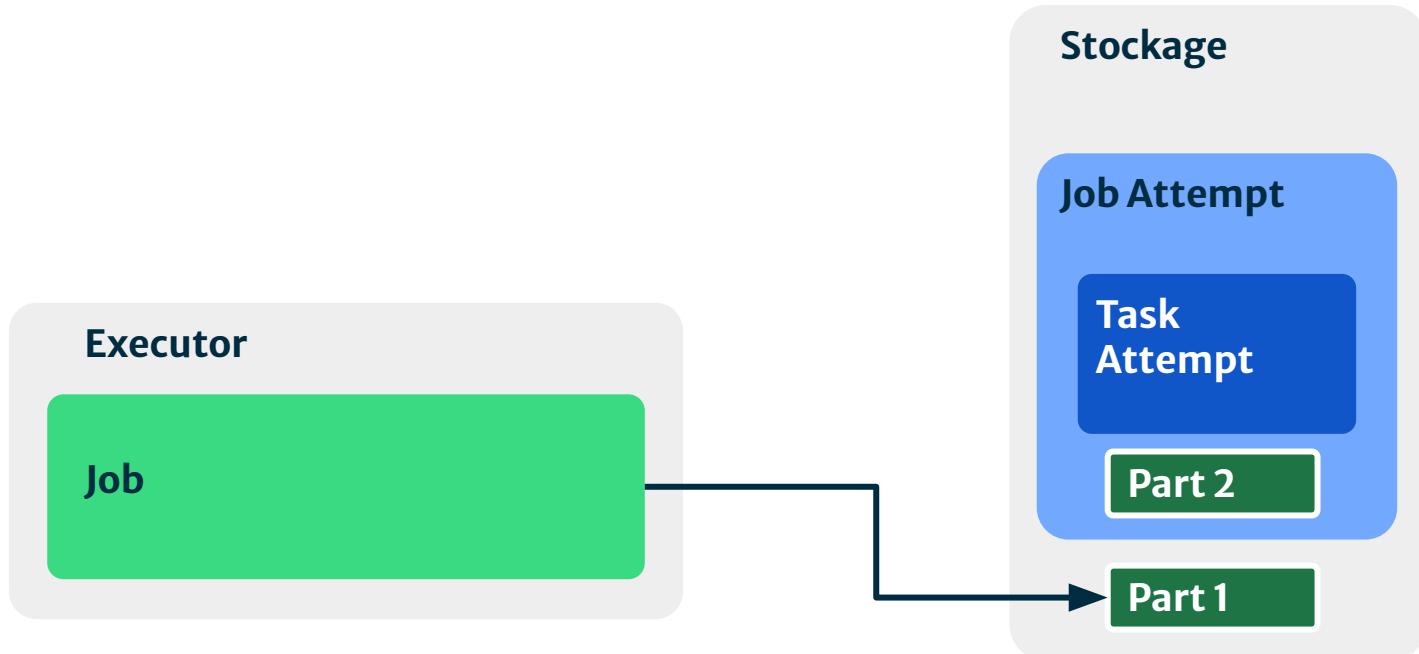
Part 1

Part 2

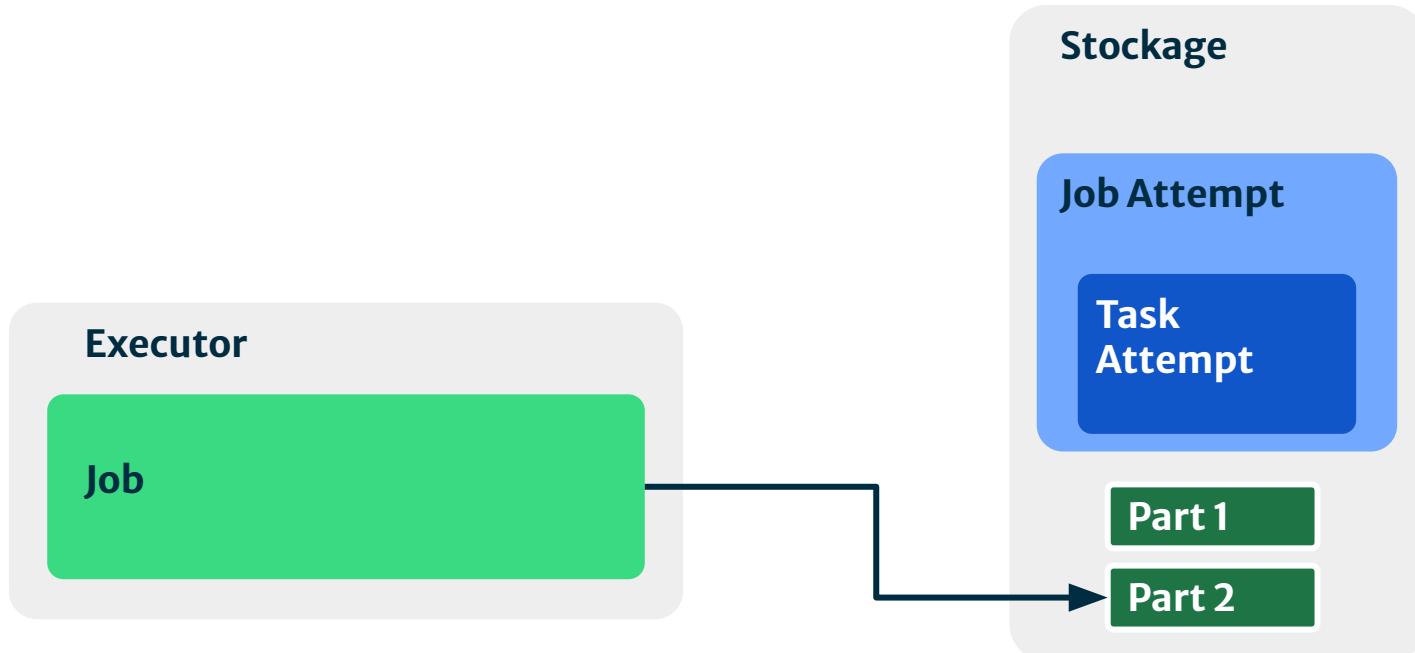
Job Commit



Job Commit



Job Commit



Job Commit



Executor

Stockage

Job Attempt

Task
Attempt

Part 1

Part 2

AWS et Scaleway : attention aux performances



- Spark écrit puis déplace 2 fois la donnée
- AWS et Scaleway sont très peu performants sur ces opérations
- Nécessite de bien configurer son connecteur



Take away

- Privilégier Spark pour le traitement batch
- 1 seule SparkSession par application Spark
- Dataset ou DataFrame,
on oublie les RDD
- Écrire en Parquet
- Les stockages objets fonctionnent bien avec
Spark
- Attention à AWS et Scaleway



Transformer sa donnée

Les bases

Programme



- Opération Map
- Opération Reduce
- Les fonctions sur colonne
- L'objet Column

Les opérations Spark – Rappel



Actions

- collect
- take
- show
- count
- write
- foreach
- ...

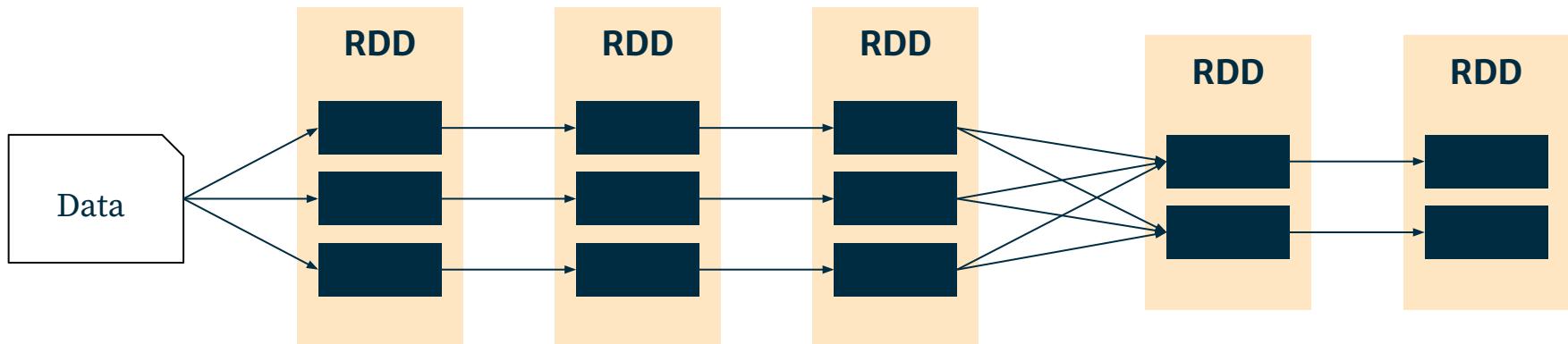
Transformations

- map
- select
- filter
- where
- group by
- join
- with column
- ...



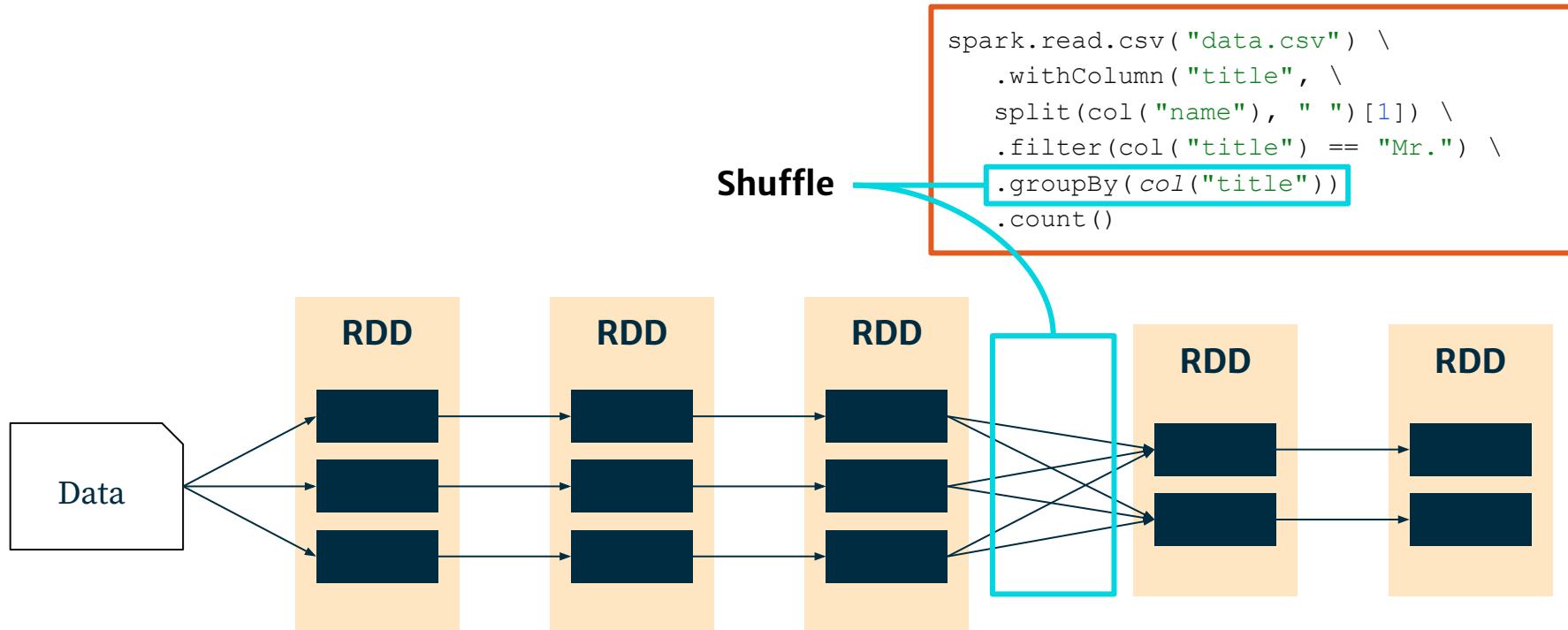
Task, Stage et Job – Rappel

```
spark.read.csv( "data.csv" ) \
    .withColumn( "title", \
    split(col( "name"), " ") [1]) \
    .filter(col( "title") == "Mr.") \
    .groupByKey( col("title")) \
    .count()
```





Task, Stage et Job – Rappel



Map, Reduce



Map

- withColumn
- withColumnRenamed
- select
- filter / where
- drop

Reduce

- groupBy
- join
- orderBy / sort



Les fonctions sur colonne

- Prend une colonne
- Retourne une colonne
- Crée une nouvelle colonne à partir d'une autre



Map

Map - withColumn



- Ajoute une colonne
- Très utilisé
- Très pratique

```
df.withColumn("title", split(col("name"), " ")  
[1])
```

name	gender	age
Kelly Mr. James	male	34



name	gender	age	title
Kelly Mr. James	male	34	Mr.

Map - withColumns



- Depuis Spark 3.3.0
- Prend en paramètre une Map
 - Map[String, Column]
- Applique plusieurs withColumn à la fois

Map - withColumnRenamed



- Renomme une colonne
- Peu utilisé
- Il existe plus pratique
 - Méthode de colonne

```
df.withColumnRenamed("title", "titre")
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.



name	gender	age	titre
Kelly Mr. James	male	34	Mr.

Map - select



- Sélectionne des colonnes
- Très utilisé
- Fonctionne comme en SQL

```
df.select("name", "title")
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.



name	title
Kelly Mr. James	Mr.

Map - select



```
df.select(col("name"), col("gender"), col("age"), split(col("name"), "  
") [1])
```

- Sélectionne des colonnes
- Très utilisé
- Fonctionne comme en SQL
- Peut ajouter des colonnes
- Il faut renommer la colonne

name	gender	age
Kelly Mr. James	male	34



name	gender	age	split(name, , -1)[1]
Kelly Mr. James	male	34	Mr.

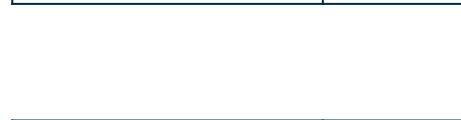
Map - filter / where



- Supprime des lignes
- Très utilisé
- Fonctionne comme en SQL

```
df.where(col("title") ==  
"Mr.")
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
Nolly Miss. Kate	Female	30	Miss.



name	gender	age	title
Kelly Mr. James	male	34	Mr.

Map - drop



- Supprime des colonnes
- Pratique dans certains cas
- Tout garder sauf 1 ou 2 colonnes
- Fréquent après un join

```
df.drop("title")
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.



name	gender	age
Kelly Mr. James	male	34



Reduce

Reduce - groupBy



- Regroupe selon une clé
- Très utilisé
- Fonctionne comme en SQL

```
df.groupBy( col("title")
)
.count()
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
...

title	count
Mr.	1938
Miss.	4329

Reduce - join



- Fusionne selon une clé
- Très utilisé
- Fonctionne comme en SQL
 - inner (défaut)
 - left
 - outer
 - right
 - ...

```
df.join(spark.read.csv("referential"), "title")
```



Reduce - orderBy / sort

- Trie les données
- Fonctionne comme en SQL

```
df.sort("title")
```



Les fonctions sur colonne

Les fonctions sur colonne



- Scala : org.apache.spark.sql.functions._
- Python : pyspark.sql.functions
- Permet de créer une nouvelle colonne

```
def split(str: Column, pattern: String): Column
```

Splits str around matches of the given pattern.

str a string expression to split

pattern a string representing a regular expression. The regex string should be a Java regular expression.

Since 1.5.0

Les différents types de fonctions



- **Fonction d'agrégation**
- **Fonction de collection**
- **Fonction de date / time**
- Fonction mathématiques
- Fonction divers
- **Fonction de non agrégation**
- Fonction de transformation en partition
- Fonction de tri
- **Fonction de chaîne de caractère**
- Fonction UDF
- Fonction Window



Fonction de **non** agrégation



Fonction de non agrégation

- col / column
- lit
- when & otherwise

Fonction de non agrégation - col / column



- La plus simple
- La plus utilisée
- Récupère une colonne par son nom

```
df.select(col("title"))
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
Nolly Miss. Kate	Female	30	Miss.

title
Mr.
Miss.

Fonction de non agrégation - lit



- Crée une colonne
- Avec une valeur fixe

```
df.withColumn("lit",  
lit(10))
```

name	gender	age
Kelly Mr. James	male	34
Nolly Miss. Kate	female	30



name	gender	age	lit
Kelly Mr. James	male	34	10
Nolly Miss. Kate	female	30	10

Fonction de non agrégation - when & otherwise



```
df.withColumn("flag", when(col("gender") == "male",  
1).otherwise(0))
```

- Crée une colonne
- Avec une condition sur le résultat
- Peut être chaîné
 - when().when().when()...
- Si pas de otherwise, return null

name	gender	age
Kelly Mr. James	male	34
Nolly Miss. Kate	Female	30



name	gender	age	flag
Kelly Mr. James	male	34	1
Nolly Miss. Kate	Female	30	0



Fonction de chaîne de caractère

Fonction de chaîne de caractère



- Les plus utilisées
 - length
 - lower / upper
 - split
 - substring
- S'applique uniquement sur des colonnes de type String
- Crée une nouvelle colonne
 - Pas forcément de type String (Split return un Array)



Fonction de date / time



Fonction de date / time

- Comme les fonctions de String mais sur des Date / Time
- Quelques exemples
 - current_timestamp / current_date
 - to_date / to_timestamp
 - dayofweek / dayofmonth / dayofyear
- Astuce : diviser la date en 3 colonnes
 - Pour partitionner year / month / day



Fonction de collection



Fonction de collection

- Les collections
 - array
 - array d'array
 - map
 - json
- Exemples :
 - array_contains
 - array_sort
 - explode

Fonction de collection - array_contains



```
df.withColumn("contains_titi", array_contains(col("list"),  
"titi"))
```

- Crée une colonne
- Retourne true, false ou null
- Peut être utilisé dans un when
 - when(array_contains(), 1)
 - null équivaut à false

list
["tata", "titi", "toto"]
null



list	contains_titi
["tata", "titi", "toto"]	true
null	null

Fonction de collection - array_sort



- Crée une colonne
- Retourne une nouvelle liste triée

```
df.withColumn("sorted",  
array_sort(col("list")))
```



Fonction de collection - explode



- Duplique une ligne
- Autant de fois qu'il y a d'élément dans la liste
- Souvent on drop la colonne list

```
df.withColumn("name", explode(col("list")))
```

list
["titi", "tata", "toto"]



list	name
["titi", "tata", "toto"]	"tata"
["titi", "tata", "toto"]	"titi"
["titi", "tata", "toto"]	"toto"



Fonction d'agrégation



Fonction d'agrégation

- S'utilise après un groupBy
- Calcule une valeur à partir d'un groupe
 - avg
 - count
 - collect_set
 - sum
 - ...

Fonction d'agrégation - count



- Applique une seule agrégation
- Sans distinction de colonne

```
df.groupBy( col("title" ) ).count()  
)
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
...



title	count
Mr.	1938
Miss.	4329



Fonction d'agrégation - avg, collect_set

```
df.groupBy(col("title")).agg(avg(col("age")),  
collect_set(col("gender")))
```

- Applique plusieurs agrégations
- À des colonnes précises
- Attention aux noms par défaut

name	gender	age	title
Franck Mr.	male	30	Mr.
Damien Mr.	male	40	Mr.
...

title	avg(age)	collect_set(gender)
Mr.	35	[male]
Miss.	32,55	[female]



L'objet **Column**



L'objet Column

- C'est un morceau de schéma
- Possède sa “recette”
- Ne contient aucune donnée
- Peut être appliqué à n'importe quelle dataframe
- Rencontre une erreur s'il fait référence à une colonne inexistante



Exemple

```
my_col = when(col("title") == "Mr.", 1).otherwise(lit(0))
df_with_title.withColumn("flag", my_col)
df_without_title.withColumn("flag", my_col)
```

- `my_col` créé sans faire référence à une Dataframe
- Dans `df_with_title` tout se passe bien
- Dans `df_without_title` ça échoue

```
org.apache.spark.sql.AnalysisException: cannot resolve 'title' given input columns: [age, gender, name];;
```

Les méthodes de Column



- Quelques doublons avec les fonctions sur colonnes
 - substr et substring
 - when qui permet d'en enchaîner plusieurs
- De quoi vérifier si les valeurs dans la colonne sont nulles
 - isNull
 - isNotNull
- Des fonctions très pratiques de renommage
 - as
 - alias
 - name

Les méthodes de Column – `isNull` / `isNotNull`



```
df.withColumn("flag", when(col("title").isNull(),  
1).otherwise(0))
```

Les méthodes de Column - as / alias / name



```
df.groupBy(col("title")).agg(avg(col("age")),
collect_set(col("gender")))
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
...



title	avg(age)	collect_set(gender)
Mr.	38,24	[male]
Miss.	32,55	[female]

Les méthodes de Column - as / alias / name



```
df.groupBy(col("title")).agg(avg(col("age")).alias("age_avg"))  
,  
collect_set(col("gender")).alias("genders"))
```

name	gender	age	title
Kelly Mr. James	male	34	Mr.
...



title	age_avg	genders
Mr.	38,24	[male]
Miss.	32,55	[female]



Take away

- Les opérations Map / Reduce sont copiés du SQL
- Plusieurs fonctions peuvent faire la même chose
 - when / filter
 - Performances identiques
- Il existe des fonctions sur les colonnes pour les modifier
- L'objet colonne contient quelques fonctions pratiques
- L'objet colonne ne porte pas la donnée
 - Que le schéma



À vous de jouer !



Tester
son application **Spark**



Programme

- Créer de la donnée
- Tests unitaires
- Tests d'intégrations
- Optimiser son SparkSession pour les tests



Créer de la donnée - en Scala

Créer son jeu de données dans le code



- La donnée à côté du test
 - Plus lisible
 - Plus maintenable
- Plus long à écrire
 - Plus optimisé
 - Plus petit
- Chaque test doit avoir son propre jeu de données

La méthode `createDataFrame`



```
val spark: SparkSession = ...  
  
val rdd: RDD[Row] = ...  
  
val schema: StructType = ...  
  
val dataFrame: DataFrame = spark.createDataFrame(rdd, schema)
```



La méthode `createDataFrame`

```
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{DataFrame, Row}
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}

val rdd: RDD[Row] = spark.sparkContext.parallelize(
    List(
        Row("Turing", 21),
        Row("Hopper", 42)
    )
)
```

La méthode `createDataFrame`



```
val rdd: RDD[Row] = ...
```

```
val schema: StructType = StructType(  
    Array(  
        StructField("name", StringType),  
        StructField("age", IntegerType)  
    )  
)
```

```
val dataFrame: DataFrame = spark.createDataFrame(rdd, schema)
```

+-----+ 	name	age	
+-----+	Turing	21	
+-----+	Hopper	42	
+-----+			

La méthode `createDataFrame`



- Très verbeux
- Beaucoup d'étapes intermédiaires
- Peu utilisé



La méthode toDF

```
import spark.implicits._

val dataFrame: DataFrame = List(
    ("Turing", 21),
    ("Hopper", 42)
).toDF("name", "age")
```

```
+-----+
| name|age|
+-----+
| Turing| 21|
| Hopper| 42|
+-----+
```

La méthode toDF avec case class



```
import spark.implicits._

case class Person(name: String, age: Int)

val dataFrame: DataFrame = List(
    Person("Turing", 21),
    Person("Hopper", 42)
).toDF()
```

```
+-----+
| name|age|
+-----+
| Turing| 21|
| Hopper| 42|
+-----+
```



La méthode toDF

- Beaucoup moins verbeux
- Plus proche du Scala natif
- Recommandé
- *spark.implicits._* permet d'ajouter la méthode toDF à l'objet List



Créer de la donnée - en Python



La méthode createDataFrame

```
from pyspark.sql import Row  
  
dataFrame = spark.createDataFrame([  
    Row(name='Turing', age=21),  
    Row(name='Hopper', age=42)  
])
```

```
dataFrame = spark.createDataFrame(  
    [  
        ('Turing', 21),  
        ('Hopper', 42)  
    ],  
    ['name', 'age'])
```

```
+-----+  
| name | age |  
+-----+  
| Turing | 21 |  
| Hopper | 42 |  
+-----+
```



Tests unitaires

Tests unitaires – Tester une valeur



```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.{col, max}

def getMaxFromColumn(df: DataFrame, columnName: String): Double = {
    val maxColumnName = s"${columnName}_max"

    df
        .agg(max(col(columnName)).as(maxColumnName))
        .first()
        .getAs[Double](maxColumnName)
}
```

Tests unitaires - Tester une valeur



```
"getMaxFromColumn" should "return the max value of a given column in a given DataFrame" in {
    import spark.implicits._

    // Given
    val df = List(1d, 3d, 4d, 1d).toDF("double-col")

    // When
    val actual = getMaxFromColumn(df, "double-col")

    // Then
    actual shouldBe 4.0
}
```

Tests unitaires – Tester une valeur



```
package fr.hymaia.training.sparkfordev.testing

import org.apache.spark.sql.SparkSession
import fr.hymaia.training.sparkfordev.testing.Testing.getMaxFromColumn
import org.scalatest.{FlatSpec, Matchers}

class TestingTest extends FlatSpec with Matchers {
    val spark: SparkSession = SparkSession.builder().master("local[*]").getOrCreate()

    import spark.implicits._

    "getMaxFromColumn" should "return the max value of a given column in a given DataFrame" in {
        // Given
        val df = List(1d, 3d, 4d, 1d).toDF("double-col")

        // When
        val actual = getMaxFromColumn(df, "double-col")

        // Then
        actual shouldBe 4.0
    }
}
```

Full test class example Scala

Tests unitaires – Tester une valeur



```
import unittest
from pyspark.sql import SparkSession
from pyspark.sql import Row
from testing.testing_file import get_max_from_column

class SimpleSQLTest(unittest.TestCase):
    spark = SparkSession.builder.master("local[*]").getOrCreate()

    def test_should_return_max_value(self):
        # Given
        df = self.spark.createDataFrame([Row(doubleCol=1), Row(doubleCol=2)])

        # When
        actual = get_max_from_column(df, "doubleCol")

        # Then
        self.assertEquals(actual, 2.0)
```

Full test class example Python

Tests unitaires - Tester une valeur



- Besoin d'un SparkSession pour les tests
- Créer sa donnée en input
- Utilisation d'un comparatif classique
 - Scala : shouldBe
 - Python : assertEquals

Tests unitaires - Tester un Dataframe



```
import org.apache.spark.sql.DataFrame  
  
import org.apache.spark.sql.functions.col  
  
val ADULT_AGE = 18  
  
def keepAdults(dataFrame: DataFrame, ageColumnName: String): DataFrame = {  
    dataFrame  
        .filter(col(ageColumnName) >= ADULT_AGE)  
}
```

Tests unitaires - Tester un Dataframe



```
import spark.implicits._

"keepAdults" should "return a DataFrame with only people over 18 in it" in {
    // Given
    val df = List(
        ("Turing", 21),
        ("Gates", 12),
        ("Hopper", 42)
    ).toDF("name", "age")

    // When
    val actual = keepAdults(df, "age")
```

Tests unitaires - Tester un Dataframe



```
...
// Then
val expected = Array(
  Row("Turing", 21),
  Row("Hopper", 42)
)
actual.collect() should contain theSameElementsAs expected
}
```

Tests unitaires - Tester un Dataframe via un Dataset



```
case class Person(name: String, age: Int)

import spark.implicits._

"keepAdults" should "return a DataFrame with only people over 18 in it" in {
    // Given
    val df = List(
        Person("Turing", 21),
        Person("Gates", 12),
        Person("Hopper", 42)
    ).toDF()
```

Tests unitaires - Tester un Dataframe via un Dataset



```
...
// When
val actual = keepAdults(df, "age")

// Then
val expected = Array(
    Person("Turing", 21),
    Person("Hopper", 42)
)
actual.as[Person].collect() should contain theSameElementsAs expected
}
```

Tests unitaires - Tester un Dataframe



- Pour utiliser un comparateur scalatest on collect la Dataframe
- On peut utiliser une case class pour tester le schéma

Tests unitaires - Comparer directement des Dataframe



- Il existe des bibliothèques pour tester du code Spark
- Développé par Holden Karau
- <https://github.com/holdenk/spark-testing-base>
- Plusieurs modules
 - SharedSparkContext : Gère un SparkSession partagé à travers vos tests
 - DataFrameSuiteBase : Fournis les comparateurs pour 2 Dataframes
 - DatasetSuiteBase : Fournis les comparateurs pour 2 Datasets
 - ...

Tests unitaires - Comparer directement des Dataframe



```
libraryDependencies +=  
  "com.holdenkara" %% "spark-testing-base" % "3.2.0_1.1.1" % Test
```

build.sbt

```
<dependency>  
  <groupId>com.holdenkara</groupId>  
  <artifactId>spark-testing-base_2.12</artifactId>  
  <version>3.2.0_1.1.1</version>  
  <scope>test</scope>  
</dependency>
```

pom.xml

Tests unitaires - Comparer directement des Dataframe



```
import com.holdenkarau.spark.testing.DataFrameSuiteBase
import fr.hymaia.training.sparkfordev.testing.KeepAdults
import org.scalatest.{FlatSpec, Matchers}

class TestingTestWithSuiteBase extends FlatSpec with Matchers with DataFrameSuiteBase {
  import spark.implicits._

  "keepAdults" should "return a DataFrame with only people over 18 in it" in {
    // Given
    val df = List(
      Person("Turing", 21),
      Person("Gates", 12)
    ).toDF()

    // When
    val actual = keepAdults(df, "age")

    // Then
    val expected = List(
      Person("Turing", 21)
    ).toDF()

    assertDataFrameEquals(actual, expected)
  }
}
```

spark-testing-base library:

- S'utilise comme le reste de Scalatest, par héritage
- Fournit un SparkSession
- assertDataFrameEquals permet de comparer 2 Dataframes
- Messages d'erreur plus précis :
 - Quand le schéma est différent
 - Quand le nombre de ligne est différent
 - Affiche un diff des lignes différentes

Tests unitaires - spark-testing-base python



- Aucune mise à jour depuis plusieurs années
- À oublier
- On restera sur des collect()

Tests unitaires - spark-testing-base vs scalatest natif



Libraries spark-testing-base

✓ Message d'erreur explicit en cas d'échec (schéma, nombre d'éléments...)

✗ Les Dataframes à comparer doivent être dans le même ordre, un sort peut être nécessaire

Classic mode

✗ Le message d'erreur sur 2 collections différentes est moins précis et ne gère pas les schémas

✓ La fonction **theSameElementsAs** n'a pas besoin d'avoir des collections dans le même ordre



Tests d'intégrations

Tests d'intégrations – Définition



On appelle test d'intégration un test qui couvre toutes les transformations apportées à la donnée entre la lecture et l'écriture de celle-ci.

Tests d'intégrations – Scénario



Nous avons 2 tables **people** et **city**

people

name	age	zip
Turing	21	10001
Gates	12	10001
Hoper	42	90001

city

zip	city
10001	New York
90001	Los Angeles



Tests d'intégrations – Scénario

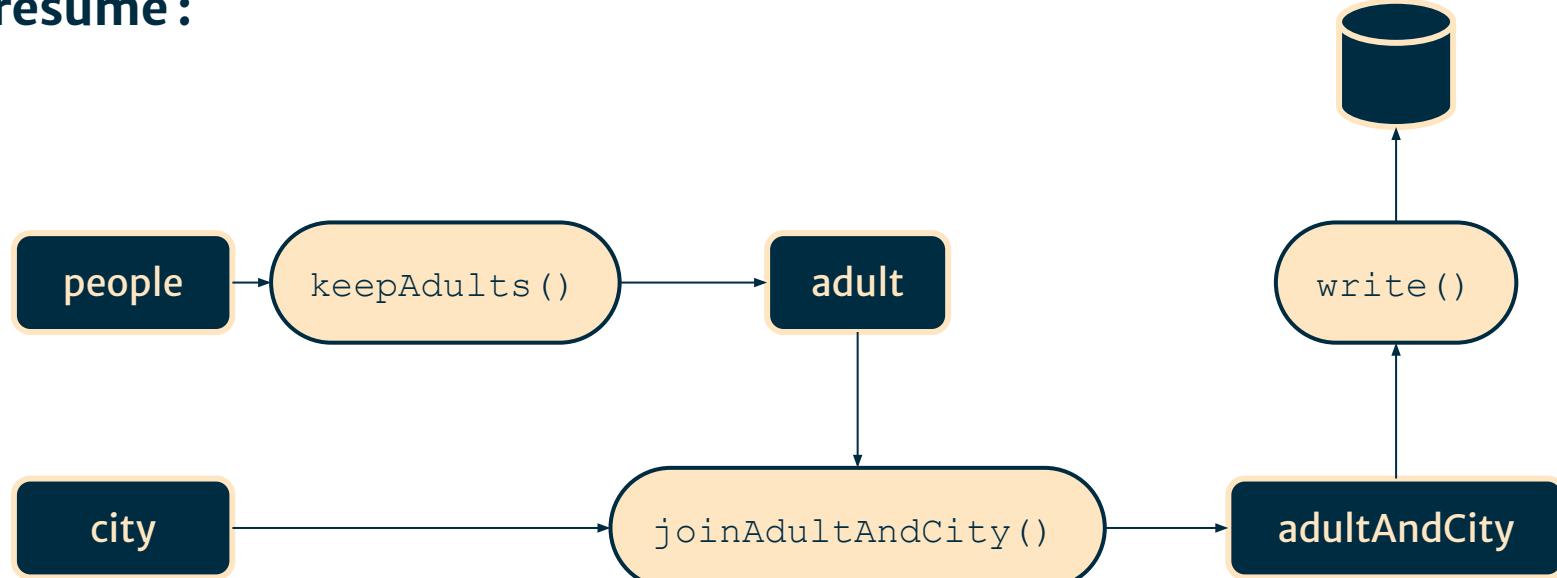
Notre job Spark ne garde que les adultes, ajoute le nom des villes et écrit le résultat que voici :

name	age	zip	city
Turing	21	10001	New York
Hoper	42	90001	Los Angeles

Tests d'intégrations – Scénario



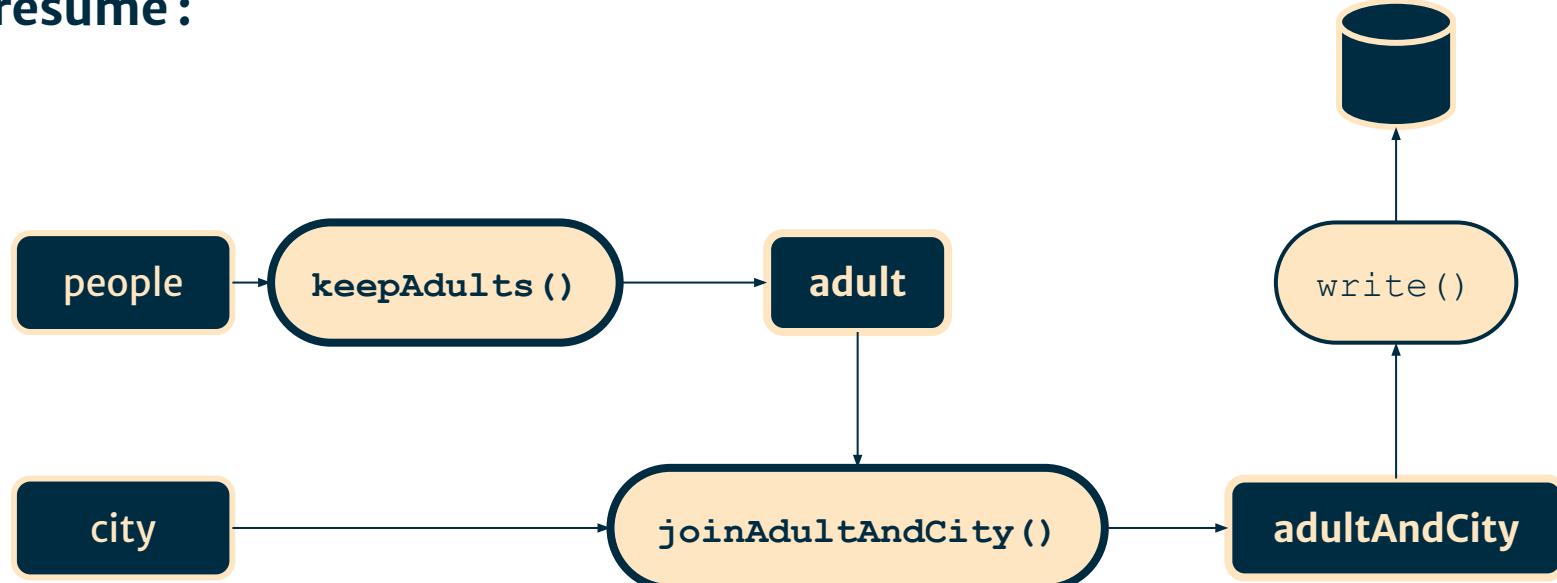
En résumé :



Tests d'intégrations – Scénario



En résumé :



Tests d'intégrations – Fonction keepAdults



```
import org.apache.spark.sql.DataFrame  
  
import org.apache.spark.sql.functions.col  
  
val ADULT_AGE = 18  
  
def keepAdults(people: DataFrame, ageColumn: String): DataFrame = {  
    people.filter(col(ageColumn) >= ADULT_AGE)  
}
```

Tests d'intégrations - Fonction joinAdultAndCity



```
import org.apache.spark.sql.DataFrame

def joinAdultAndCity(adult: DataFrame,
                     city: DataFrame,
                     joinColumn: String): DataFrame = {
    adult.join(city, Seq(joinColumn), "left")
}
```

Tests d'intégrations - Fonction joinAndWritePeopleAndCity



```
import org.apache.spark.sql.{DataFrame, SparkSession}

def getAdultAndCity(people: DataFrame, city: DataFrame): DataFrame = {
    val adult = keepAdults(people, "age")
    val adultAndCity = joinAdultAndCity(adult, city, "zip")

    returns adultAndCity
}
```

Tests d'intégrations – Job Spark



```
import org.apache.spark.sql.{DataFrame, SparkSession}

def main(args: Array[String]): Unit = {
    val people: DataFrame = spark.read.csv("people")
    val city: DataFrame = spark.read.csv("city")

    val adultAndCity = getAdultAndCity(people, city)

    adultAndCity.write.parquet("people_and_city")
}
```

Tests d'intégrations – Créer le modèle



```
case class People(name: String, age: Int, zip: String)
```

```
case class City(zip: String, city: String)
```

```
case class PeopleAndCity(name: String, age: Int, zip: String, city: String)
```

Tests d'intégrations – Création du SparkSession



```
val spark: SparkSession = SparkSession  
  .builder()  
  .master("local[*]")  
  .getOrCreate()  
  
import spark.implicits._
```

Tests d'intégrations – Création de la donnée



```
// Given  
  
val people = List(  
    People("Turing", 21, "10001"),  
    People("Gates", 12, "10001"),  
    People("Hopper", 42, "90001")  
).toDF()
```

```
val city = List(  
    City("10001", "New York"),  
    City("90001", "Los Angeles")  
).toDF()
```

Tests d'intégrations – Exécution de la fonction



```
...  
  
// When  
  
val actual = getAdultAndCity(people, city)
```

```
...
```

Tests d'intégrations – Vérification des résultats



```
...  
  
// Then  
  
val expected = Array(  
    PeopleAndCity("Turing", 21, "10001", "New York"),  
    PeopleAndCity("Hopper", 42, "90001", "Los Angeles")  
)  
  
actual.as[PeopleAndCity].collect() should contain theSameElementsAs expected
```



Partager son SparkSession



Partager son SparkSession

- Créer un SparkSession prend du temps
- Aucun intérêt à en créer 1 par classe de test
- On en crée 1 qu'on partage à tout le monde
 - Concept de spark-testing-base

Partager son SparkSession



- On crée une SparkSession dans un singleton
- On peut changer le niveau de logs
- Des optimisations sont possibles

```
object SparkSessionProvider {  
  
    val spark: SparkSession = SparkSession  
        .builder()  
        .appName("Test Spark")  
        .master("local[*]")  
        .getOrCreate()  
  
    spark.sparkContext.setLogLevel("ERROR")  
  
}
```



Partager son SparkSession

```
import fr.hymaia.training.sparkfordev.SparkSessionProvider.spark

class MyTest extends FlatSpec with Matchers {

    import spark.implicits._

    "My test" should "..." in {
        // The test content
    }
}
```

Partager son SparkSession - optimisation



- Peu de données dans les tests
- Shuffle = 200 partitions par défaut
- Beaucoup de partitions vides
- Changer cette configuration pour 3
- Désactiver l'optimisation Spark

```
object SparkSessionProvider {  
  
    val spark: SparkSession = SparkSession  
        .builder()  
        .appName("Test Spark")  
        .master("local[*]")  
        .conf("spark.sql.shuffle.partitions", 3)  
        .conf("spark.sql.adaptive.enabled", false)  
        .getOrCreate()  
  
    spark.sparkContext.setLogLevel("ERROR")  
}
```



Take away

- Il faut créer son jeu de données dans chaque test
- En Scala, les Datasets facilitent la comparaison avec schéma
- La bibliothèque spark-testing-base apporte des comparateurs spécifiques à Spark
- Il faut partager son SparkSession
- Des optimisations sont possibles pour réduire la durée des tests



Le fonctionnement interne de Spark



Programme

- Le cache
- La gestion de la mémoire
- Les jointures distribuées
- Le shuffle



Comment éviter de tout recalculer à chaque action ?



Le cache

- df.persist()
 - Mémoire vive
 - Disque dure
 - Mixte
- Réutiliser une DataFrame
- C'est une suggestion



Options du persist

- **MEMORY_ONLY** (par défaut)
- **MEMORY_AND_DISK**
 - Si ça ne rentre pas entièrement en mémoire vive
- **MEMORY_ONLY_SER, MEMORY_AND_DISK_SER**
 - La sérialisation réduit la taille mais augmente le temps de traitement
- **DISK_ONLY**
- **MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc**
 - RéPLICATION de la donnée sur un autre exécuteur

Example



data.parquet



name	town	age
	.	



name	age

```
val myDF = spark.read.parquet("data")
    .select("name","age")
```

Example



data.parquet



name	town	age



name	age

age \geq 35 age $<$ 35

name	age

name	age

```
val myDF = spark.read.parquet("data")
    .select("name", "age")
myDF.where("age >= 35")
myDF.where("age < 35")
```

Example



data.parquet



name	town	age
turing	london	21
hopper	new york	42
...



name	age
turing	21
hopper	42
...	...

age ≥ 35 ↗ age < 35 ↘

name	age
hopper	42
...	...

name	age



472876

```
val myDF = spark.read.parquet("data")
    .select("name", "age")
myDF.where("age >= 35").count()
myDF.where("age < 35").count()
```

Example



data.parquet



name	town	age



name	age

age ≥ 35

name	age

age < 35

name	age

```
val myDF = spark.read.parquet("data")
    .select("name","age")
myDF.where("age >= 35").count()
myDF.where("age < 35").count()
```

Example



data.parquet



name	town	age
turing	london	21
hopper	new york	42
...



name	age
turing	21
hopper	42
...	...

age ≥ 35



name	age

age < 35



name	age
turing	21
...	...



563543

```
val myDF = spark.read.parquet("data")
    .select("name","age")
myDF.where("age >= 35").count()
myDF.where("age < 35").count()
```

Example



data.parquet



name	town	age



name	age

```
val myDF = spark.read.parquet("data")
    .select("name","age")
    .persist()
```

Example



data.parquet



name	town	age



name	age

age ≥ 35



name	age

age < 35



name	age

```
val myDF = spark.read.parquet("data")
    .select("name","age")
    .persist()
myDF.where("age >= 35").count()
myDF.where("age < 35").count()
```

Example



data.parquet



name	town	age
turing	london	21
hopper	new york	42
...



name	age
turing	21
hopper	42
...	...

age ≥ 35

name	age
hopper	42
...	...

age < 35

name	age

472876

```
val myDF = spark.read.parquet("data")
    .select("name","age")
    .persist()
myDF.where("age >= 35").count()
myDF.where("age < 35").count()
```

Example



data.parquet



name	town	age



name	age
turing	21
hopper	42
...	...

age ≥ 35



name	age

563543

age < 35



name	age
turing	21
...	...

```
val myDF = spark.read.parquet("data")
    .select("name","age")
    .persist()
myDF.where("age >= 35").count()
myDF.where("age < 35").count()
```

Cache : Spark UI



Jobs

Stages

Storage

Environment

Executors

SQL

Zeppelin application UI

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
*FileScan csv [col1#11,col2#12] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/sagean/test1.csv, file:/home/sagean/test2.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<col1:string,col2:int>	Memory Deserialized 1x Replicated	2	100%	440.0 B	0.0 B



Cache : quand l'utiliser ?

- Plus d'une action
- Traitement complexe (read + filter non)
- Peut résoudre des problèmes de plan catalyst



Unpersist ?

- `df.unpersist()`
- C'est une suggestion
- Si vous le faites pas, Spark supprime le cache le moins récemment utilisé
 - LRU (least recently used)
- Pas très important



Comment est gérée la mémoire dans **Spark** ?



Executor memory

- `spark.executor.memory = 16 Go`
- `java heap = spark.executor.memory - spark.executor.memoryOverhead`
- `spark.executor.memoryOverhead =`
 - `spark.executor.memory x 0.1`
 - Minimum 384 mo

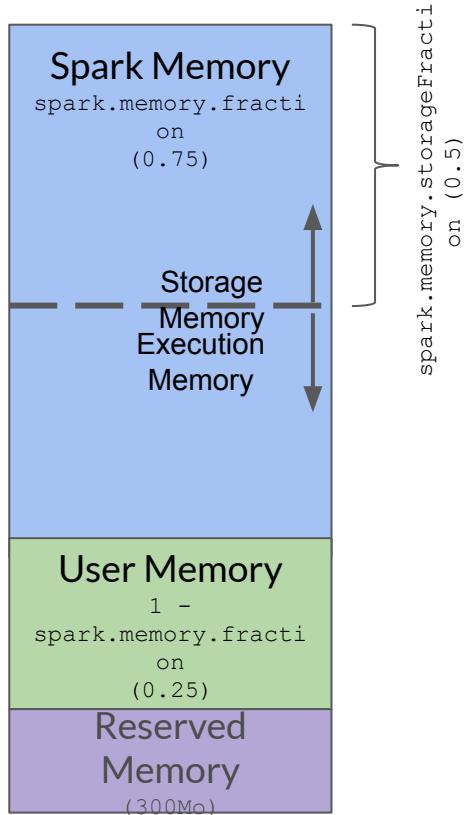
`spark.executor.memory = 16 Go`

memoryOverhead
:
1.6 Go

java heap : 14.4 Go



Spark Memory (>=1.6)



- **Reserved Memory**

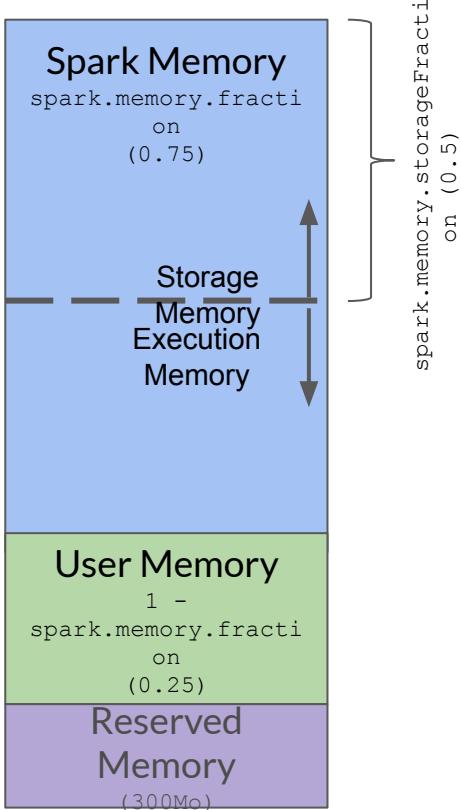
- Réservée par Spark et ne peut pas être changée (300 mo)
- Utilisée pour les objets Java de Spark

- **User Memory**

- $\text{size} = (\text{java heap} - \text{reserved memory}) * (1 - \text{spark.memory.fraction})$
- `spark.memory.fraction = 0.75`
- Utilisée pour les mapPartitions
- Attention aux OOMs



Spark Memory (>=1.6)



- **Spark Memory**

- $\text{size} = (\text{java heap} - \text{reserved memory}) * \text{spark.memory.fraction}$
- $\text{spark.memory.fraction} = 0.75$

- **Storage Memory**

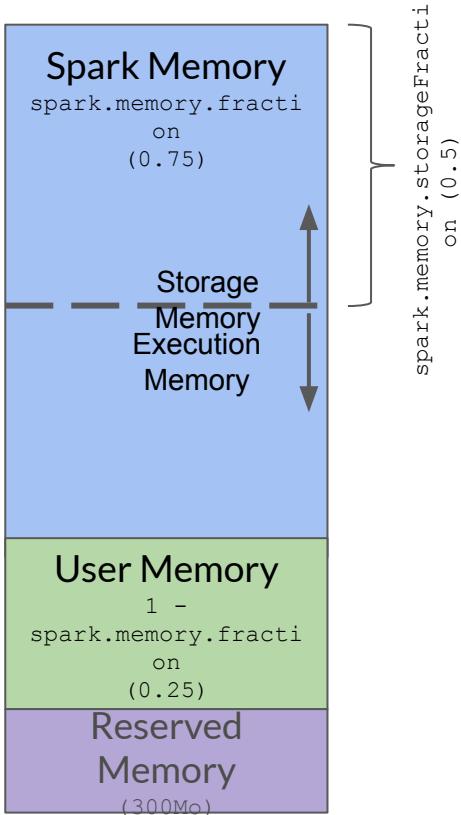
- Données cachées
- Variable broadcast

- **Execution Memory**

- Hash table
- Résultat de calcul intermédiaire
- Peut déborder sur le disque dur



Spark Memory (>=1.6)



- Spark **ne peut pas** supprimer un bloc de **l'Execution Memory**
- Spark **peut** supprimer un bloc de la **Storage Memory**
(déborde sur disque dure ou suppression)
- **Storage et Execution Memory** peuvent **déborder** l'un sur l'autre s'il reste de **l'espace libre**
- `spark.memory.storageFraction` (0.5 par défaut)



Comment faire pour avoir 16Go de
mémoire pour mes traitements **Spark** ?



Exercice

- Java heap : 16.3 Go
- `spark.memory.fraction = 0.75`
- `spark.memory.storageFraction = 0.5`
- Reserved memory = ?
- User memory = ?
- Spark memory = ?
 - Storage memory = ?
 - Execution memory = ?
- `spark.executor.memory = ?`



Exercice

- Java heap : 16.3 Go
- `spark.memory.fraction = 0.75`
- `spark.memory.storageFraction = 0.5`
- Reserved memory = **300 mo**
- User memory = ?
- Spark memory = ?
 - Storage memory = ?
 - Execution memory = ?
- `spark.executor.memory = ?`



Exercice

- Java heap : 16.3 Go
- spark.memory.fraction = 0.75
- spark.memory.storageFraction = 0.5
- Reserved memory = 300 mo
- User memory = **4 Go**
- Spark memory = **12 Go**
 - Storage memory = ?
 - Execution memory = ?
- spark.executor.memory = ?



Exercice

- Java heap : 16.3 Go
- spark.memory.fraction = 0.75
- spark.memory.storageFraction = 0.5
- Reserved memory = 300 mo
- User memory = 4 Go
- Spark memory = 12 Go
 - Storage memory = **6 Go**
 - Execution memory = **6 Go**
- spark.executor.memory = ?

Exercice



- Java heap : 16.3 Go
- spark.memory.fraction = 0.75
- spark.memory.storageFraction = 0.5
- Reserved memory = 300 mo
- User memory = 4 Go
- Spark memory = 12 Go
 - Storage memory = 6 Go
 - Execution memory = 6 Go
- **spark.executor.memory = 18.11 Go**



Comment bien configurer ses exéuteurs ?

Comment Spark crée ses partitions ?



- Dépend du parallélisme
- Dépend de la taille du fichier
 - < 4 mo : 1 partition tous les 4 mo
 - > 128 mo : 1 partition tous les 128 mo
 - Sinon : autant de partition que de parallélisme

```
spark.sql.files.maxPartitionBytes = 128 mo  
  
spark.sql.files.openCostInBytes = 4 mo  
  
spark.sql.files.minPartitionNum = spark.default.parallelism
```

 Lien

La parallélisation des tâches



- Un exécuteur peut traiter plusieurs partitions en parallèle
 - parallélisme = `spark.executor.cores` / `spark.task cpus`
 - Plus le parallélisme est élevé, plus un exécuteur à besoin de mémoire
- Peu d'exéuteurs signifie
 - Moins de données à transférer en cas de broadcast
 - Besoin de plus de ressources pour conserver de bonnes performances



Comment **Spark** fait une jointure ?

Le partitionnement des Dataframes



- Spark gère tout seul le nombre de partition d'un Dataframe
 - Selon les règles vues pour la lecture d'un fichier
 - Selon l'opération de reduce exécutée
- Lors d'un shuffle, Spark crée toujours le même nombre de partitions
 - `spark.sql.shuffle.partitions = 200`
 - Depuis Spark 3, cette valeur peut changer au cours d'un shuffle



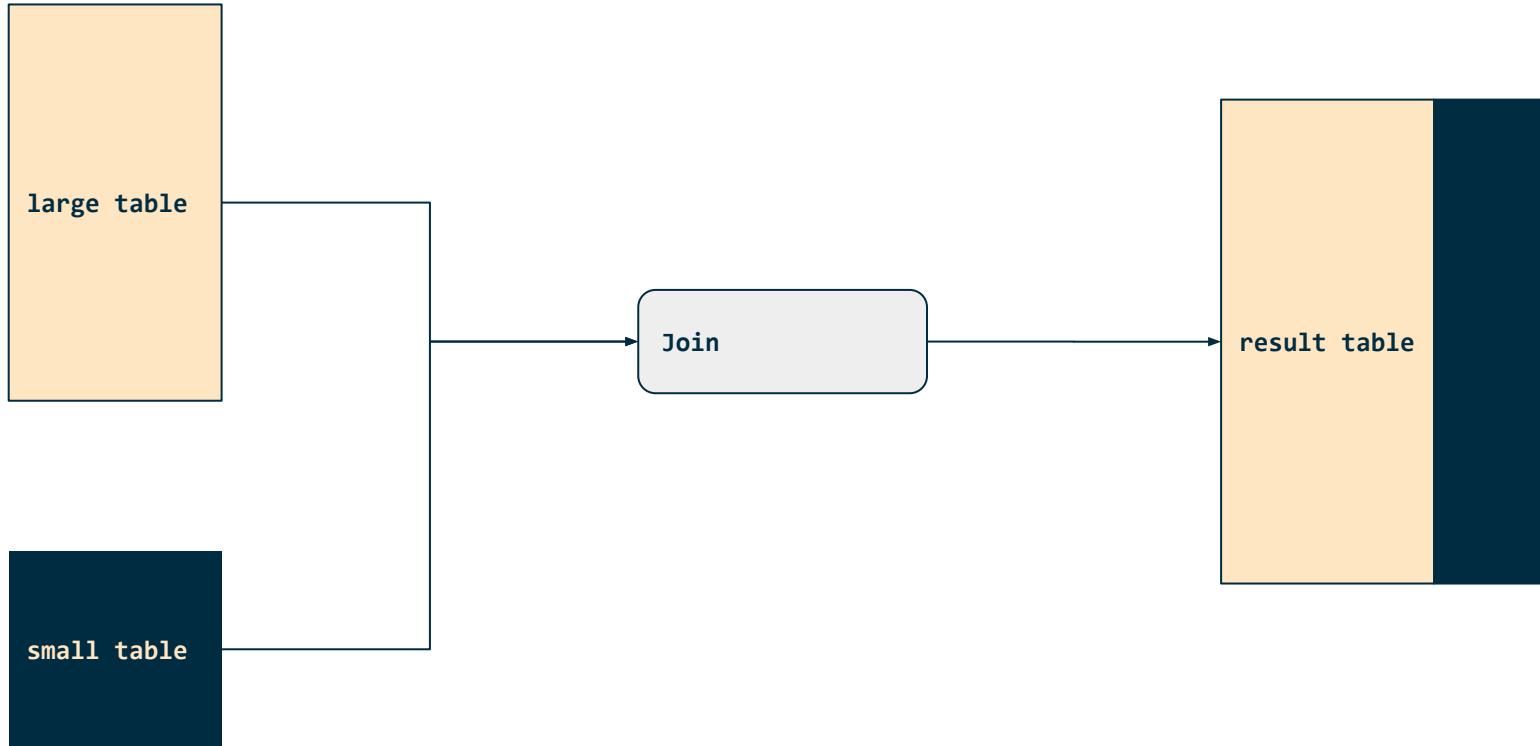
Les différents join

- inner (par défaut)
- left
- right
- left_outer
- right_outer
- full_outer
- leftsemi
- leftanti
- crossjoin (cartesian)

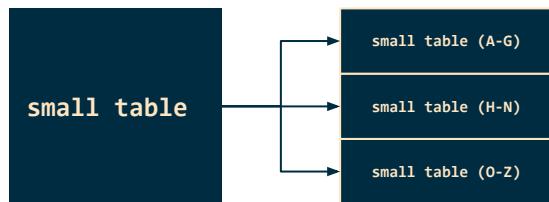
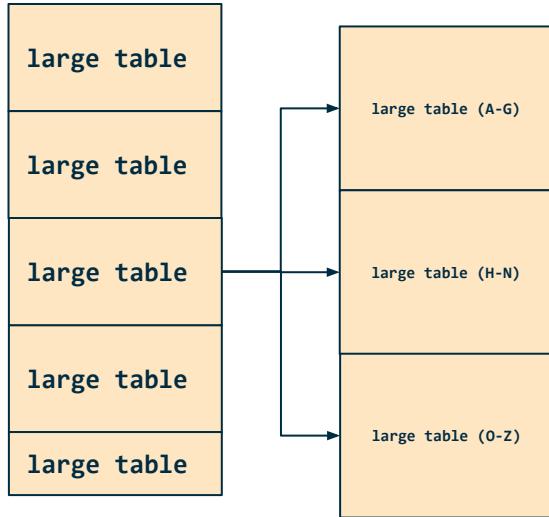
```
df1.join(df2, "column1")
df1.join(df2, Seq("column1"))
df1.join(df2,
          df1("column_a1") === df2("column_b1"))
```



Les différents join

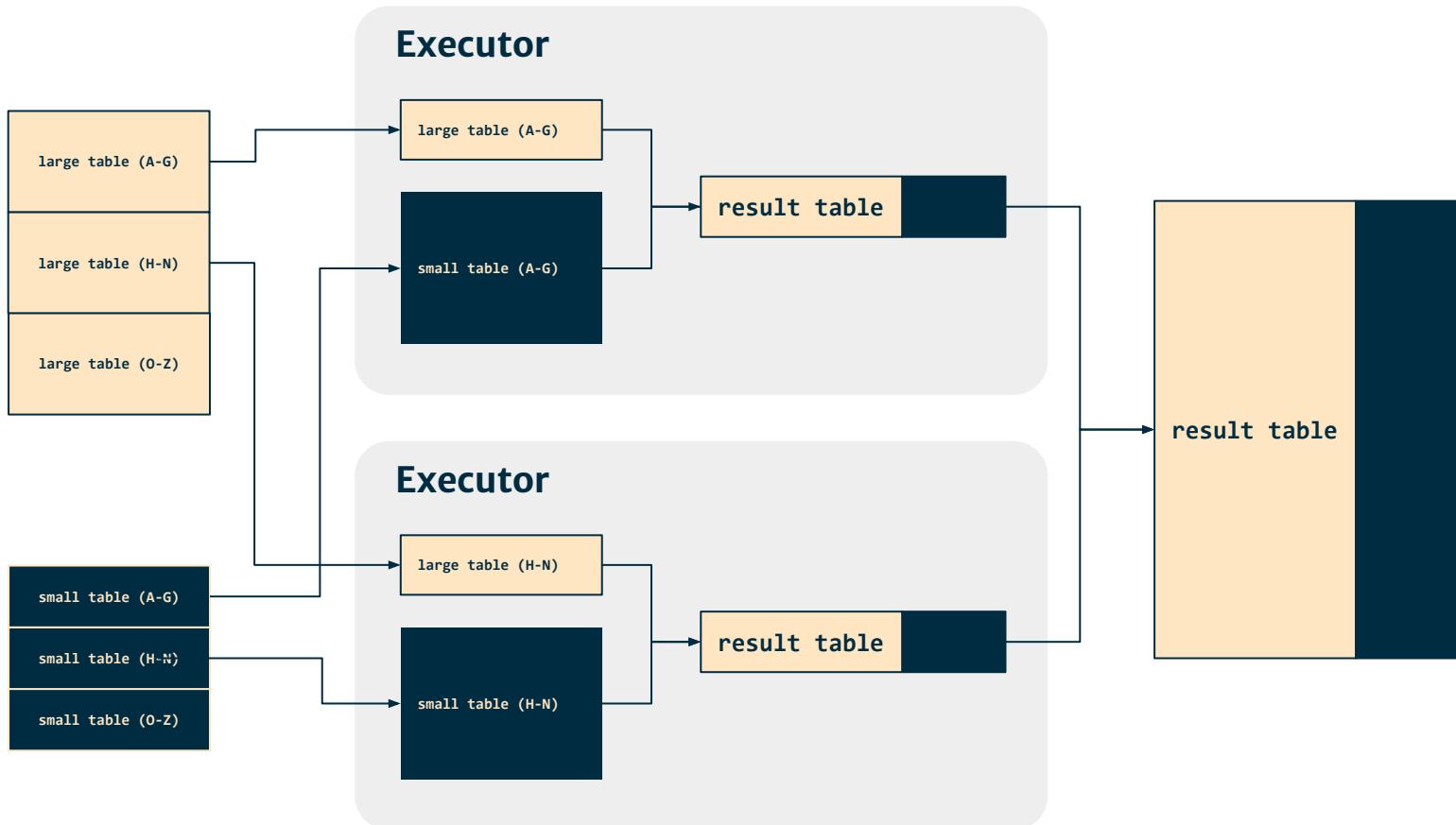


Les différents join



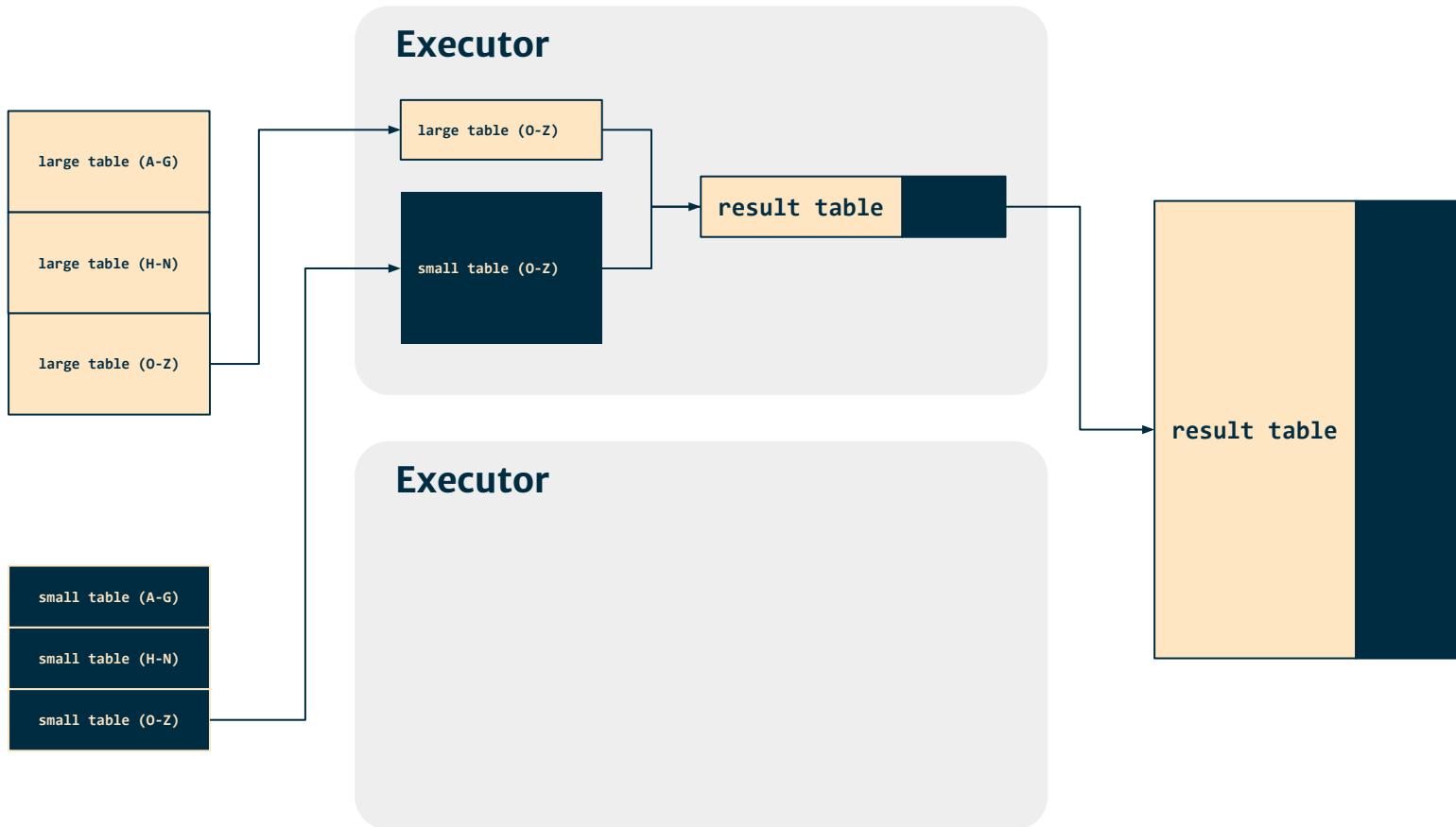


Les différents join





Les différents join

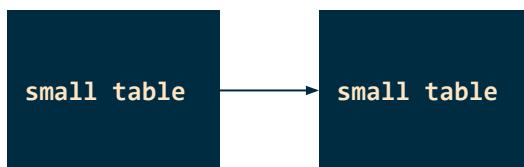
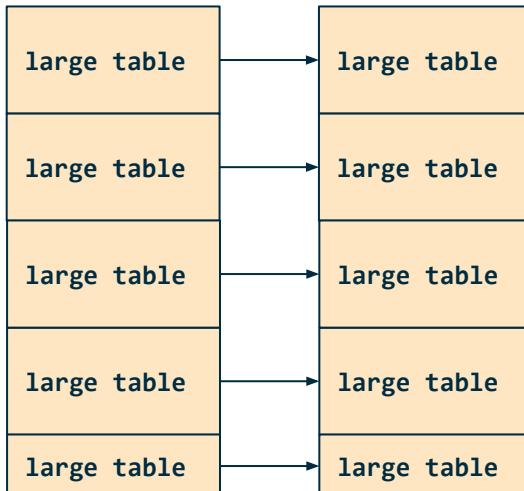


Broadcast Join

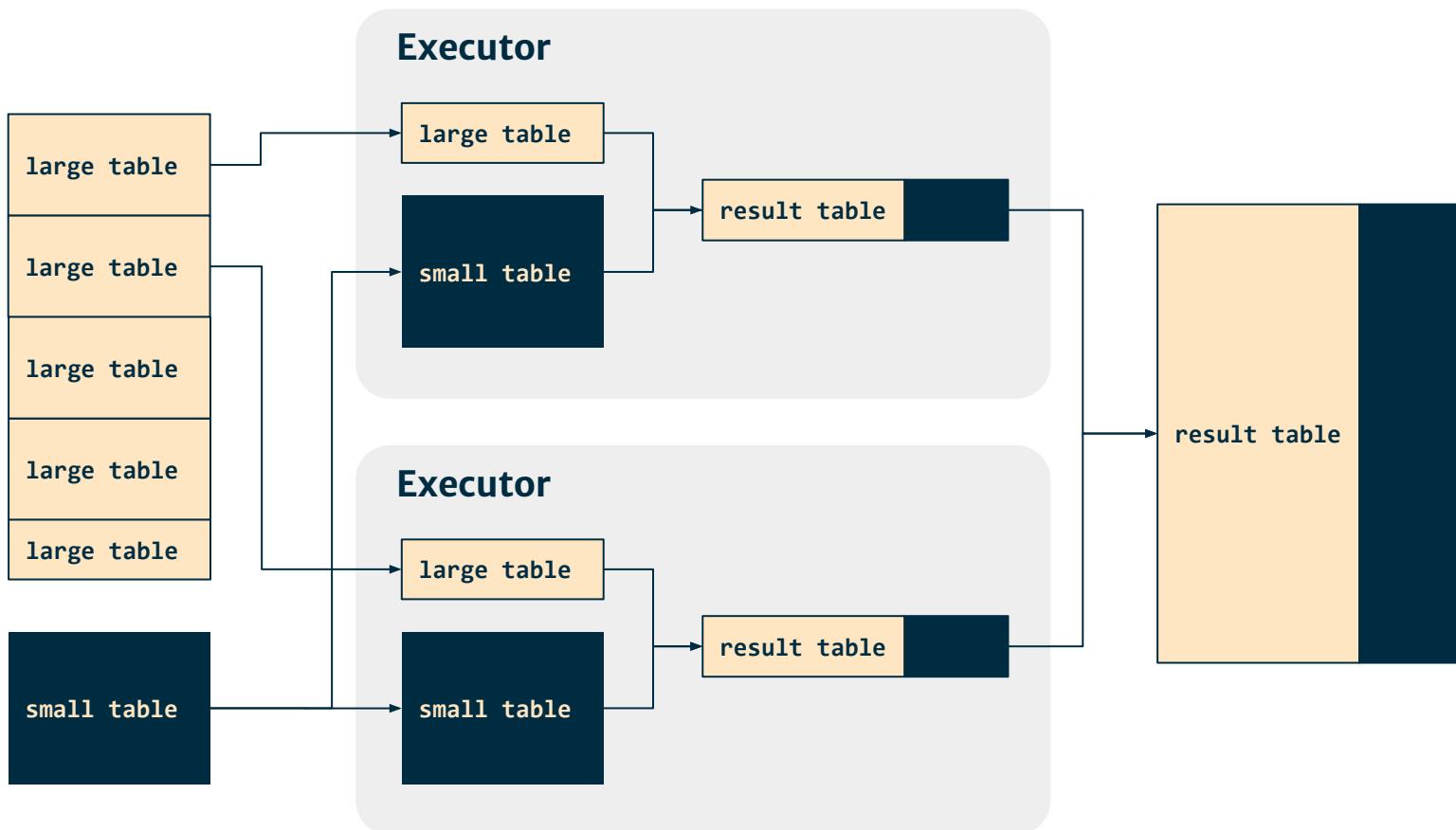


- Un join sans shuffle
- Beaucoup plus performant
- Nécessite qu'une des deux Dataframes soit petite
- La petite DataFrame est envoyée à tous les exécuteurs
 - Elle doit pouvoir entrer en mémoire
- `spark.sql.autoBroadcastJoinThreshold = 10Mb`
- C'est automatique

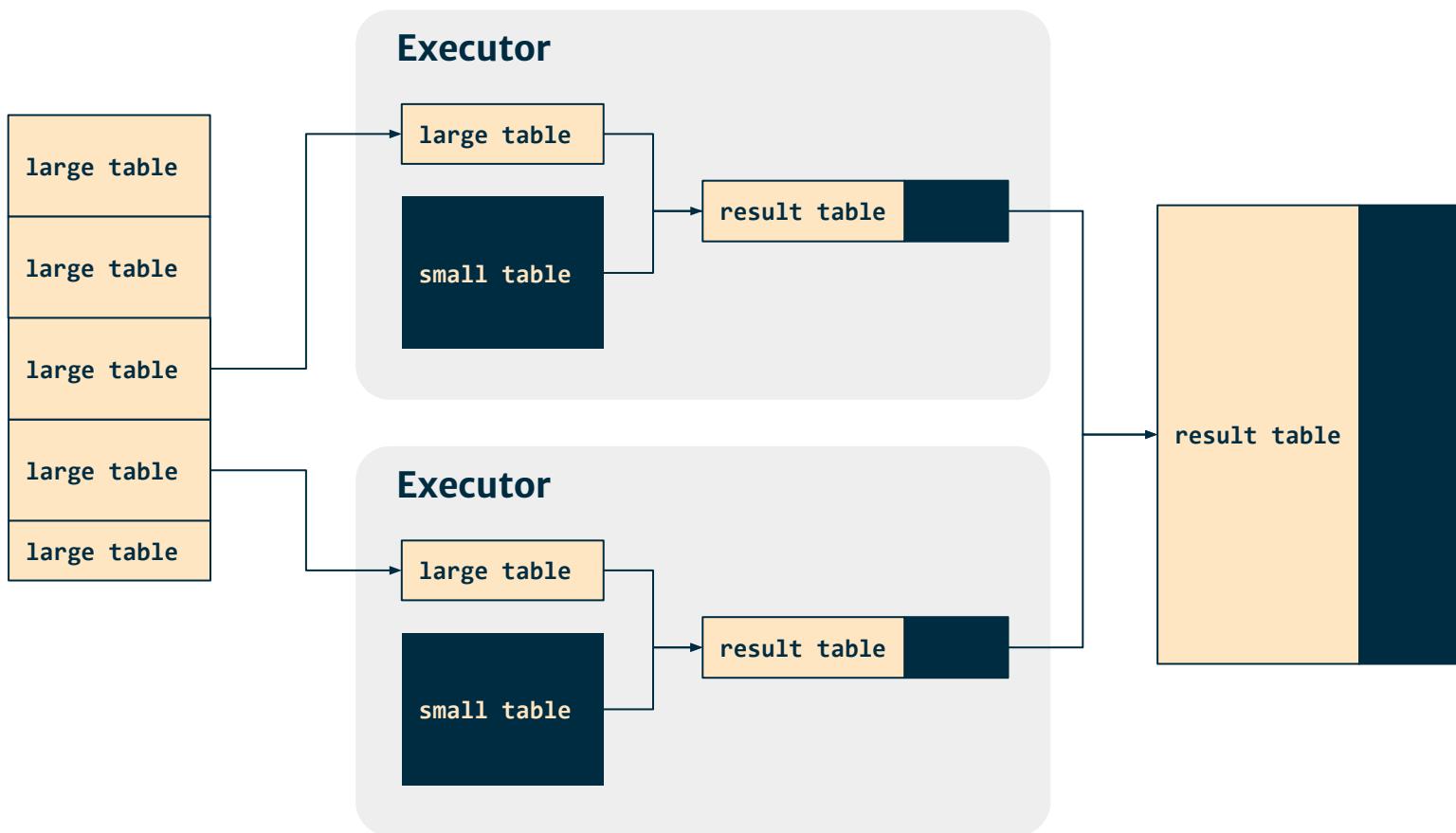
Broadcast Join



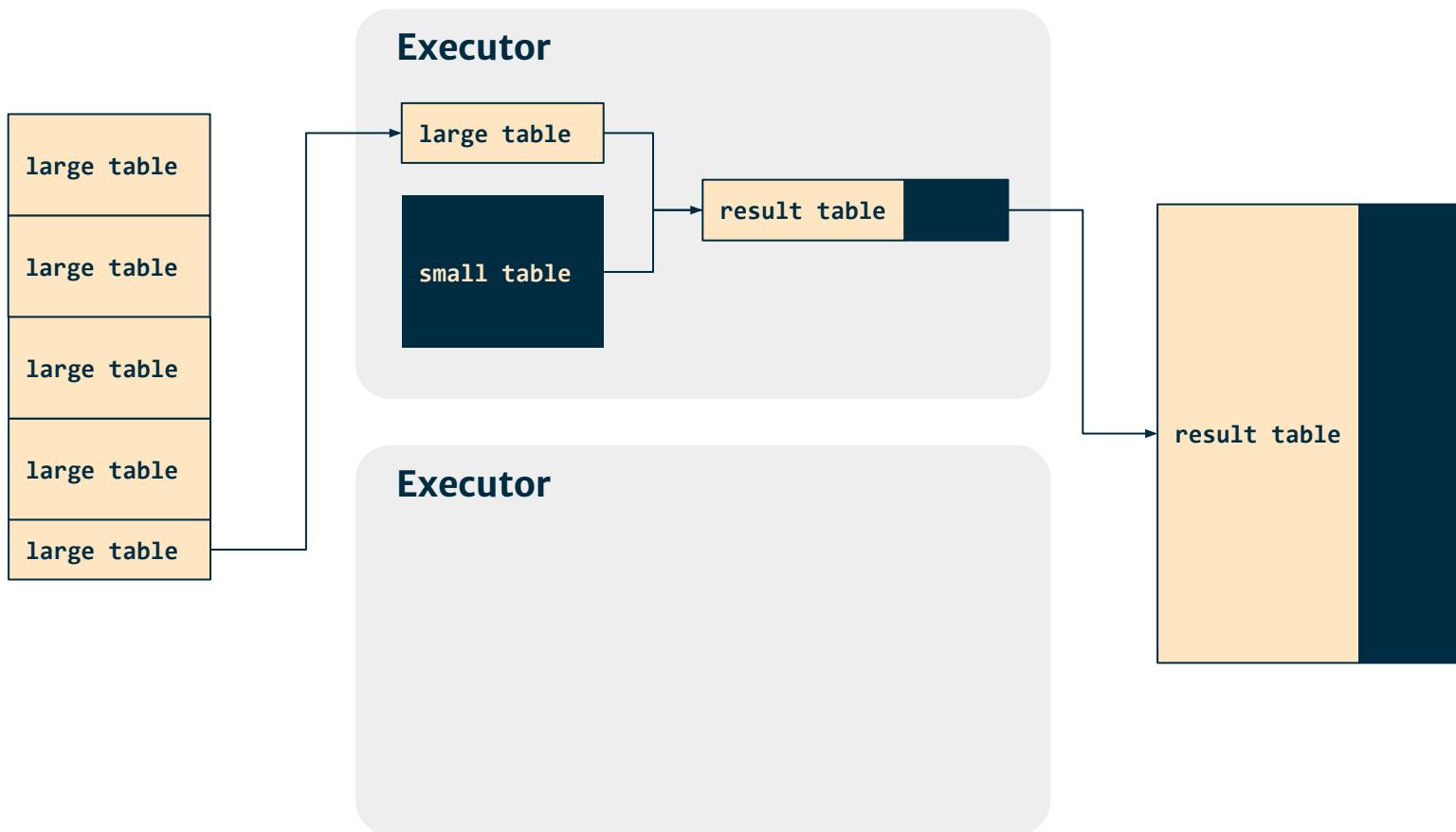
Broadcast Join



Broadcast Join



Broadcast Join





Les variables broadcast

- `val broadcastedVariable = spark.sparkContext.broadcast(myVar)`
- `broadcastedVariable.value()`
- `.unpersist()`
 - Supprime la variable des exéuteurs
- `.destroy()`
 - Détruit toute trace de la variable
- Peu utile

Qu'est-ce qu'on appelle le shuffle ?



- Envoyer les lignes correspondant à la même clé dans le même exécuteur
- La donnée est écrite sur le File System local
 - spark.shuffle.spill
- Spark a différents algorithmes pour résoudre son shuffle
 - Hash shuffle
 - Consolidate hash shuffle
 - Sort shuffle



Take away

- Le cache permet de gagner un temps considérable
- Attention à ne pas l'utiliser n'importe comment
- Un exécuteur n'utilise pas toute sa mémoire pour traiter la donnée
- Toutes les jointures ne provoquent pas de shuffle
- Les optimisations de shuffle sont automatiques
 - Catalyst



Les opérations de transformations de données avancées

Programme



- UDF
- Task not serializable error
- Window functions

User Defined Function



- Permet d'implémenter sa propre fonction sur une colonne
- Fonctionne avec les DataFrames et les Datasets



UDF - Exemple

Nous souhaitons récupérer le titre présent dans les noms

name
Allison, Miss. Helen Loraine
Andrews, Mr. Thomas Jr
Astor, Col. John Jacob
Carter, Mrs. Ernest Courtenay

UDF - 1ère méthode



- Définir une fonction extractTitle
- Appeler la fonction udf() dessus
- extractTitleUdf s'invoque comme n'importe quelle autre fonction Spark

```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.{col, udf}

val peopleDF: DataFrame = ...
val extractTitle: String => String = _.split(" ")(1)
val extractTitleUdf = udf(extractTitle)

peopleDF
  .withColumn("title", extractTitleUdf(col("name")))
  .show(truncate = false)
```

name	title
Allison, Miss. Helen Loraine	Miss.
Andrews, Mr. Thomas Jr	Mr.
Astor, Col. John Jacob	Col.
Carter, Mrs. Ernest Courtenay	Mrs.

UDF - 2ème méthode



- Définir une fonction directement la fonction udf()
- Ajouter les types d'entrée / sortie

```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.{col, udf}

val peopleDF: DataFrame = ...
val extractTitleUdf = udf[String, String](_.split(" ")(1))

peopleDF
  .withColumn("title", extractTitleUdf(col("name")))
  .show(truncate = false)
```

name	title
Allison, Miss. Helen Loraine	Miss.
Andrews, Mr. Thomas Jr	Mr.
Astor, Col. John Jacob	Col.
Carter, Mrs. Ernest Courtenay	Mrs.

UDF - 3ème méthode



- Créer une méthode explicite
- La syntaxe change un peu dans udf()
- Cette méthode est plus pratique pour les UDFs complexes

```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.{col, udf}

val peopleDF: DataFrame = ...
def extractTitle(name: String): String = name.split(" ")(1)
val extractTitleUdf = udf((s: String) => extractTitle(s))

peopleDF
  .withColumn("title", extractTitleUdf(col("name")))
  .show(truncate = false)
```

name	title
Allison, Miss. Helen Loraine	Miss.
Andrews, Mr. Thomas Jr	Mr.
Astor, Col. John Jacob	Col.
Carter, Mrs. Ernest Courtenay	Mrs.

UDF - 4ème méthode



```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.functions.{col, udf}

val peopleDF: DataFrame = ...
val extractTitleUdf = udf((name: String) => name.split(" ")(1))

peopleDF
  .withColumn("title", extractTitleUdf(col("name")))
  .show(truncate = false)
```

- Définir une lambda dans udf()

name	title
Allison, Miss. Helen Loraine	Miss.
Andrews, Mr. Thomas Jr	Mr.
Astor, Col. John Jacob	Col.
Carter, Mrs. Ernest Courtenay	Mrs.



- Les 1ère et 2ème méthodes sont peu utilisées
- La 3ème méthode est la plus pratique pour les TUs
- La 4ème méthode est la plus courte à écrire
- On recommandera la 3ème méthode

UDF - Python



En Python, il n'y a pas de type, il faut donc les définir explicitement

```
import pyspark.sql.functions as F
from pyspark.sql.types import StringType

extract_title_udf = F.udf(lambda name: name.split()[1], StringType())

df.withColumn('title', extract_title_udf(df.Name)).show(4)
```

```
import pyspark.sql.functions as F
from pyspark.sql.types import StringType

def extract_title(name):
    return name.split()[1]

extract_title_udf = F.udf(extract_title, StringType())

df.withColumn('title', extract_title_udf(df.Name)).show(4)
```



UDF - Python

- On peut aussi annoter une fonction Python avec @udf

```
import pyspark.sql.functions as F
from pyspark.sql.types import StringType

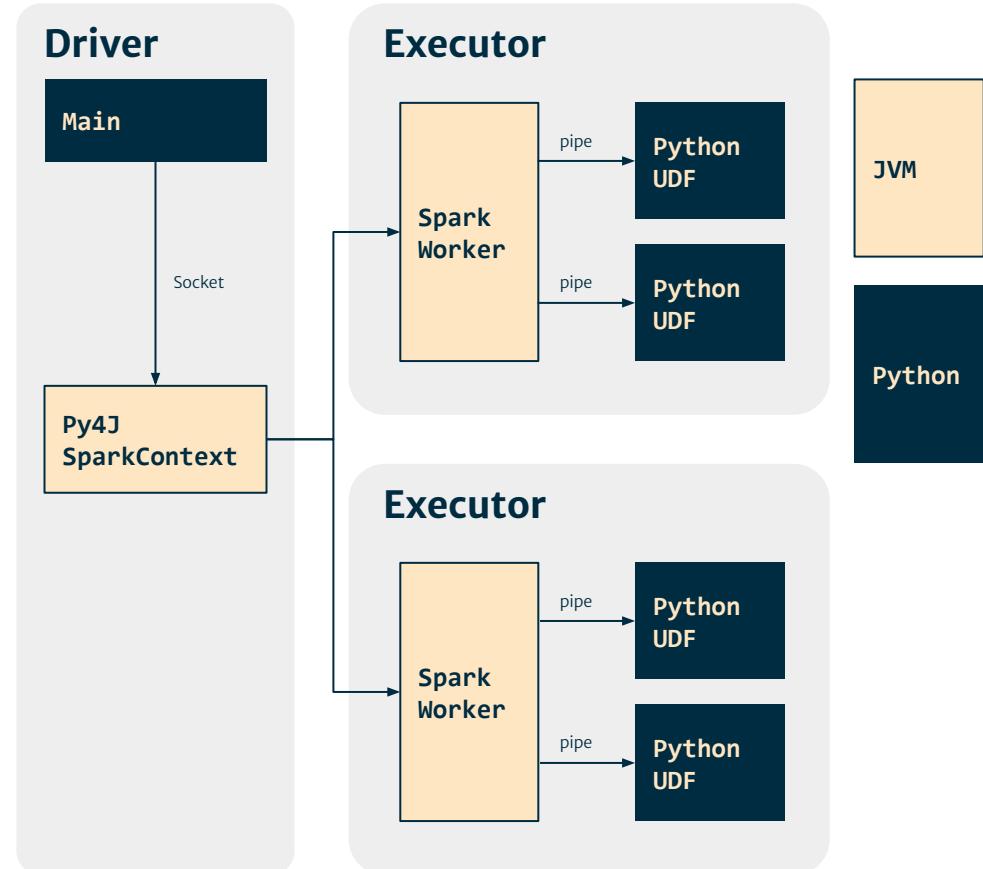
@F.udf('string')
def extract_title_udf(name):
    return name.split()[1]

df.withColumn('title', extract_title_udf(df.Name)).show(4)
```

UDF - Le problème avec Python



- Spark est dans le monde Java
- Les UDF sont en Python
- La donnée doit être envoyé depuis la JVM vers un exécuteur Python
- Grosse perte de temps





UDF Take away

- Les UDFs permettent de créer de nouvelles fonctions sur colonne
- Il faut les éviter tant qu'on peut
 - Spark ne peut pas optimiser les UDFs
- Privilégier les fonctions de Spark
- En Python, les performances sont extrêmement dégradés
- Ne faites pas d'UDF en Python



User Defined Aggregate Functions



- Comme pour les UDF mais pour des agrégations
 - `groupBy().agg(udaf())`
- N'existe pas en Python
- Exemple : moyenne géométrique

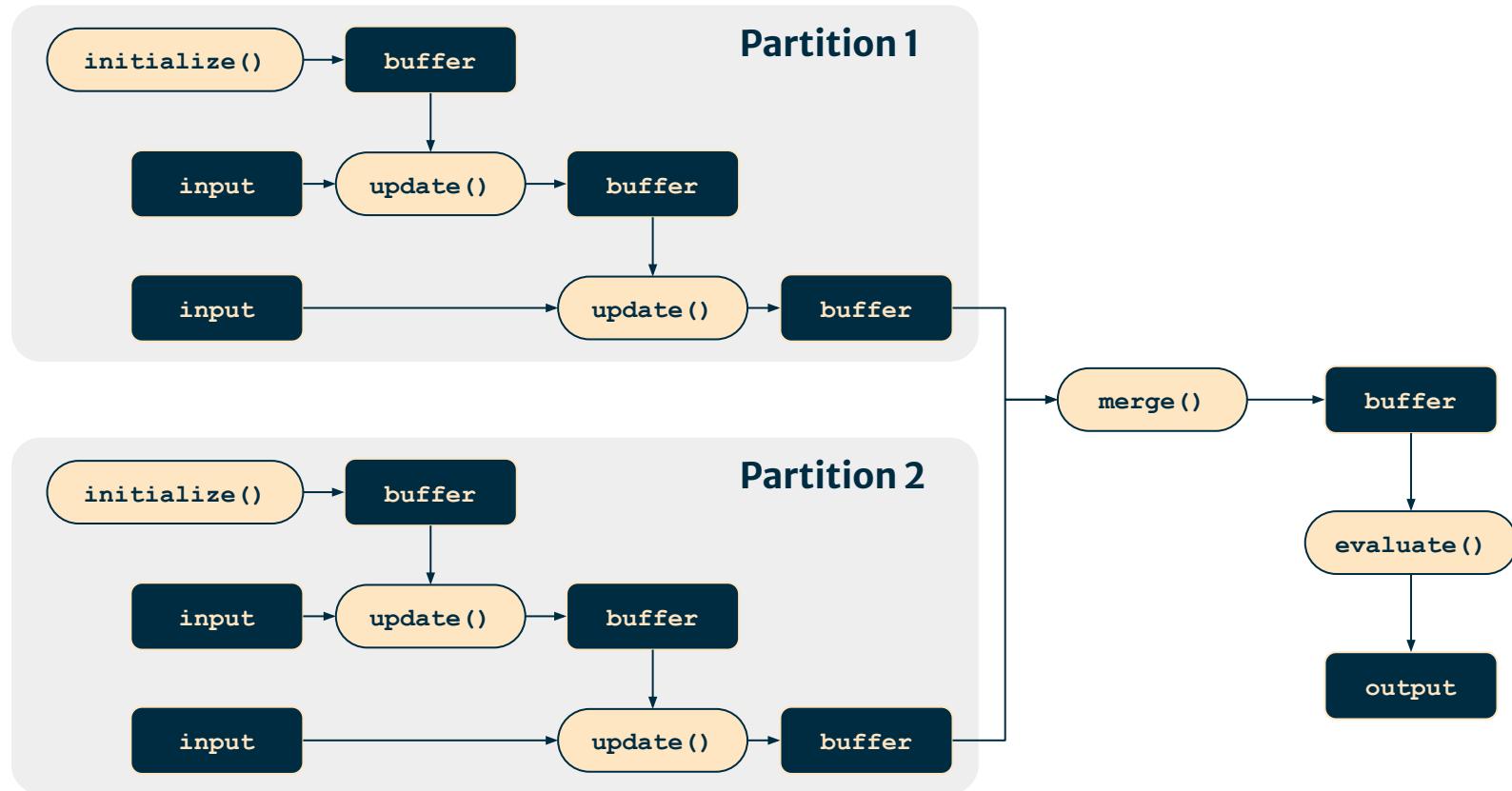
UDAF - Implémentation



- Interface **UserDefinedAggregationFunction** à implémenter
- 8 méthodes
 - **inputSchema** : StructType qui représente le schéma de l'entrée
 - **bufferSchema** : StructType qui représente le schéma des résultats intermédiaires
 - **dataType** : Type de la valeur à retourner
 - **deterministic** : Boolean qui indique si le résultat est déterministe ou pas
 - **initialize** : Valeur de base des buffers
 - **update** : Fonction qui permet de mettre à jour un buffer
 - **merge** : Fonction qui permet d'assembler 2 buffers
 - **evaluate** : Fonction qui calcule le résultat final



UDAF - Exécution





Task not serializable error



Task not serializable error

- Quand on implémente une UDF dans une classe, Spark sérialise toute la classe
- Si la classe a un attribut non sérialisable, cela déclenche une erreur
- L'objet SparkContext n'est pas sérialisable et provoque la majorité de ces erreurs

Task not serializable error - Exemple



Reprendons l'exemple de notre UDF

```
import org.apache.spark.sql.expressions.UserDefinedFunction
import org.apache.spark.sql.functions._
import org.apache.spark.sql.{DataFrame, SparkSession}

case class AddTitle(spark: SparkSession) {

    private def extractTitle(name: String): String = name.split(" ")(1)
    private val extractTitleUdf: UserDefinedFunction = udf[String, String](extractTitle)

    def addTitleColumn(file: String): DataFrame = {
        spark.read
            .option("header", "true")
            .csv(file)
            .withColumn("title", extractTitleUdf(col("name")))
    }
}
```



Task not serializable error - Exemple

Ici tout fonctionne

```
val file = "..."  
val addTitle = AddTitle(spark)  
  
addTitle.addTitleColumn(file).show(truncate = false)
```

```
+-----+-----+  
|name |title |  
+-----+-----+  
|Allison, Miss. Helen Loraine |Miss.|  
|Andrews, Mr. Thomas Jr |Mr. |  
|Astor, Col. John Jacob |Col. |  
|Carter, Mrs. Ernest Courtenay|Mrs. |  
+-----+-----+
```

Task not serializable error - Exemple



- Maintenant disons qu'on veut logger des informations sur Spark
 - Le nom de l'application
 - L'ID de l'application
 - Le master
- Ces informations sont disponible dans le SparkContext
- Le SparkContext se récupère du SparkSession

Task not serializable error - Exemple



```
case class AddTitle(spark: SparkSession) {  
  
    val Logger: Logger = LoggerFactory.getLogger(this.getClass)  
    val sc: SparkContext = spark.sparkContext  
  
    Logger.info(s"App Id: ${sc.applicationId}")  
    Logger.info(s"App Name: ${sc.appName}")  
    Logger.info(s"Master: ${sc.master}")  
  
    private def extractTitle(name: String): String = name.split(" ")(1)  
  
    private val extractTitleUdf: UserDefinedFunction = udf[String, String](extractTitle)  
  
    def addTitleColumn(file: String): DataFrame = {  
        spark.read  
            .option("header", "true")  
            .option("delimiter", ";")  
            .csv(file)  
            .withColumn("title", extractTitleUdf(col("name")))  
    }  
}
```



Task not serializable error - Exemple

On lance le code et...

```
val file = "..."  
val addTitle = AddTitle(spark)  
  
addTitle.addTitleColumn(file).show(truncate = false)
```

Task not serializable error - Exemple



```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
  at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:403)
  at org.apache.spark.util.ClosureCleaner$.org$apache$spark$util$ClosureCleaner$$clean(ClosureCleaner.scala:393)
  [...]
Caused by: java.io.NotSerializableException: org.apache.spark.SparkContext
Serialization stack:
  - object not serializable (class: org.apache.spark.SparkContext, value: org.apache.spark.SparkContext@24c7b944)
  - field (class: fr.hymaia.training.sparkfordev.udf.AddTitle, name: sc, type: class org.apache.spark.SparkContext)
  - object (class fr.hymaia.training.sparkfordev.udf.AddTitle, AddTitle(org.apache.spark.sql.SparkSession@7eb006bd))
  - field (class: fr.hymaia.training.sparkfordev.udf.AddTitle$$anonfun$1, name: $outer, type: class
fr.hymaia.training.sparkfordev.udf.AddTitle)
  - object (class fr.hymaia.training.sparkfordev.udf.AddTitle$$anonfun$1, <function1>)
  - element of array (index: 3)
  - array (class [Ljava.lang.Object;, size 4)
  - field (class: org.apache.spark.sql.execution.WholeStageCodegenExec$$anonfun$11, name: references$1, type: class [Ljava.lang.Object;)
  - object (class org.apache.spark.sql.execution.WholeStageCodegenExec$$anonfun$11, <function2>)
at org.apache.spark.serializer.SerializationDebugger$.improveException(SerializationDebugger.scala:40)
at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala:46)
at org.apache.spark.serializer.JavaSerializerInstance.serialize(JavaSerializer.scala:100)
at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:400)
... 44 more
```

Task not serializable error - Exemple



```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
  at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:403)
  at org.apache.spark.util.ClosureCleaner$.org$apache$spark$util$ClosureCleaner$$clean(ClosureCleaner.scala:393)
  [...]
Caused by: java.io.NotSerializableException: org.apache.spark.SparkContext
Serialization stack:
  - object not serializable (class: org.apache.spark.SparkContext, value: org.apache.spark.SparkContext@24c7b944)
  - field (class: fr.hymaia.training.sparkfordev.udf.AddTitle, name: sc, type: class org.apache.spark.SparkContext)
  - object (class fr.hymaia.training.sparkfordev.udf.AddTitle, AddTitle(org.apache.spark.sql.SparkSession@7eb006bd))
  - field (class: fr.hymaia.training.sparkfordev.udf.AddTitle$$anonfun$1, name: $outer, type: class
fr.hymaia.training.sparkfordev.udf.AddTitle)
  - object (class fr.hymaia.training.sparkfordev.udf.AddTitle$$anonfun$1, <function1>)
  - element of array (index: 3)
  - array (class [Ljava.lang.Object;, size 4)
  - field (class: org.apache.spark.sql.execution.WholeStageCodegenExec$$anonfun$11, name: references$1, type: class [Ljava.lang.Object;)
  - object (class org.apache.spark.sql.execution.WholeStageCodegenExec$$anonfun$11, <function2>)
at org.apache.spark.serializer.SerializationDebugger$.improveException(SerializationDebugger.scala:40)
at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala:46)
at org.apache.spark.serializer.JavaSerializerInstance.serialize(JavaSerializer.scala:100)
at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:400)
... 44 more
```

Task not serializable error - Exemple



```
case class AddTitle(spark: SparkSession) {  
  
    val Logger: Logger = LoggerFactory.getLogger(this.getClass)  
    val sc: SparkContext = spark.sparkContext  
  
    Logger.info(s"App Id: ${sc.applicationId}")  
    Logger.info(s"App Name: ${sc.appName}")  
    Logger.info(s"Master: ${sc.master}")  
  
    private def extractTitle(name: String): String = name.split(" ")(1)  
  
    private val extractTitleUdf: UserDefinedFunction = udf[String, String](extractTitle)  
  
    def addTitleColumn(file: String): DataFrame = {  
        spark.read  
            .option("header", "true")  
            .option("delimiter", ";")  
            .csv(file)  
            .withColumn("title", extractTitleUdf(col("name")))  
    }  
}
```

Task not serializable error - Exemple



- L'objet SparkContext est l'origine de l'erreur
- Il faut trouver un moyen de ne pas avoir besoin de sérialiser l'objet

Task not serializable error – Solution



```
...
import org.apache.spark.SparkConf

case class AddTitle(spark: SparkSession, conf: SparkConf) {
    val Logger: Logger = LoggerFactory.getLogger(this.getClass)

    Logger.info(s"App Id: ${conf.getAppId}")
    Logger.info(s"App Name: ${conf.get("spark.app.name")}")
    Logger.info(s"Master: ${conf.get("spark.master")}")

    private def extractTitle(name: String): String = name.split(" ")(1)
    private val extractTitleUdf: UserDefinedFunction = udf[String, String](extractTitle)

    def addTitleColumn(file: String): DataFrame = {
        spark.read
            .option("header", "true")
            .option("delimiter", ";")
            .csv(file)
            .withColumn("title", extractTitleUdf(col("name")))
    }
}
```



Task not serializable error – Solution

On lance le code et...

```
val file = "..."  
val addTitle = AddTitle(spark)  
  
addTitle.addTitleColumn(file).show(truncate = false)
```

Task not serializable error – Solution



```
INFO AddTitle: App Id: local-1548691949899  
INFO AddTitle: App Name: Advanced Spark  
INFO AddTitle: Master: local[*]
```

name	title
Allison, Miss. Helen Loraine	Miss.
Andrews, Mr. Thomas Jr	Mr.
Astor, Col. John Jacob	Col.
Carter, Mrs. Ernest Courtenay	Mrs.



Take Away

- Les objets non sérialisables les plus communs
 - SparkContext
 - Les class personnalisés (privilégiez les case class)
 - Les class des bibliothèques externes
- SparkSession et SparkConf sont sérialisables
 - Le SparkContext est annoté @transient
 - En Scala, cela signifie qu'il ne doit pas être sérialisé
 - Ce champ devient inaccessible
- Créez vos UDF dans des **case class** ou des **object**



Window functions



Window functions

- Fonctionne comme en SQL
- Permet d'utiliser des fonctions d'agrégations sans agréger un Dataframe



Window functions - Exemple

Quel est le prix moyen par classe ?

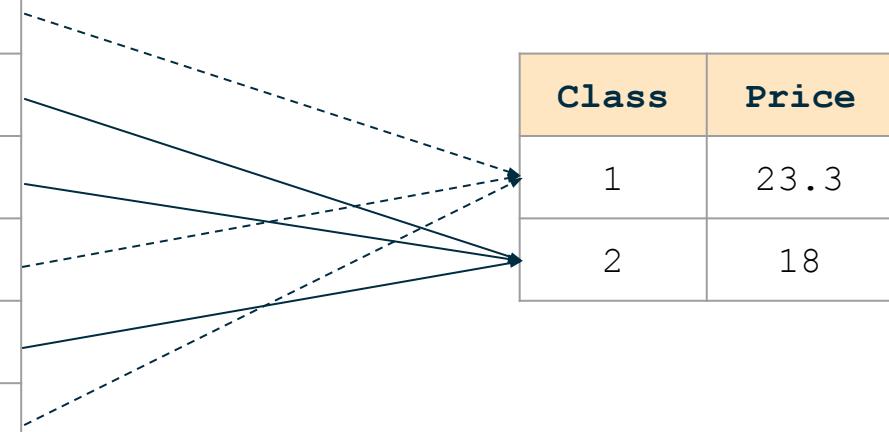
ID	Class	Price
001	1	9
002	2	24
003	2	24
004	1	39
005	2	6
006	1	21



Window functions - Exemple

- Quel est le prix moyen par classe ?
 - On peut facilement résoudre ça avec une agrégation

ID	Class	Price
001	1	9
002	2	24
003	2	24
004	1	39
005	2	6
006	1	21





Window functions - Exemple

- Pour chaque article, quelle est la différence entre son prix et la moyenne de prix dans sa classe ?

ID	Class	Price
001	1	9
002	2	24
003	2	24
004	1	39
005	2	6
006	1	21

Window functions - Exemple



- Pour chaque article, quelle est la différence entre son prix et la moyenne de prix dans sa classe ?

ID	Class	Price
001	1	9
002	2	24
003	2	24
004	1	39
005	2	6
006	1	21

Avg class 2
 $(24+24+6)/3=18$



ID	Class	Price	Diff
001	1	9	
002	2	24	
003	2	24	6
004	1	39	
005	2	6	
006	1	21	



Window functions

- 3 types de Window functions
 - Fonction de classement : `rank`, `dense_rank`, `percent_rank`,
`ntile`, `row_number`...
 - Fonction d'analytique : `cumeDist`, `lag`, `lead`...
 - Fonction d'agrégation : Toutes celles qu'on a déjà vu
- Les Windows functions peuvent être utilisées en SQL

Window functions - Spécifications



- Pour créer une Window function il faut créer une `WindowSpec`
- Une `WindowSpec` définit les lignes qui seront dans notre Window
- Il y a trois spécifications que l'on peut remplir :
 - `partitionBy` : Clé de partitionnement, comme un `groupBy`
 - `orderBy` : Utile pour les fonctions de classement
 - `rowsBetween` / `rangeBetween` : Définit la taille de la fenêtre par rapport à une ligne



Window functions - Over

- L'objet colonne possède une méthode **over()**
- Cette méthode prend en paramètre une **windowSpec**
- Elle applique une **Window** à une fonction sur colonne

```
// rank compute given a WindowSpec  
rank().over(windowSpec)
```

Window functions - Exemple 1 : partitionBy



J'ajoute à chaque ligne la moyenne de prix de sa classe

```
val window = Window.partitionBy("class")  
  
val avgPriceCol: Column = avg("price").over(window)  
  
val dfWithAvg: DataFrame = df  
  .withColumn("avg_class_price", avgPriceCol)  
  
dfWithAvg.show()
```

+-----+-----+-----+-----+	id class price avg_class_price	+-----+-----+-----+-----+
001 1 9 23.0		
004 1 39 23.0		
006 1 21 23.0		
002 2 24 18.0		
003 2 24 18.0		
005 2 6 18.0		

Window functions - Exemple 2: orderBy



Pour chaque classe, je trie les lignes par prix et ajoute leur rang

```
val window = Window
    .partitionBy("class")
    .orderBy(col("price").desc)

val dfWithRank = df
    .withColumn("price_rank", rank().over(window))

dfWithRank.show()
```

+-----+-----+-----+-----+				
id class price price_rank				
+-----+-----+-----+-----+				
004 1 39 1				
006 1 21 2				
001 1 9 3				
002 2 24 1				
003 2 24 1				
005 2 6 3				
+-----+-----+-----+-----+				

Window functions - Exemple 3 : dense_rank



Pour chaque classe, je trie les lignes par prix et ajoute leur rang

```
val window = Window
    .partitionBy("class")
    .orderBy(col("price").desc)

val dfWithRank = df
    .withColumn("price_rank", dense_rank().over(window))

dfWithRank.show()
```

+-----+-----+-----+-----+	id class price price_rank	+-----+-----+-----+-----+
004 1 39 1		
006 1 21 2		
001 1 9 3		
002 2 24 1		
003 2 24 1		
005 2 6 2		



Window functions - Exemple 4 : rangeBetween

Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+ <th> id class price count <th>+-----+-----+-----+</th></th>	id class price count <th>+-----+-----+-----+</th>	+-----+-----+-----+
005 2 6		
001 1 9		
006 1 21		
002 2 24		
003 2 24		
004 1 39		

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+	
id class price <th> count </th>	count	
+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+	
005 2 6	1	
001 1 9	-4; 16	
006 1 21	[
002 2 24	16]	
003 2 24	[
004 1 39	16]	
	+-----+-----+-----+ <th></th>	

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+
id class price	count
+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+
005 2 6	1
001 1 9	1
006 1 21	1
002 2 24	1
003 2 24	1
004 1 39	1
+-----+-----+-----+	+-----+-----+-----+

[-4; 16 [

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+
id class price <th> count </th>	count
+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+
005 2 6	1
001 1 9	
006 1 21	
002 2 24	
003 2 24	
004 1 39	
	+-----+-----+-----+

[-4; 16 [



Window functions - Exemple 4 : rangeBetween

Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+				
id class price count				
+-----+-----+-----+				
005 2 6 1				
001 1 9 1	[-1; 19 [
006 1 21 1				
002 2 24 1				
003 2 24 1				
004 1 39 1				
+-----+-----+-----+				



Window functions - Exemple 4 : rangeBetween

Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+
id class price <th> count </th>	count
+-----+-----+-----+ <th>+-----+-----+-----+</th>	+-----+-----+-----+
005 2 6	1
001 1 9	
006 1 21	
002 2 24	
003 2 24	
004 1 39	
+-----+-----+-----+	+-----+-----+-----+

[-1; 19 [

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
  .orderBy("price")
  .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
  .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+				+-----+
id class price count				+-----+
+-----+-----+-----+				+-----+
005 2 6 1				
001 1 9 1				[-1; 19 [
006 1 21 1				
002 2 24 1				
003 2 24 1				
004 1 39 1				
+-----+-----+-----+				+-----+

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+	id class price count	+-----+-----+-----+	
005 2 6 1			
001 1 9 1			
006 1 21 1			
002 2 24 1			
003 2 24 1			
004 1 39 1			
+-----+-----+-----+		[11; 31 [

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+	id class price count	+-----+-----+-----+
005 2 6 1		
001 1 9 1		
 006 1 21 1 		
002 2 24 1		
003 2 24 1		
004 1 39 1		

[11; 31 [



Window functions - Exemple 4 : rangeBetween

Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+	id class price count	+-----+-----+-----+
005 2 6 1		
001 1 9 1		
006 1 21 2	[11; 31 [
002 2 24 1		
003 2 24 1		
004 1 39 1		
+-----+-----+-----+		+-----+-----+-----+

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+	id class price count	+-----+-----+-----+
005 2 6 1		
001 1 9 1		
006 1 21 2		
 002 2 24 2 	[14; 34 [
003 2 24 1		
004 1 39 1		
+-----+-----+-----+		+-----+-----+-----+

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+	id class price count	+-----+-----+-----+
005 2 6 1		
001 1 9 1		
006 1 21 2		
002 2 24 2		
003 2 24 2	[14; 34 [
004 1 39 1		



Window functions - Exemple 4 : rangeBetween

Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

+-----+-----+-----+	id class price count	+-----+-----+-----+
005 2 6 1		
001 1 9 1		
006 1 21 2		
002 2 24 2		
003 2 24 2		
004 1 39 0		[29; 49 [
+-----+-----+-----+		

Window functions - Exemple 4 : rangeBetween



Pour chaque produit, je veux savoir combien de produits coûtent entre 10€ de moins et de plus

```
val windowRange = Window
    .orderBy("price")
    .rangeBetween(-10, 10)

// '- 1' because count() includes the current
// row and we don't want to count it
val countCol = count("*").over(windowRange) - 1

val dfWithCount: DataFrame = df
    .withColumn("count", countCol)

dfWithCount.show()
```

id	class	price	count
005	2	6	1
001	1	9	1
006	1	21	2
002	2	24	2
003	2	24	2
004	1	39	0

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
  .orderBy("year", "quarter")
  .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
  .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	
2018	Q2	21	
2018	Q3	32	
2018	Q4	45	
2019	Q1	13	
2019	Q2	64	
2019	Q3	23	
2019	Q4	13	

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	
2018	Q3	32	
2018	Q4	45	
2019	Q1	13	
2019	Q2	64	
2019	Q3	23	
2019	Q4	13	

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	
2018	Q4	45	
2019	Q1	13	
2019	Q2	64	
2019	Q3	23	
2019	Q4	13	

12+21

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
  .orderBy("year", "quarter")
  .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
  .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	
2019	Q1	13	
2019	Q2	64	
2019	Q3	23	
2019	Q4	13	

12+21+32

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	110
2019	Q1	13	
2019	Q2	64	
2019	Q3	23	
2019	Q4	13	

12+21+32+45

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	110
2019	Q1	13	111
2019	Q2	64	
2019	Q3	23	
2019	Q4	13	

21+32+45+13

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	110
2019	Q1	13	111
2019	Q2	64	154
2019	Q3	23	
2019	Q4	13	

32+45+13+64

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
  .orderBy("year", "quarter")
  .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
  .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	110
2019	Q1	13	111
2019	Q2	64	154
2019	Q3	23	145
2019	Q4	13	

45+13+64+23

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	110
2019	Q1	13	111
2019	Q2	64	154
2019	Q3	23	145
2019	Q4	13	113

13+64+23+13

Window functions - Exemple 5 : rowsBetween



Pour chaque quarter, je veux connaitre les dépense totales de l'année passée

```
val windowRows = Window
    .orderBy("year", "quarter")
    .rowsBetween(-3, 0)

val sumColumn = sum("expense").over(windowRows)

val dfWithRows = df
    .withColumn("ly_exp", sumColumn)

dfWithRows.show()
```

year	quarter	expense	ly_exp
2018	Q1	12	12
2018	Q2	21	33
2018	Q3	32	65
2018	Q4	45	110
2019	Q1	13	111
2019	Q2	64	154
2019	Q3	23	145
2019	Q4	13	113

Window functions - Exemple initial



Pour chaque article, quelle est la différence entre son prix et la moyenne de prix dans sa classe ?

```
val window = Window  
  .partitionBy("class")  
  
val avgCol = avg(col("price")).over(window)  
val diffCol = col("price") - avgCol  
  
val dfWithDiff: DataFrame = df  
  .withColumn("dif_price_avg", diffCol)  
  
dfWithDiff.show()
```

id	class	price	dif_price_avg
001	1	9	-14.0
004	1	39	16.0
006	1	21	-2.0
002	2	24	6.0
003	2	24	6.0
005	2	6	-12.0



Take away

- Les **UDFs** permettent de créer nos propres fonctions sur colonne
 - Si possible mieux vaut s'en passer
 - En **Scala** c'est assez commun
 - En **Python** il ne faut pas en faire à cause des performances dégradées
- Les **UDAF** permettent de créer nos propres fonctions d'agrégations sur colonne
- Les **Window functions** permettent de calculer des agrégations sans **groupBy()**



Déployer un job Spark



Programme

- Packager son application Spark
- Choisir sa plateforme
- Spark-submit
- Configurer son job Spark



Comment packager une application **Spark** ?

Rappel – Une application Spark



```
from pyspark.sql import SparkSession

def main():
    spark = SparkSession.builder.getOrCreate()
```

Packaging Scala - Les dépendances Spark



- Nécessite 2 dépendances
 - spark-core
 - spark-sql
- Si votre application a besoin d'autres dépendances, il faudra créer un fatjar
- Pour créer un fatjar, il faut utiliser un plugin
 - assembly : Maven et SBT
 - shade : Maven
- Les dépendances Spark doivent être notés provided
 - Les dépendances sont très lourdes (plusieurs centaines de mo)
 - Elles seront installées là où vous lancerez votre application

Packaging Scala - Les outils de build



- Deux outils très populaires pour construire une application Scala
 - Maven
 - SBT
- On crée un jar qui servira à lancer notre application Spark

Packaging Scala - Maven



- Conçu pour Java
- Très mature
- Très populaire
- Syntaxe XML très verbeuse
- Nécessite l'utilisation d'un plugin spécifique pour Scala

Maven™

Packaging Scala - Maven



pom.xml

```
<properties>
    <spark.version>3.3.0</spark.version>
    <scala.dep.version>2.12</scala.dep.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_${scala.dep.version}</artifactId>
        <version>${spark.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_${scala.dep.version}</artifactId>
        <version>${spark.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Packaging Scala – SBT



- Simple Build Tool (renommé en Scala Build Tool)
- Conçu pour Scala
- Syntaxe Scala
- Peu verbeuse
- Assez complexe
- Mature (2008)
- Moins populaire

sbt

Packaging Scala – SBT



```
scalaVersion := "2.12.8"
```

build.sbt

```
val sparkVersion = "2.4.3"
```

```
LibraryDependencies += "org.apache.spark" %% "spark-core" % sparkVersion % "provided"
```

```
LibraryDependencies += "org.apache.spark" %% "spark-sql" % sparkVersion % "provided"
```

Packaging Scala



```
sbt clean package
```

sbt

```
mvn clean package
```

Maven

Packaging Python



- Poetry crée un wheel et un .tar.gz
 - poetry build
- Spark-submit ne prend que des egg ou des zips
- On oublie Poetry

Packaging Python



- **setuptools** permet de créer des eggs (entre autre)
- Il faut créer une structure stricte
 - **setup.py** à la racine du projet
 - **__init__.py** dans chaque package
- Le package est créé dans le dossier dist/ à la racine du projet

```
python setup.py bdist_egg
```



```
from setuptools import setup, find_packages
setup(
    name="HelloWorld",
    version="0.1",
    packages=find_packages(),

    # Project uses reStructuredText, so ensure that the docutils get
    # installed or upgraded on the target machine
    install_requires=["docutils>=0.3"],

    package_data={
        # If any package contains *.txt or *.rst files, include them:
        "": ["*.txt", "*.rst"],
        # And include any *.msg files found in the "hello" package, too:
        "hello": ["*.msg"],
    },

    # metadata to display on PyPI
    author="Me",
    author_email="me@example.com",
    description="This is an Example Package",

    # could also include long_description, download_url, etc.
)
```



Comment bien choisir sa plateforme ?



Les différents supports pour Spark

- Local sur 1 ou plusieurs threads
- Yarn (Hadoop) avec 2 modes
 - client
 - cluster
- Mesos : mais plus personne ne fait ça
- Kubernetes avec 2 modes
 - client
 - cluster



Et le Cloud dans tout ça ?

- **AWS**
 - EMR (VM)
 - Glue (Serverless)
- **GCP**
 - Dataproc (VM ou Serverless)
- **Azure**
 - Databricks (VM)
- **Databricks**

Ce sont tous des services managés Hadoop (Yarn)



- **Hadoop**

- Beaucoup de services managés dans les 3 gros Cloud Provider
- Très mature
- Cher
- Peut manquer de souplesse (serverless)

- **Kubernetes**

- Aucun service managé Spark sur Kubernetes Cloud
- Peu cher
- Très compliqué
- Approche Cloud Native
- Service managé indépendant : Ocean Spark



Qu'est-ce qu'on choisit ?

Le plus simple reste le service managé proposé par votre Cloud Provider en serverless si possible



Spark-submit



Spark-submit

- Binaire fourni par Spark
- Sert à demander des ressources à un resource manager
- Permet de configurer le job Spark

Spark-submit - Scala



```
spark-submit --master yarn --deploy-mode cluster \  
--name "myApp" --class MyClass MyFatJarFile.jar <args>
```

Scala



Spark-submit - Python

```
spark-submit --master yarn --deploy-mode cluster \  
--py-files my_deps.egg \  
my_app.py first_arg second_arg
```

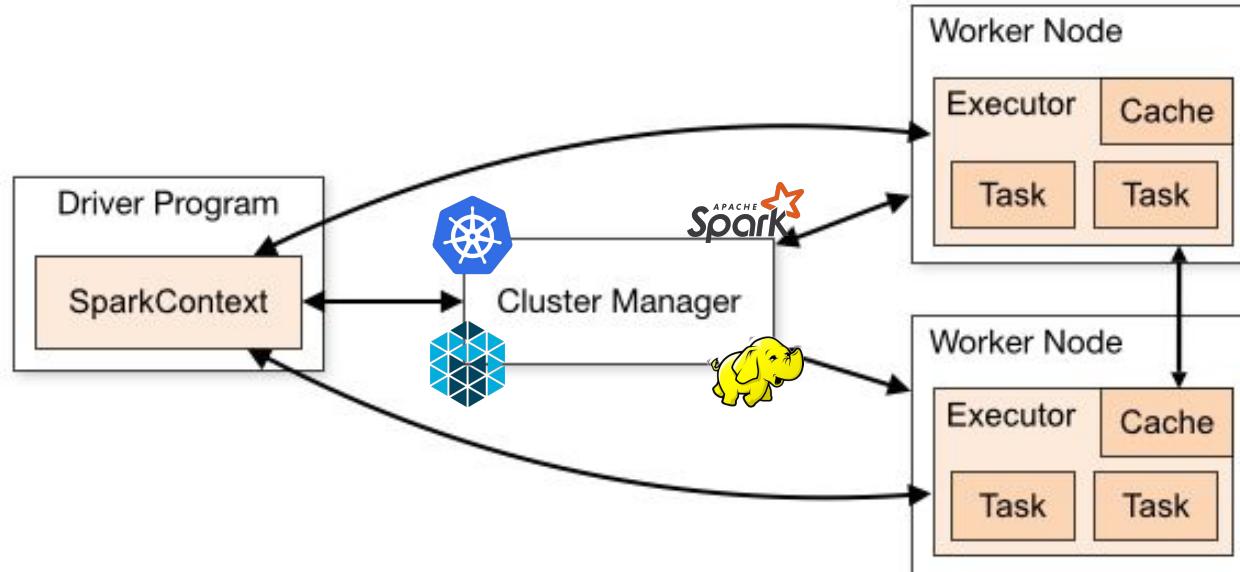
Python

Spark-submit - Deployment mode



Command	Comment
--master local	Local on 1 thread
--master local[n]	Local on n threads
--master local[*]	Local on all available threads
--master spark://<host>:<port>	Spark Standalone
--master yarn --deploy-mode client	Yarn Client
--master yarn --deploy-mode cluster	Yarn Cluster
--master mesos://<host>:<port>	Mesos
--master k8s://<host>:<port> --deploy-mode client	Kubernetes Client
--master k8s://<host>:<port> --deploy-mode cluster	Kubernetes Cluster

Spark-submit - Exécution



Deploy



YARN - Client mode

Edge Node

Worker

Worker

Master

Worker

YARN - Cluster mode

Edge Node

Worker

Worker

Master

Worker

Deploy



YARN - Client mode

Edge Node

Worker

Worker

Master

Worker

YARN - Cluster mode

Edge Node

Worker

Worker

Container

Master

Worker

Deploy



YARN - Client mode

Edge Node

Worker

Worker

Master

Worker

YARN - Cluster mode

Edge Node

Worker

Worker

Container

Worker

Master

Deploy



YARN - Client mode

Edge Node

Worker

Worker

Master

Worker

YARN - Cluster mode

Edge Node

Worker

Worker

Container

Application Master

Driver

Spark
Session

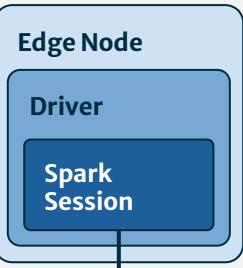
Master

Worker

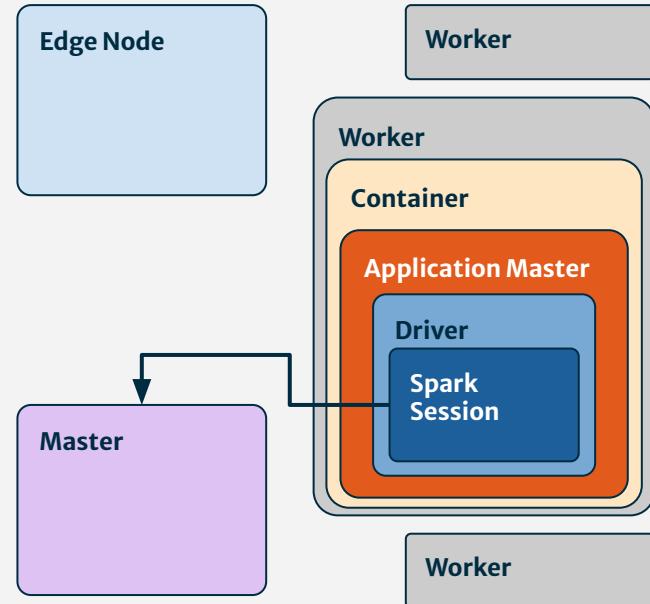
Deploy



YARN - Client mode



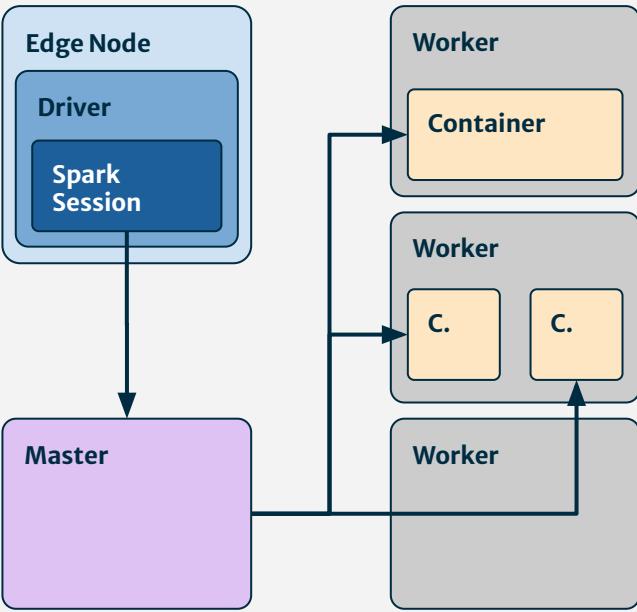
YARN - Cluster mode



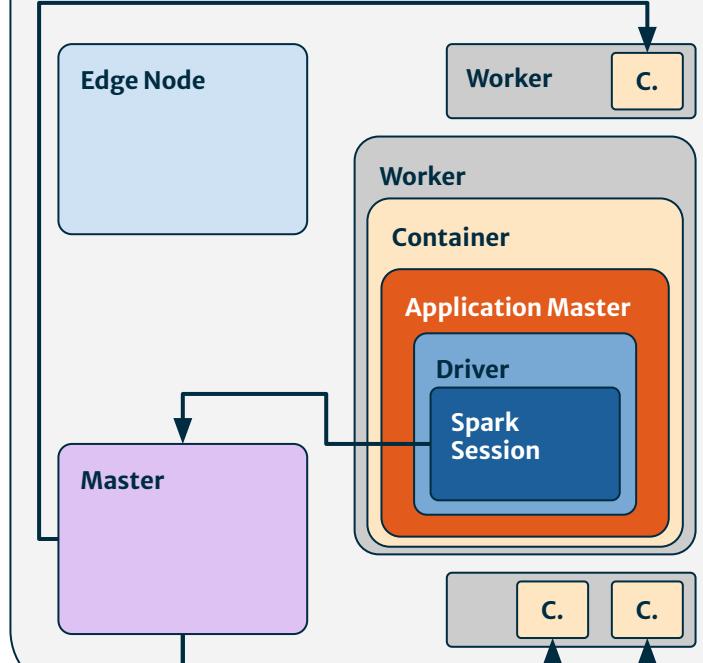
Deploy



YARN - Client mode

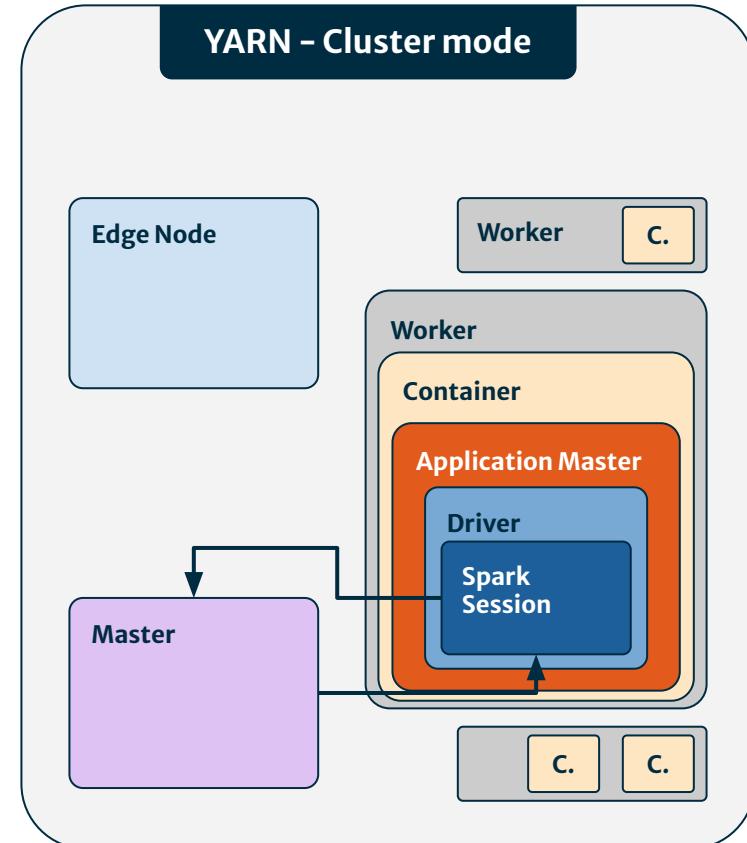
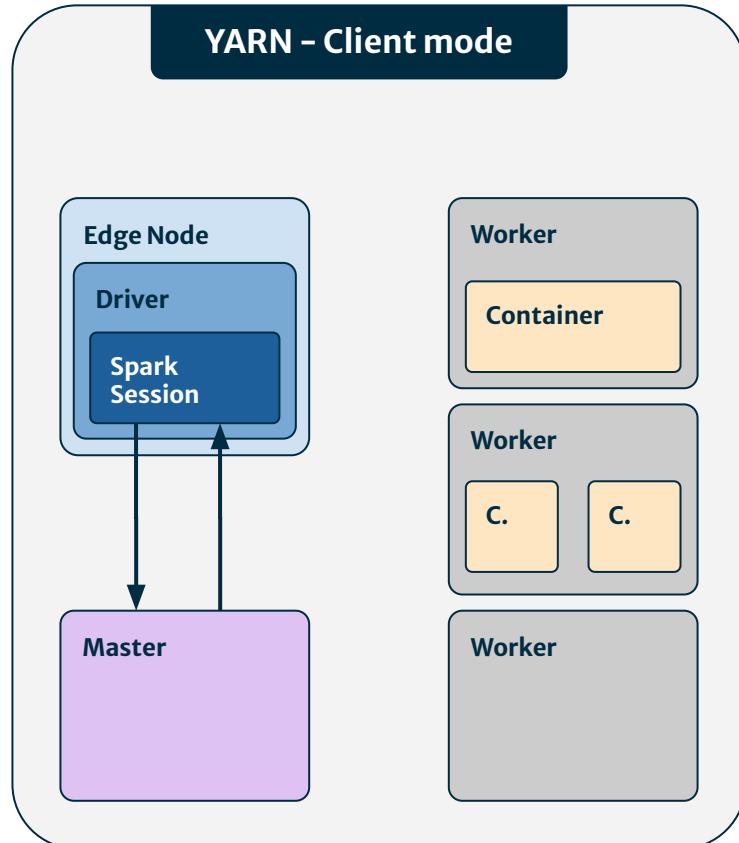


YARN - Cluster mode





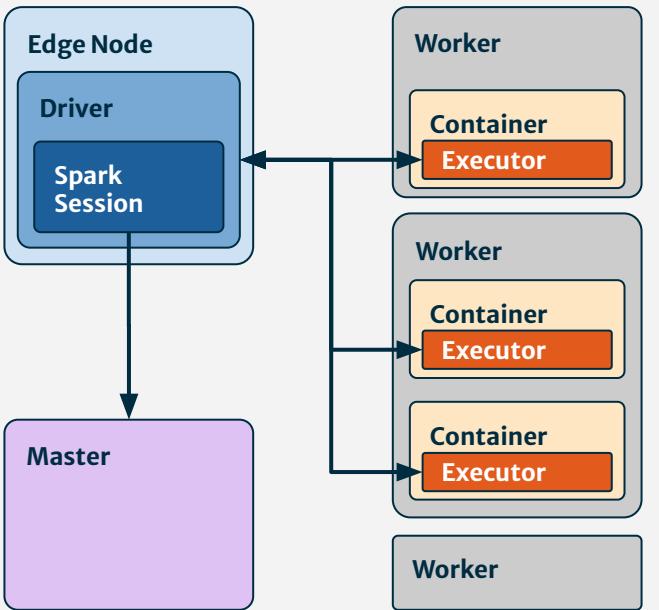
Deploy



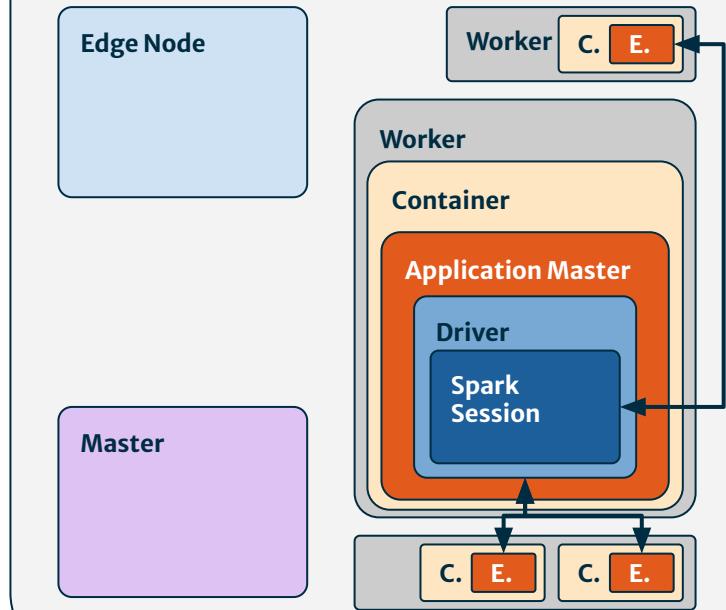
Deploy



YARN - Client mode



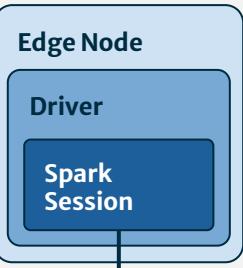
YARN - Cluster mode



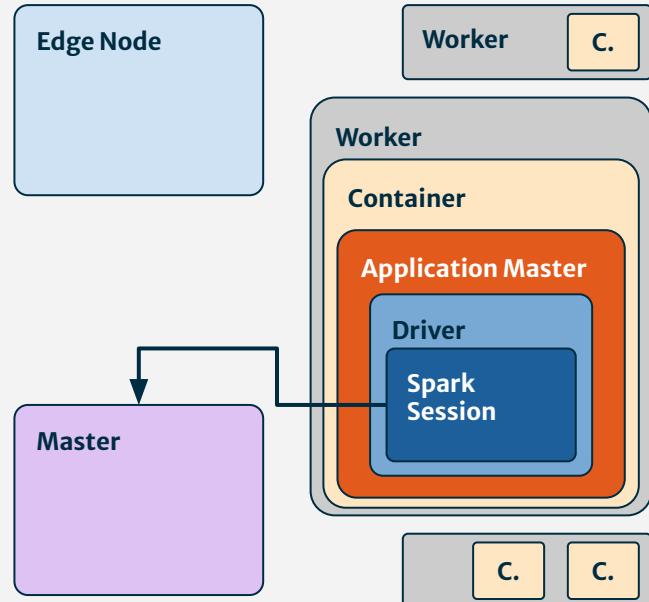
Deploy



YARN - Client mode



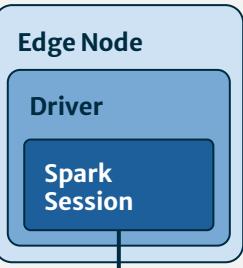
YARN - Cluster mode



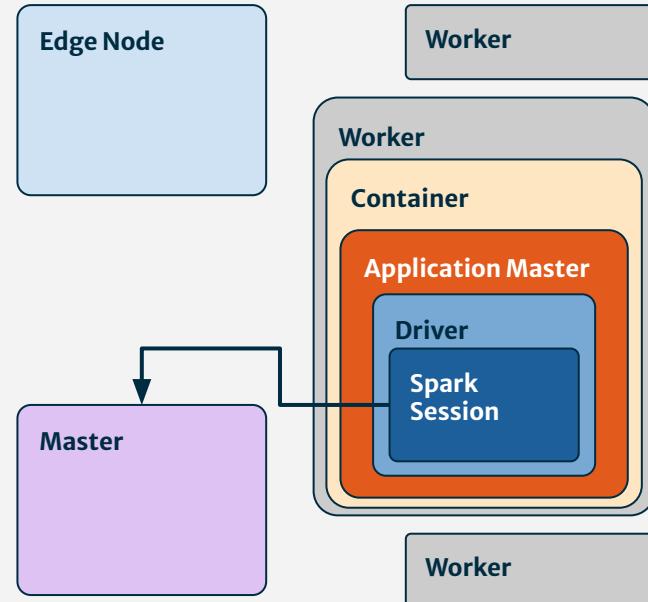
Deploy



YARN - Client mode



YARN - Cluster mode



Deploy



YARN - Client mode

Edge Node

Worker

Worker

Master

Worker

YARN - Cluster mode

Edge Node

Worker

Worker

Master

Worker



Configurer son job Spark



Configurer son job Spark

- Il existe beaucoup de propriétés à configurer dans Spark
- Les propriétés pour dimensionner les ressources
 - spark.executor.memory
 - spark.executor.cores
 - ...
- Les propriétés pour le fonctionnement de Spark
 - spark.sql.shuffle.partitions
 - spark.sql.autoBroadcastJoinThreshold
 - ...
- Les arguments du job Spark
 - Arguments du main
 - Variables d'environnements
- 4 possibilités pour les renseigner

 Lien

Configurer son job Spark – Fichier properties



- Spark en fournit un par défaut déjà entièrement complété
- On peut le surcharger avec notre propre fichier
- Attention celui fournit par Spark est alors ignoré
- Peu pratique

```
./spark-submit --properties-file=dir/my-properties.conf
```

Configurer son job Spark – Spark-submit



- Beaucoup de propriétés ont des flags spéciaux
 - --master
 - --name
 - --deploy-mode
- Pour tout le reste ~~il y a mastercard~~
 - Il y a --conf
- Attention à la commande à rallonge quand on définit trop de propriétés

```
./spark-submit --conf spark.pyspark.python=/alt/path/to/python
```

Configurer son job Spark – Dans le code



- On peut configurer notre SparkSession dans le code
- Des fonctions dédiées existent dans le builder
- `config()` définit n'importe quelle propriété
- Empêche toutes autres modifications des configurations dans le code

```
import org.apache.spark.sql.{DataFrame, SparkSession}

val spark: SparkSession = SparkSession.builder()
    .master("local[*]")
    .appName("my-spark-app")
    .config("spark.ui.port", "5050")
    .getOrCreate()
```

Configurer son job Spark – Ordre de priorité



1. Dans le code
2. Dans le spark-submit
3. Dans le fichier properties personnalisé
4. Dans le fichier properties par défaut de Spark

Configurer son job Spark – Bonnes pratiques



- Rien de moderne (approche Cloud Native)
- Chaque service managé a son propre système
 - Infra as Code
 - Environnement
- Sur Kubernetes vous pouvez installer un SparkOperator
 - Permet de versionner sa configuration dans un yaml
 - Permet d'utiliser les environnements



AWS Glue

AWS Glue – Déployer une configuration Spark



- Script de déploiement
- Code source
- Paramètres de job

AWS Glue – Script de déploiement



- Python ou scalascript
- 5 étapes à réaliser dedans :
 - Créer un SparkSession
 - Créer un GlueContext
 - Créer un job Glue
 - Récupérer les paramètres du job Spark
 - Appeler la fonction d'entrée de son code

AWS Glue – Script de déploiement



```
from ...

if __name__ == '__main__':
    spark = SparkSession.builder.getOrCreate()

    glueContext = GlueContext(spark.sparkContext)

    job = Job(glueContext)

    args = getResolvedOptions(sys.argv, [ "JOB_NAME", "PARAM_1",
"PARAM_2"])
    job.init(args[ 'JOB_NAME' ], args)

    main(spark, args)

job.commit()
```



- **Formats acceptés :**
 - jar
 - egg
 - zip
 - wheel
 - tar.gz
- On privilégiera jar, wheel ou tar.gz
- **Scala :**
 - mvn package
- **Python :**
 - poetry build

AWS Glue – Paramètres de job



- Combien de machines
- Quelle taille ?
- Python ou Scala ?
- Dépendances
- Activation des métriques Spark
- Argument de job Spark
- ...

AWS Glue – Paramètres de job



- Tout peut / doit se faire en Infra as Code
- L'IaC n'exécute aucun job Spark
- Elle déploie un template de job Spark à exécuter à volonté

Script

Job details

Runs

Schedules

Version Control

IAM Role

Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.

glue_role

**Type**

The type of ETL job. This is set automatically based on the types of data sources you have selected.

Spark

**Glue version** [Info](#)

Glue 3.0 - Supports spark 3.1, Scala 2, Python 3

**Language**

Python 3

**Worker type**

Set the type of predefined worker that is allowed when a job runs.

G 1X

(4vCPU and 16GB RAM)

 **Automatically scale the number of workers**

AWS Glue will optimize costs and resource usage by dynamically scaling the number of workers up and down throughout the job run. Requires Glue 3.0 or later.

Requested number of workers

The number of workers you want AWS Glue to allocate to this job.

3





Take away

- En **Scala** : Maven ou SBT pour construire son application
- En **Python** : setuptools pour spark-submit, poetry pour AWS Glue
- Les fatjars sont très pratiques pour les dépendances
- Aujourd’hui on a principalement **2 choix** pour déployer un job Spark
 - **Yarn** (Hadoop) avec les services managés
 - **Kubernetes**, un peu plus manuel
- Faites attention aux configurations Spark dans le code
 - Master en local[*] et vous êtes foutus
- Déployer un job Glue n’exécute pas de job Spark