

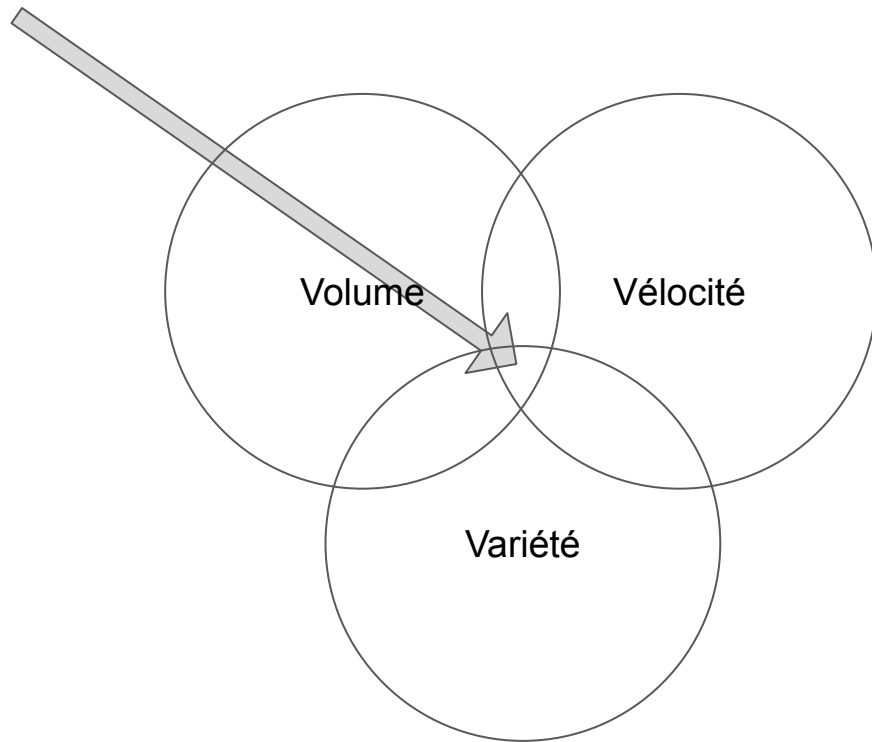
Introduction

| Introduction

Nous générons 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

- ▼ Presques 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

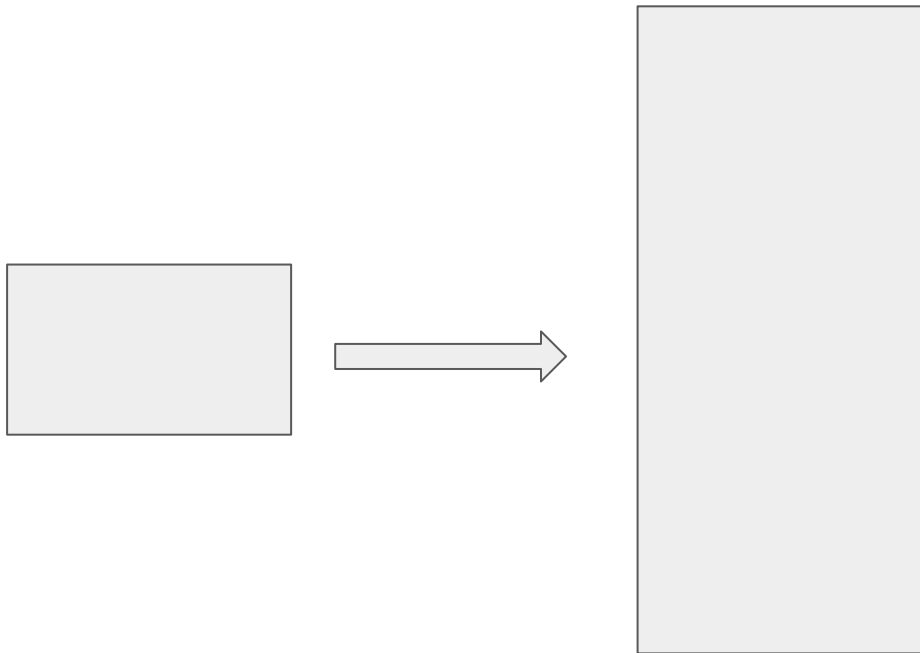
- ▼ Donnée Structurée
 - ▼ BDD
- ▼ Donnée Non Structurée
 - ▼ Page Web
 - ▼ Log
 - ▼ Image
 - ▼ Vidéo

| Le problème : Vitesse

- ▼ Fréquence de mise à jour
- ▼ Temps Réel

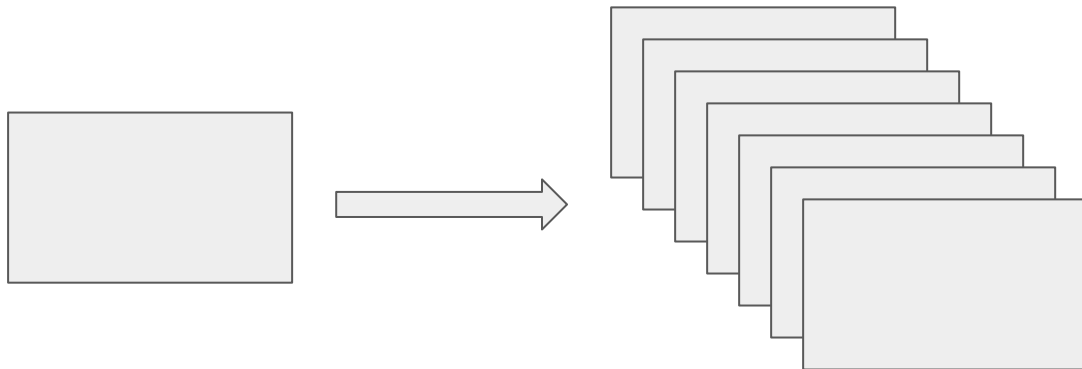
| Approche Vertical

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



| Approche Horizontal

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



| Problème de la donnée

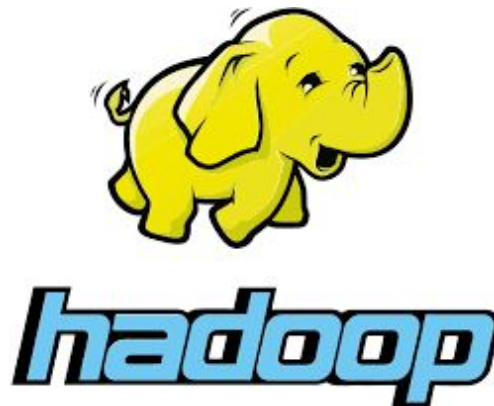
- ▼ Traditionnellement centralisé
- ▼ Transfert de donnée pour les traitements
- ▼ Bande passante réseau limité

| Le besoin

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

| Naissance de Hadoop

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



I 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
- ▼ Différent 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 écosystème

- ▼ riche
- ▼ open source
- ▼ grandissant
 - ▼ Spark (2014)
 - ▼ Kafka (2012)

L'écosystème Hadoop

HDFS - Hive - YARN - Oozie

HDFS

Hadoop Distributed File System

| HDFS

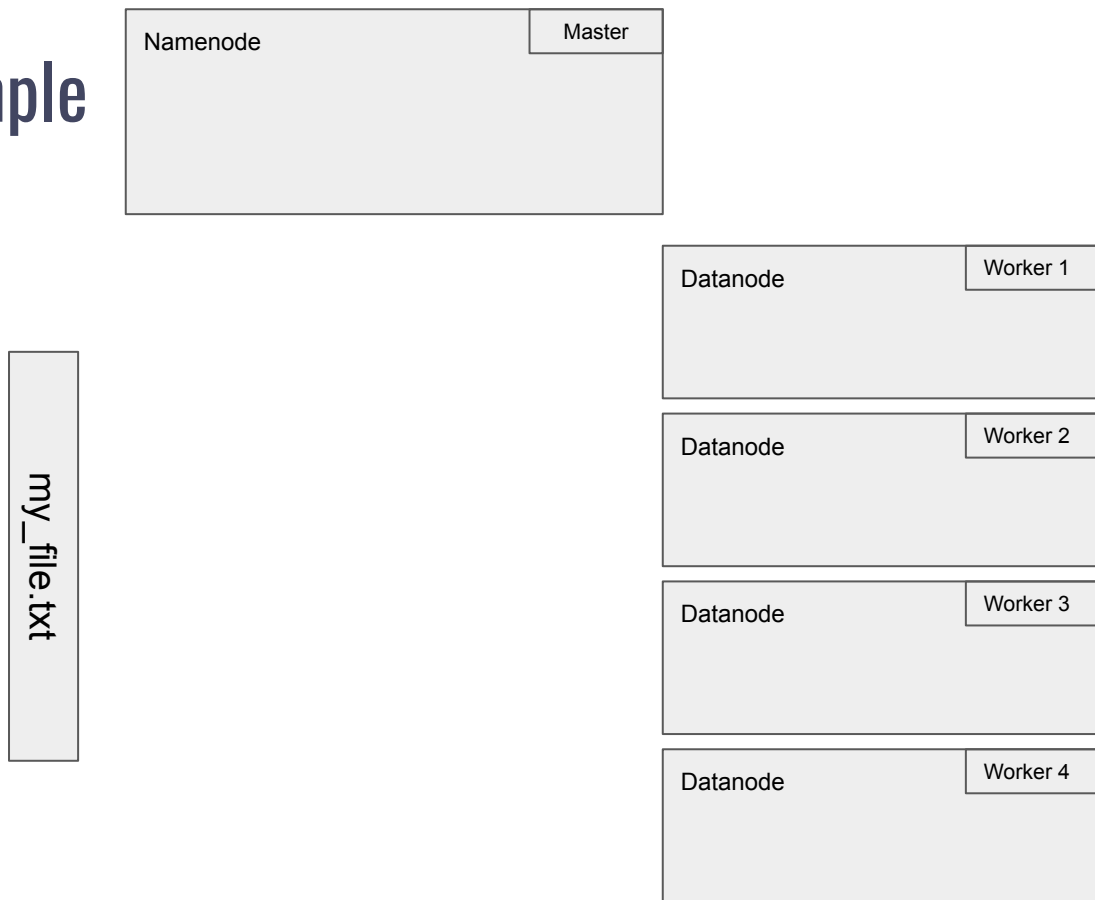
- ▼ Comme un système de fichier classique
- ▼ distribué
- ▼ haute disponibilité
- ▼ résilience



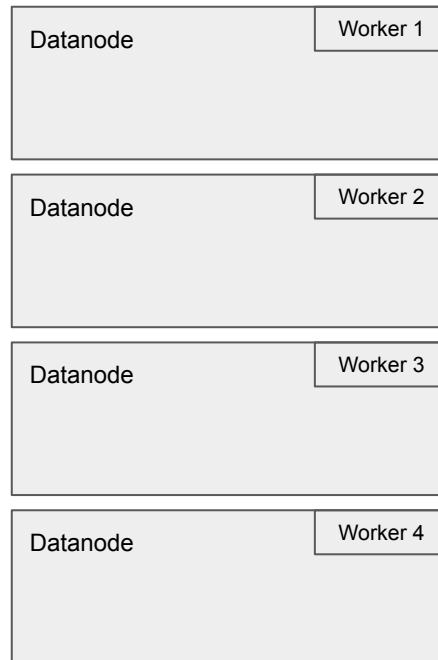
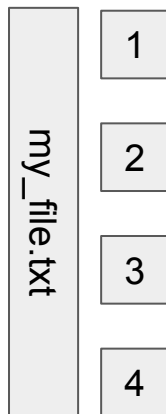
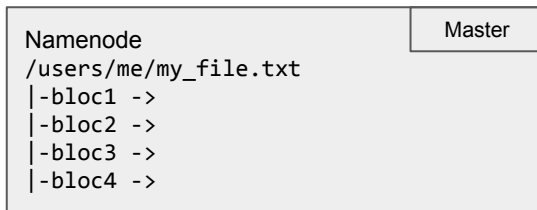
| HDFS utilisation et limitation

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

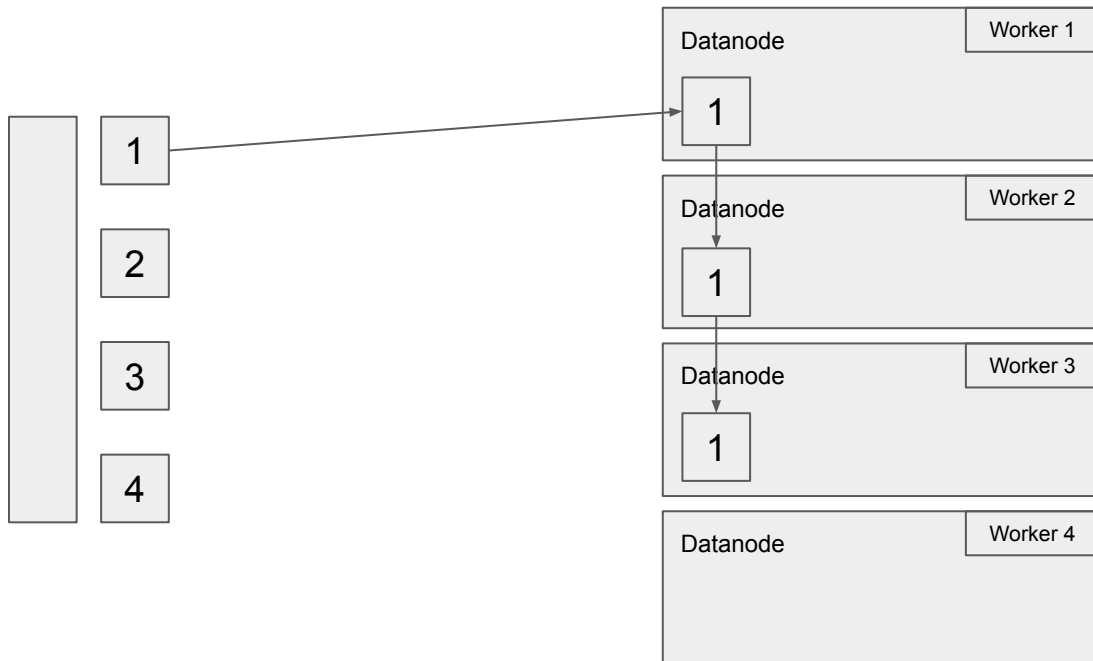
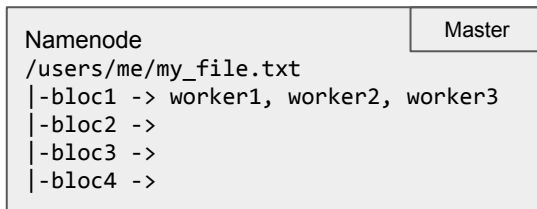
HDFS exemple



HDFS exemple



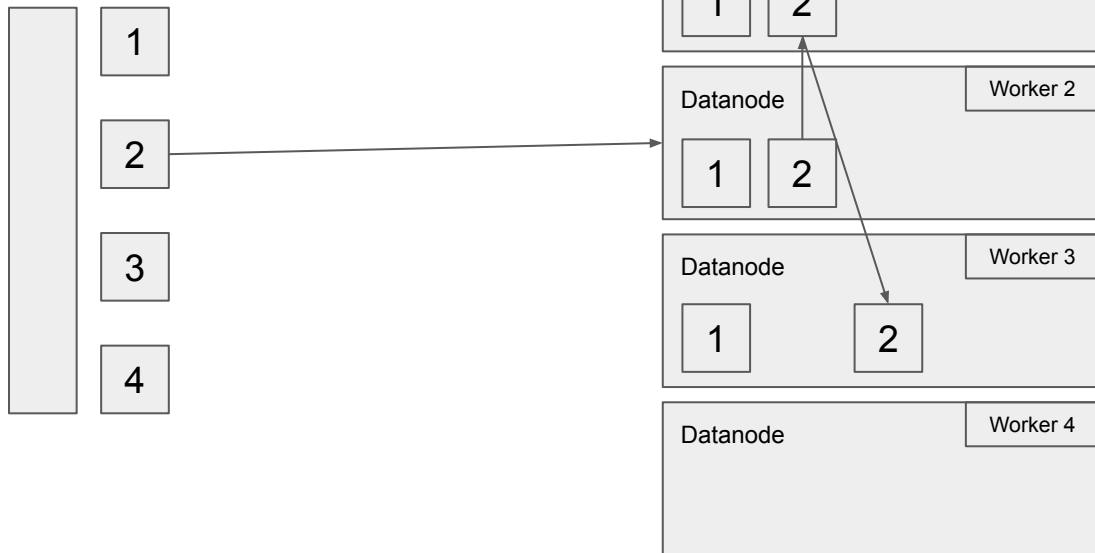
HDFS exemple



HDFS exemple

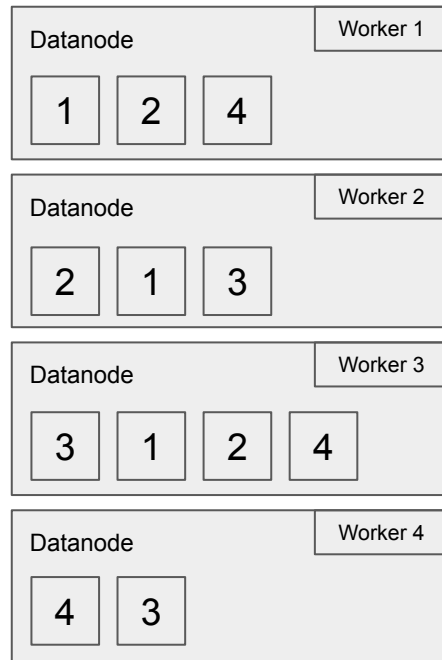
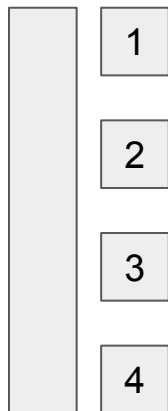
```
Namenode
/users/me/my_file.txt
|-bloc1 -> worker1, worker2, worker3
|-bloc2 -> worker2, worker1, worker3
|-bloc3 ->
|-bloc4 ->
```

Master

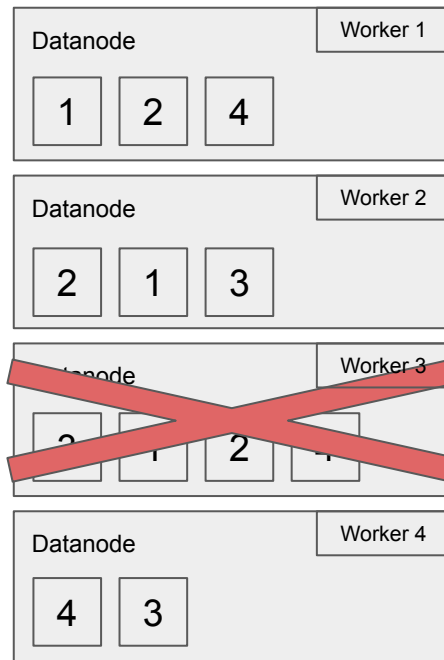
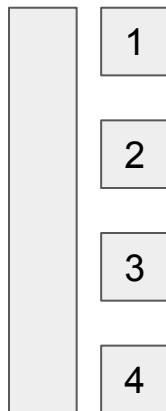
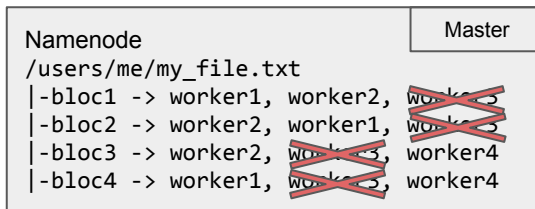


HDFS exemple

Namenode	Master
/users/me/my_file.txt	
-bloc1 -> worker1, worker2, worker3	
-bloc2 -> worker2, worker1, worker3	
-bloc3 -> worker2, worker3, worker4	
-bloc4 -> worker1, worker3, worker4	



HDFS exemple



Hive

| Hive

- ▼ Vue SQL sur les fichiers du HDFS
- ▼ Data Warehouse
- ▼ HiveQL \approx SQL
- ▼ JDBC



| Hive

- ▼ Génère du code spark ou mapreduce
- ▼ Donnée Structurée (Parquet, Avro, ...)
- ▼ Pas besoin de programmer (SQL)



| Hive vs RDBMS

	RDBMS	Hive
Langage de requête	SQL	SQL
Update	Oui	Non
Latence	Faible	Élevé
Scalable	Difficile	Facile
Coût	Élevé	Faible

YARN

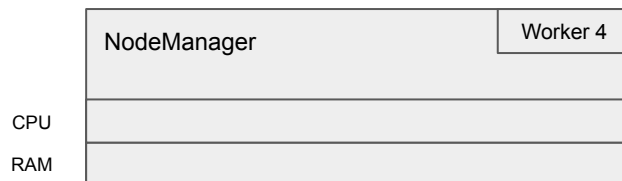
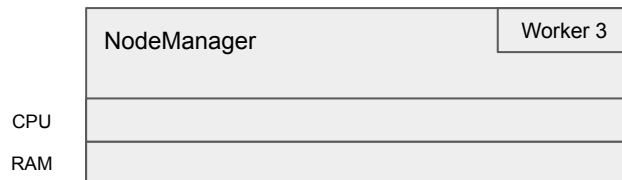
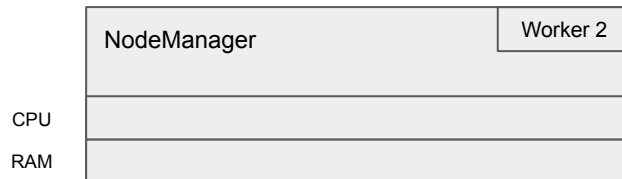
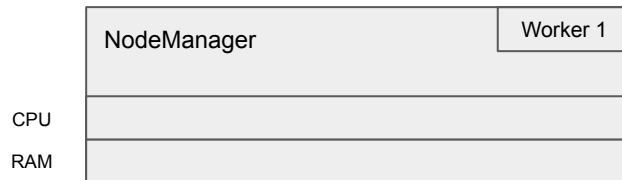
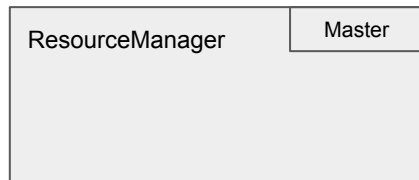
Yet Another Resource Negotiator

| YARN

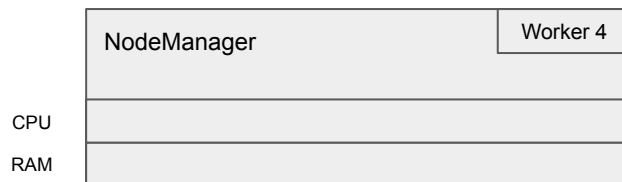
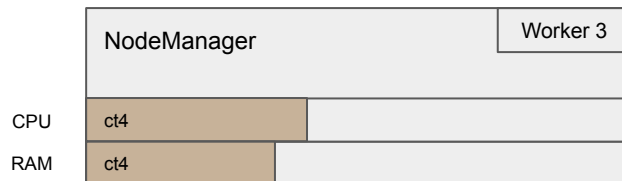
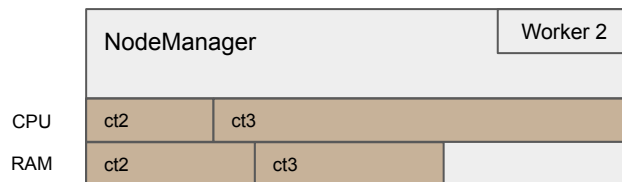
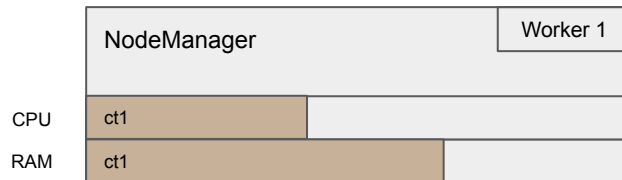
- ▼ Yet Another Resource Negotiator
- ▼ Gère la répartition des ressources de calcul
- ▼ Conteneur = CPU + RAM
- ▼ Queue pour la répartition équitable



YARN

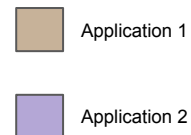
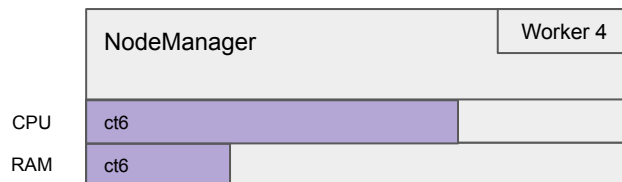
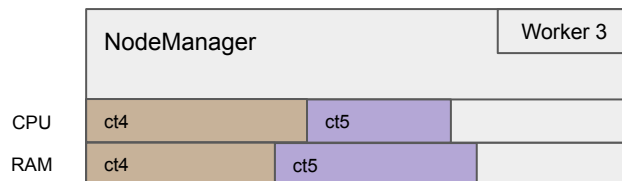
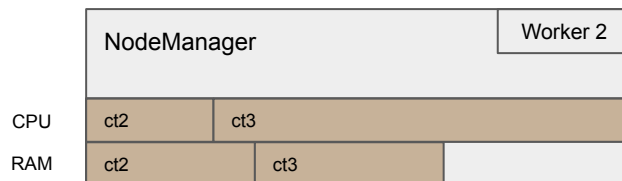
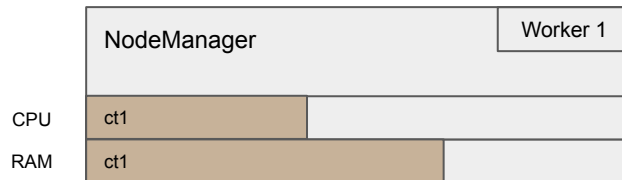
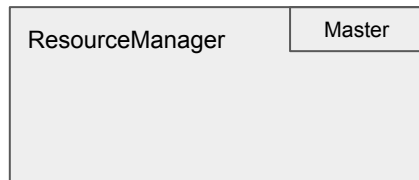


YARN

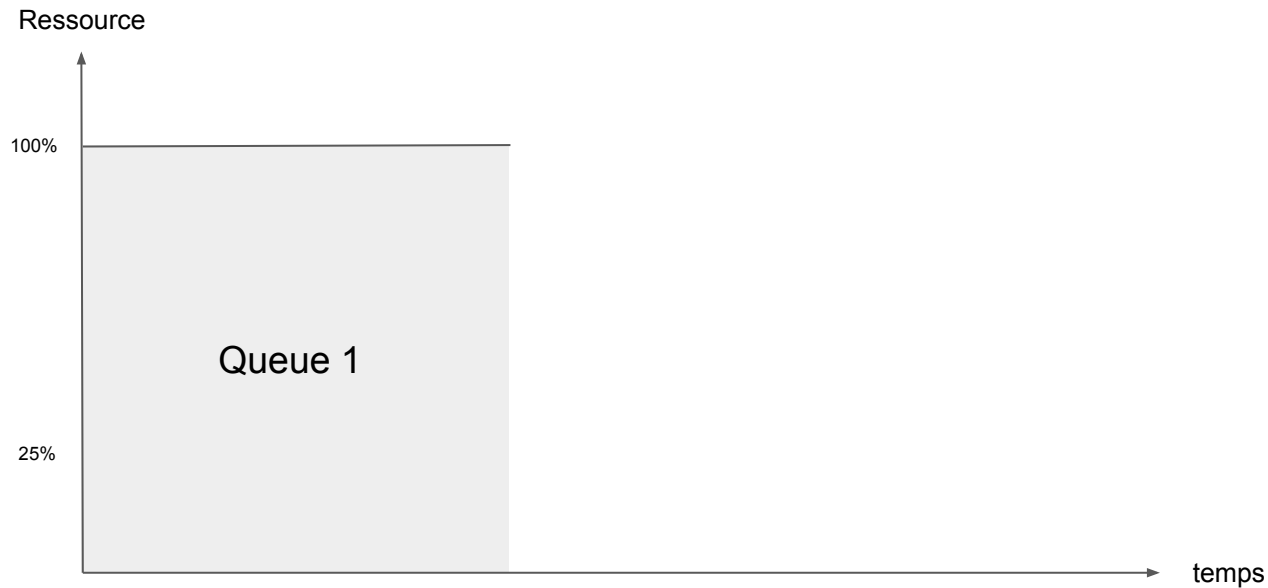


 Application 1

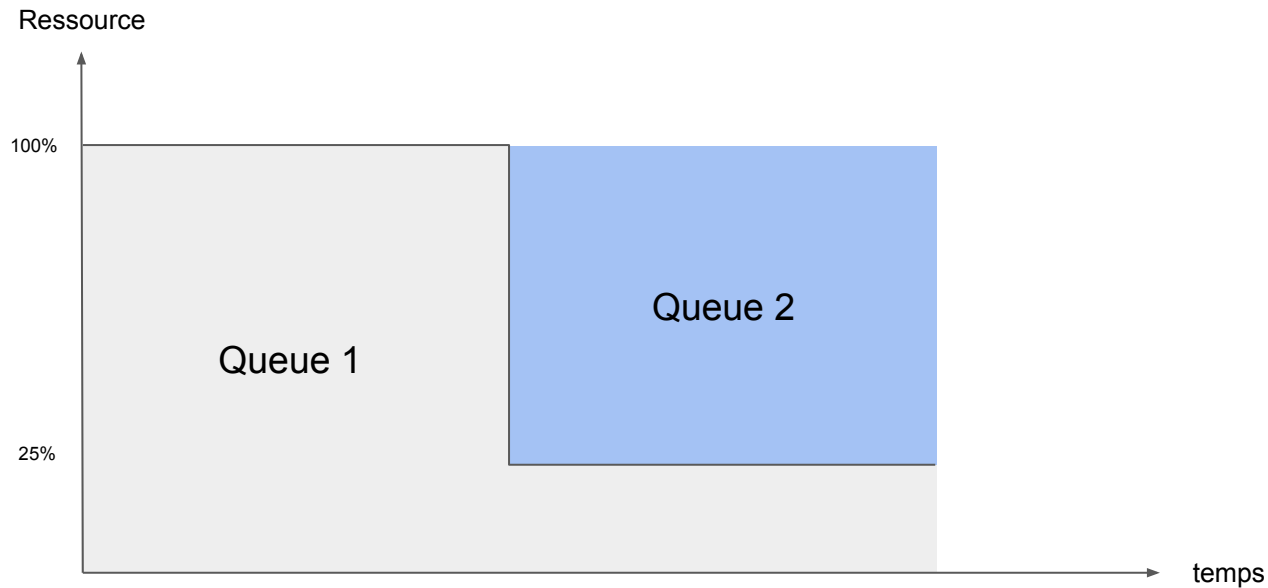
YARN



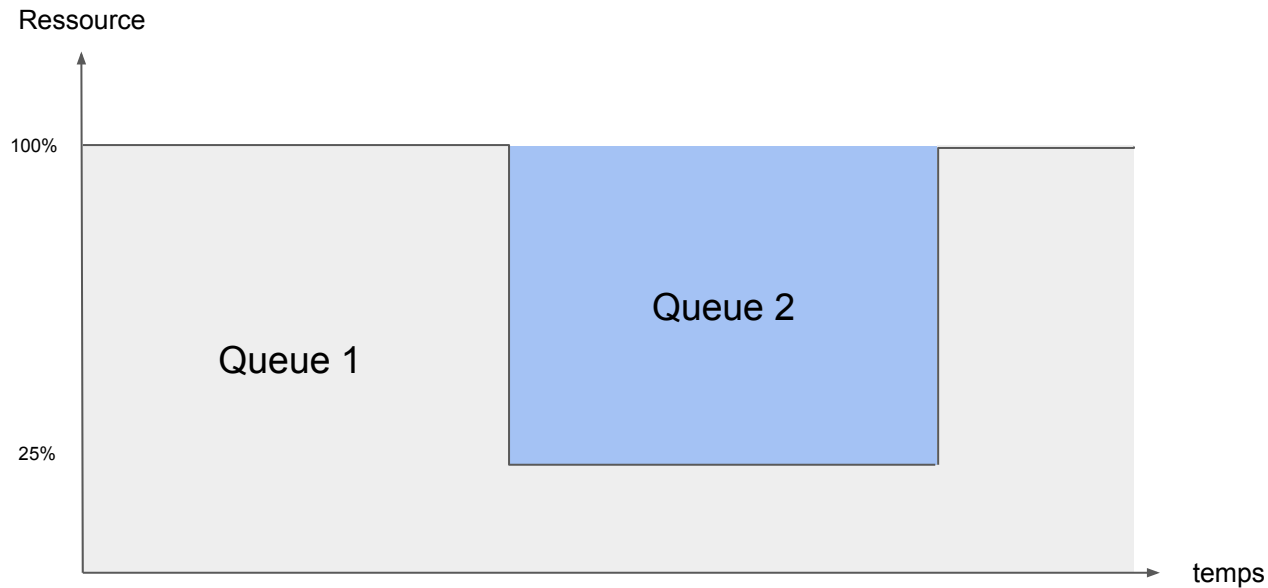
| YARN Queue



| YARN Queue



YARN Queue



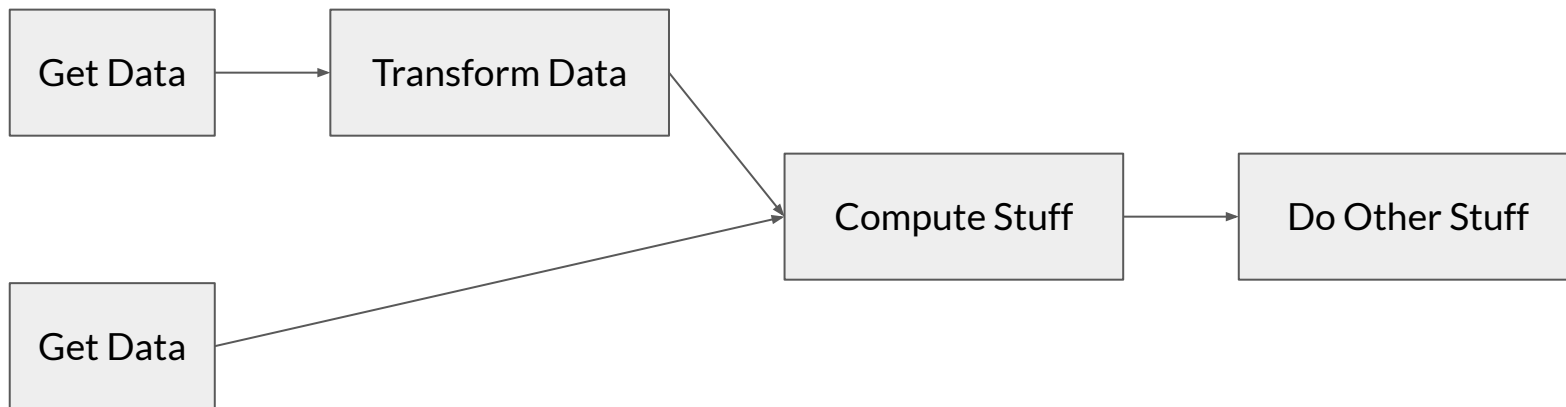
Oozie

Oozie

- ▼ Workflow
 - ▼ suite de job ayant des dépendances
- ▼ Coordinator
 - ▼ lancement de workflow à heure fixe
 - ▼ lancement suite à événement (arrivé d'un fichier)

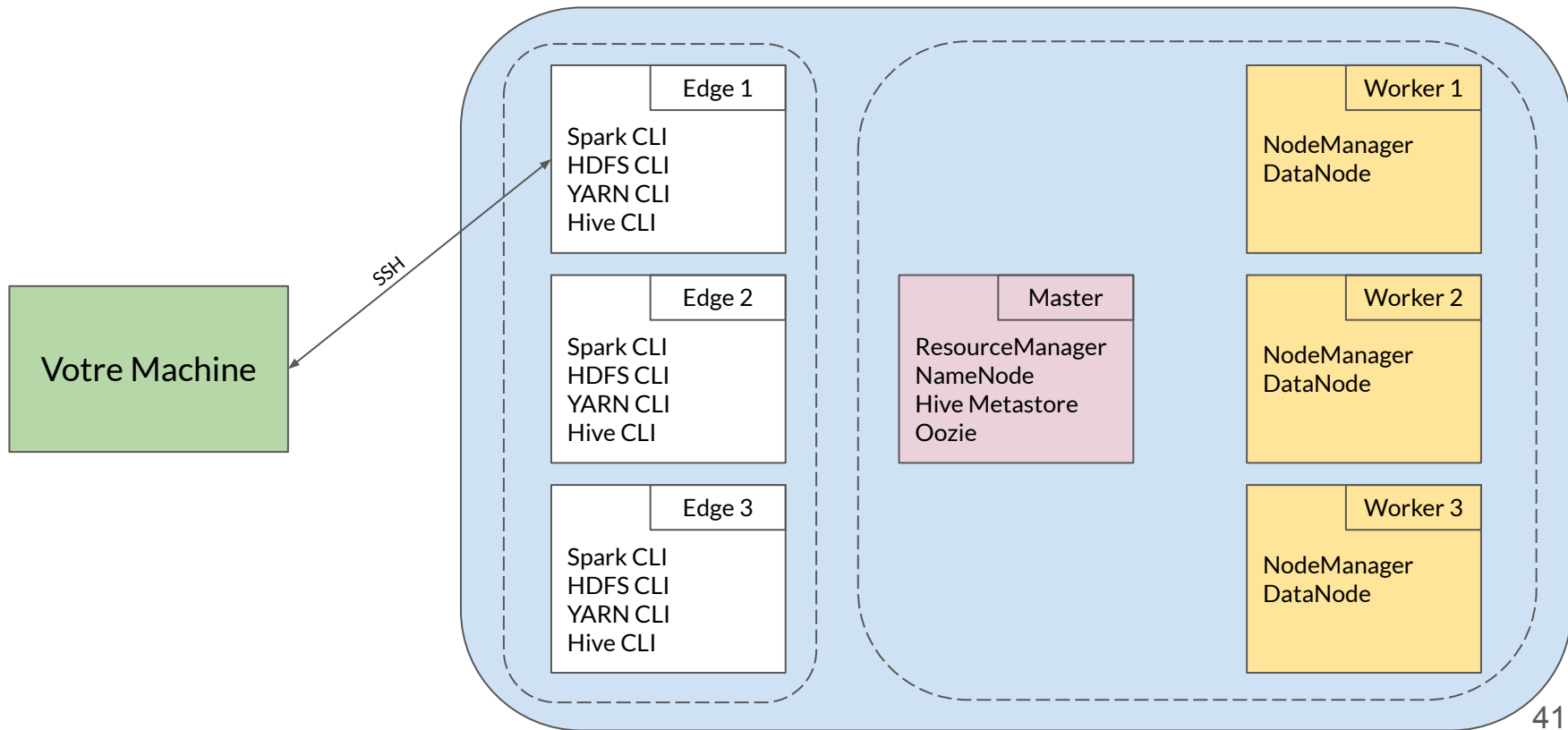


| Oozie : Workflow



Organisation d'un Cluster

Organisation d'un cluster



Analyse de données avec Spark

Introduction

Python & Scala basics

Introduction

Python & Scala basics

- ▼ Quelques bases de Python et Scala
- ▼ Déclaration de variable
- ▼ Structures de contrôle
- ▼ Définition de fonction
- ▼ Fonctions anonymes
- ▼ Création des principales structures de données
- ▼ Class

Python

- ▼ Pas de mot clé nécessaire
- ▼ Pas de spécification de type

```
# Déclaration d'une variable  
  
my_int = 2  
  
my_string = "Hello, World!"
```

Scala

- ▼ Mot clé val (ou var)
- ▼ Spécification de type possible

```
// Déclaration d'une variable  
  
val myInt: Int = 2  
  
val myString: String = "Hello, World!"
```

Python

```
if a == "foo":  
    print "foo"  
elif a == "bar":  
    print "bar"  
else:  
    print "unknown"
```

Scala

```
if(a == "foo") {  
    println("foo")  
} else if(a == "bar") {  
    println("bar")  
} else {  
    println("unknown")  
}  
  
// autre mécanisme  
a match {  
    case "foo" => println("foo")  
    case "bar" => println("bar")  
    case _ => println("unknown")  
}
```

Python

```
i = 0
while i < 5:
    print i
    i += 1

# ou

for i in range(0, 5):
    print i
```

Scala

```
var i = 0
while(i < 5) {
    println(i)
    i += 1
}

// ou

for(i <- 0 until 5) {
    println(i)
}
```

Python

- ▼ Un tuple est une structure formée d'un ou plusieurs éléments dont les types peuvent différer.

```
my_tuple = ("key", 1)

my_tuple[0] # Affiche key
my_tuple[1] # Affiche 1
```

Scala

```
val myTuple = ("key", 1)

println(myTuple._1) // Affiche key
println(myTuple._2) // Affiche 1
```


Python

- ▼ Une liste est une structure contenant plusieurs éléments.

```
# Création d'une liste
my_list = [1,2,3,4]

# Accéder aux éléments d'une liste
my_list[0]

# Longueur de la liste
len(my_list)

# Itérer sur la liste
for v in my_list:
    print v
```

Scala

- ▼ En scala les listes sont typées. Les éléments doivent donc être du même type

```
// Création d'une liste
val myList = List(1,2,3,4)
val myList2 = List[String]("1","2","3","4")

// Accéder aux éléments d'une liste
myList(0)

// Longueur de la liste
myList.size

// Itérer sur la liste
for(v <- myList) {
    println(v)
}
```

Python

- ▼ Utilisation du mot clé *def* pour les fonctions
- ▼ Mot clé *return* permet de retourner une valeur

```
# Définition d'une fonction
def sum(n1, n2):
    return n1 + n2

# Définition d'une fonction avec
# une valeur par défaut
def sum(n1, n2=3):
    return n1 + n2
```

Scala

- ▼ Utilisation du mot clé *def* pour les fonctions
- ▼ La dernière expression est retournée

```
// Définition d'une fonction
def sum(n1: Double, n2: Double) = {
    n1 + n2
}

// Définition d'une fonction avec une
// valeur par défaut
def sum(n1: Double, n2: Double = 3) = {
    n1 + n2
}
```

Python

▼ `lambda var1, var2: <corps sur une ligne>`

```
lambda a, b: a + b
```

Scala

▼ `(var1, var2) => <corps sur une ligne>`

```
(a, b) => a + b
```

// ou

```
(a, b) => {
```

```
    a + b
```

```
}
```

// ou

```
_ + _
```

Python

- ▼ On applique une fonction anonyme sur
chaques élément de la liste

```
# Création d'une liste
my_list = [1,2,3,4]

# Transformer une liste
map(lambda a: a * 2, my_list)

# Filtrer une liste
filter(lambda a: a % 2 == 0, my_list)

# Sommer les éléments d'une liste
reduce(lambda a, b: a + b, my_list)
```

Scala

```
// Création d'une liste
val myList = List(1,2,3,4)

// Transformer une liste
myList.map(_ * 2)

// Filtrer une liste
myList.filter(_ % 2 == 0)

// Sommer les éléments d'une liste
myList.reduce(_ + _)
```

▼ En Python

- ▼ association clé / valeur, clés et valeurs peuvent être de différents types
- ▼ création via fonction dict ou sucre syntaxique {}

```
d = {  
    "foo": 1,  
    32 : ["a", "b"],  
}  
  
print d["foo"]
```

▼ En scala

- ▼ association clé/valeur
- ▼ le type de la clé peut différer du type de la valeur
- ▼ toutes les clés doivent être du même type (de même pour les valeurs)

```
val m: Map[String, Double] = Map("un" -> 1.0, "deux" -> 2.0)  
  
println(m("un"))
```

Python

```
import sys

def main(argv):
    print(argv[1])

if __name__ == "__main__":
    main(sys.argv)
```

Scala

```
package fr.esgi.course

object MyApp {
    def main(args: Array[String]): Unit = {
        println(args(0))
    }
}
```

Python

```
import sys

class MyClass(AnotherClass):
    def __init__(self, my_var):
        self.my_var = my_var

    def one_func(self, key):
        return key + self.my_var + 2

obj = MyClass(2)
print(MyClass.one_func(3)) # 6
```

Scala

```
package fr.esgi.course

class MyClass(myVar: Int) extends AnotherClass {
    def oneFunc(key: Int): Int = {
        key + myVar + 1
    }
}

val obj = new MyClass(2)
println(obj.oneFunc(3)) // 6
```


Introduction à Spark

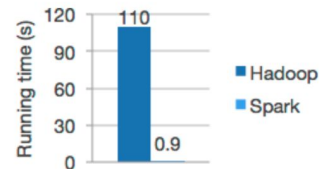
Prise en main

Introduction à Spark

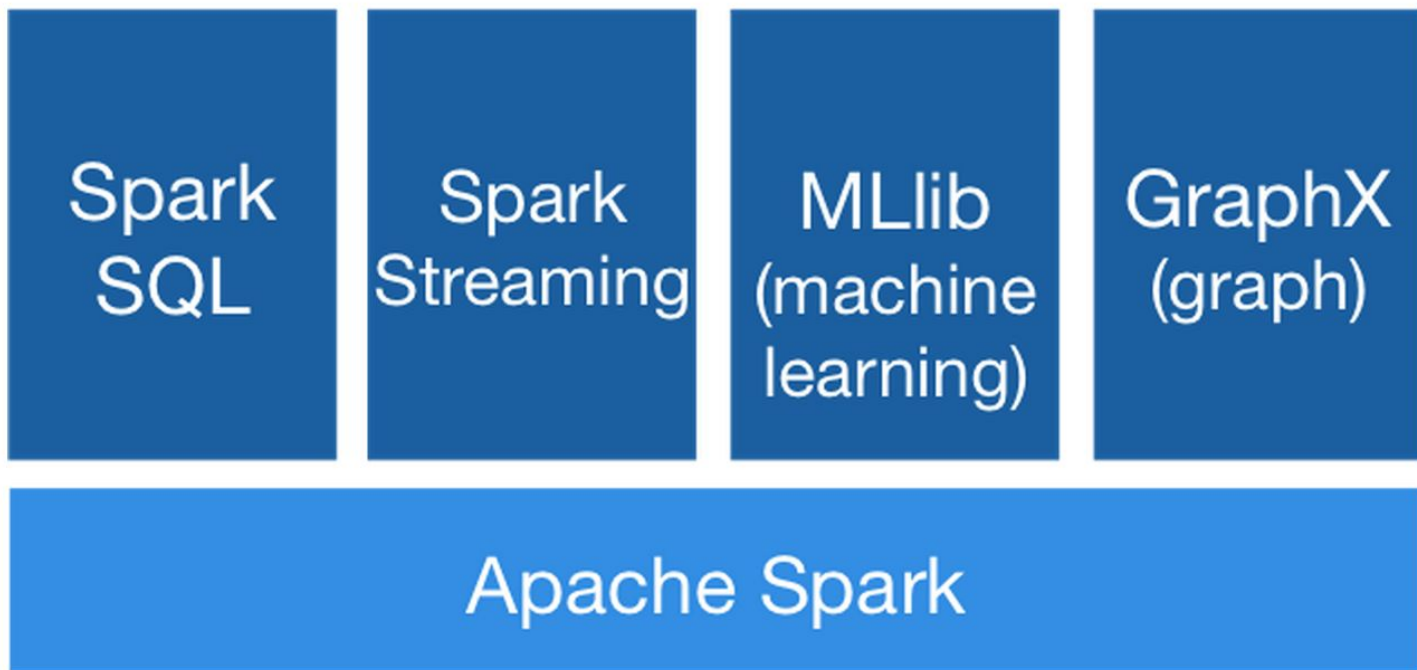
Prise en main

- ▼ Introduction à MapReduce
- ▼ Introduction à Spark
- ▼ Exécution d'un job avec Spark
- ▼ Travail interactif avec un shell ou un notebook

- ▼ Framework de manipulation de données
- ▼ Créé en 2010 à l'AMPLab de l'UC Berkeley
 - ▼ Aujourd'hui géré par Databricks
- ▼ Projet Apache en juin 2013
 - ▼ Apache Top-Level Project en février 2014
- ▼ Fréquemment utilisé avec Hadoop (HDFS et YARN)
- ▼ Ne se limite pas au concept de *map-reduce*
- ▼ Basé sur la montée en mémoire des données
 - ▼ Performances jusqu'à 100 fois plus rapides qu'avec du MapReduce
- ▼ Aujourd'hui en version 2.4.X



Logistic regression in Hadoop and Spark



Map Reduce

| Map Reduce

Modèle de programmation inventé en 2004 par Google. Il est utilisé pour paralléliser les calculs en tirant parti d'un système de fichier distribué.

- ▼ Map

- ▼ Projection (SELECT)

- ▼ Filtre (WHERE)

- ▼ Reduce

- ▼ Aggregation (GROUP BY)

- ▼ Jointure (JOIN)

| Compter des Lego avec map reduce

Combien y a-t-il de brique 8 et 4 dans ce tas de cartes ?

1. distribuer des lego à chaque node
2. chaque node filtre les 8 et 4
3. chaque node compte les 8 et 4
4. tous les comptes de 8 sont envoyés à un node
5. tous les comptes de 4 sont envoyés à un node
6. Le "Node 8" ajoute les différents comptes
7. Le "Node 4" ajoute les différents comptes

| Spark

- ▼ Découpe le dataset en petit morceaux
- ▼ Traite un morceaux à la fois
- ▼ Enchaîne les phases de map et reduce

Architecture de Spark

YARN - Client mode

Edge Node

worker

worker

Master

worker

YARN - Cluster mode

Edge Node

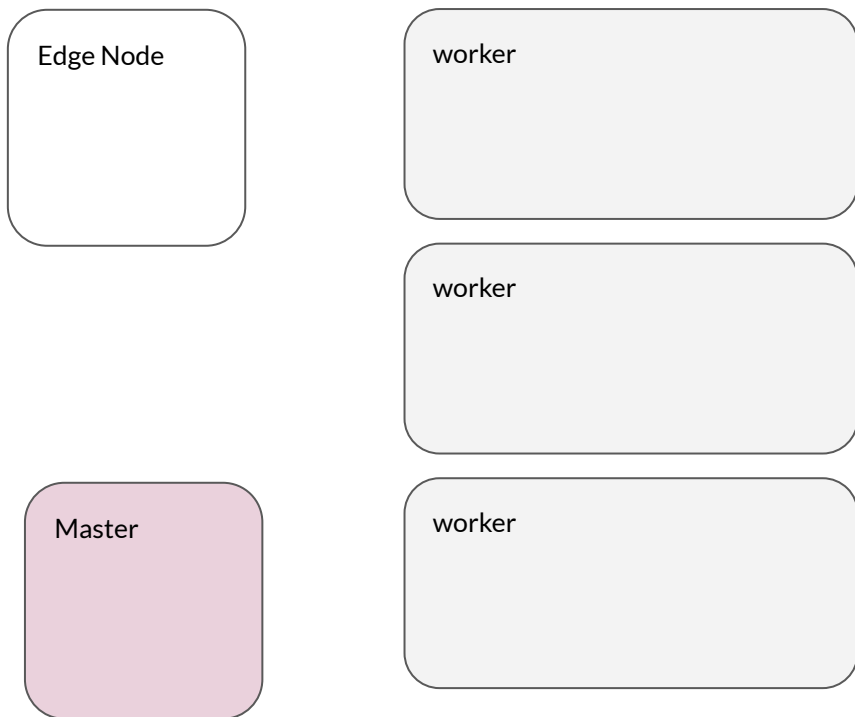
worker

worker

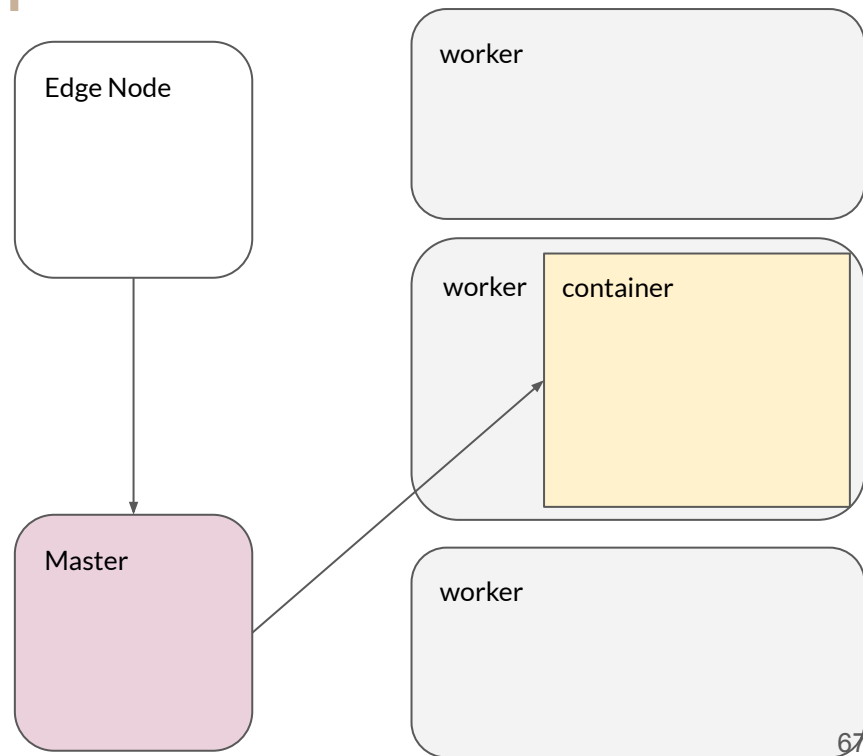
Master

worker

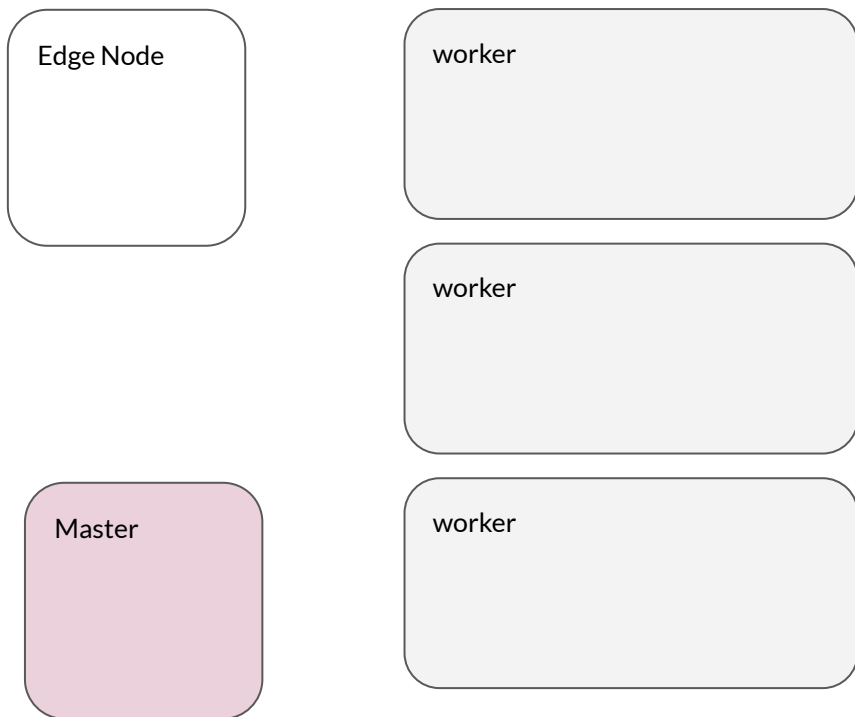
YARN - Client mode



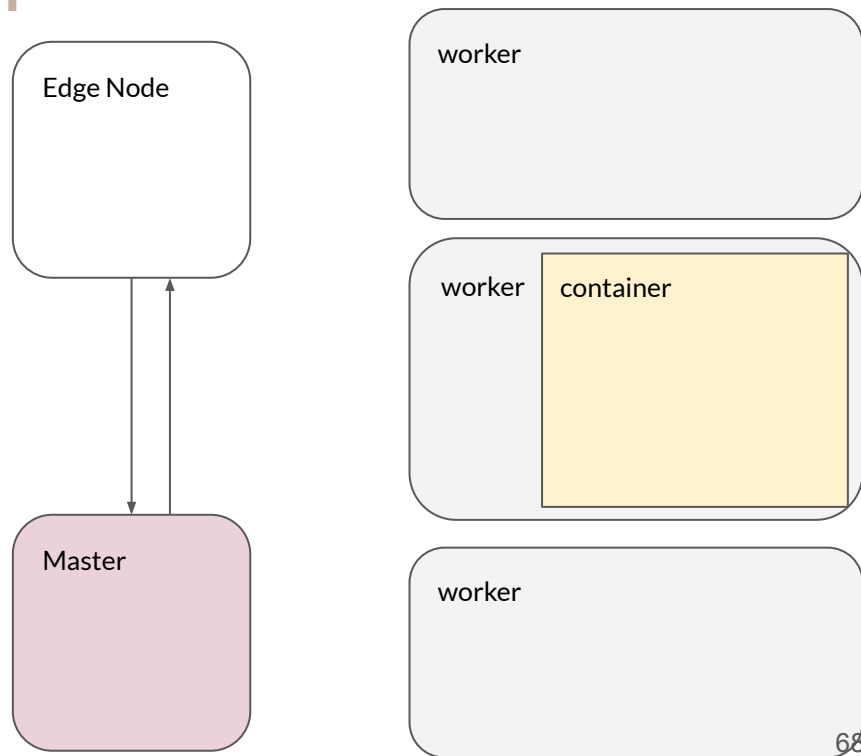
YARN - Cluster mode



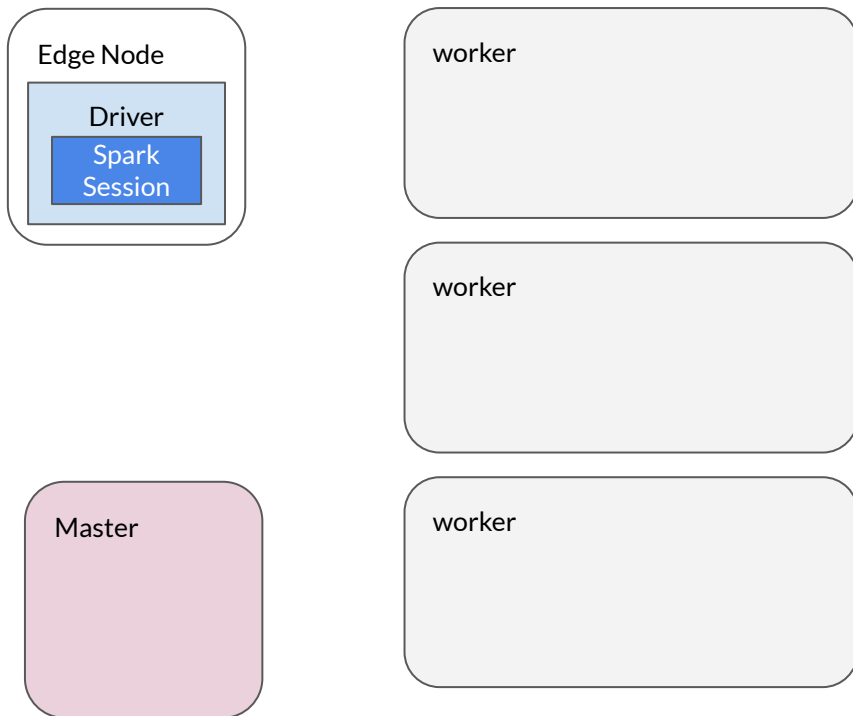
YARN - Client mode



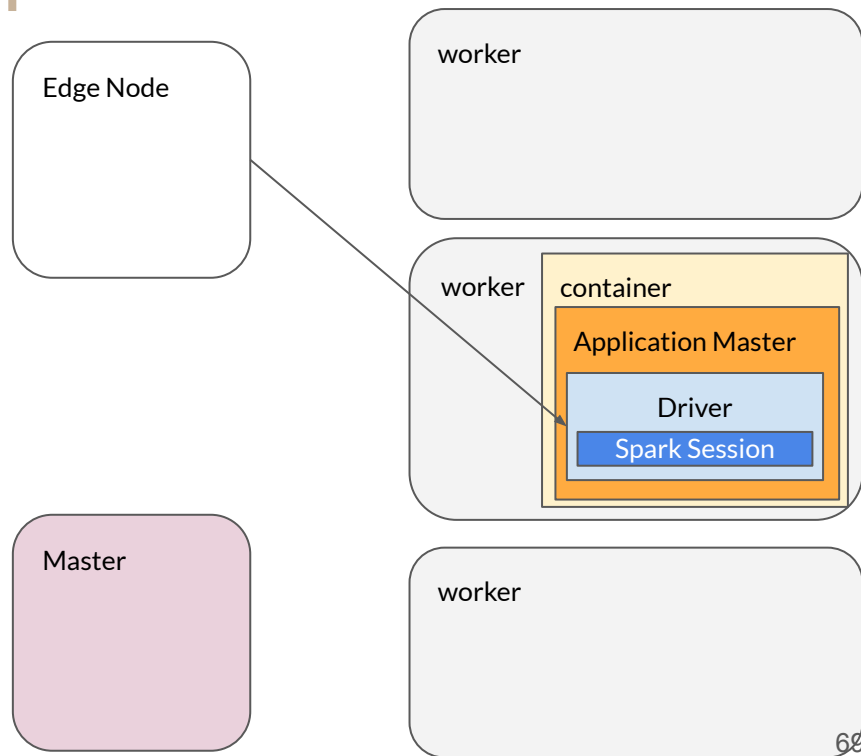
YARN - Cluster mode



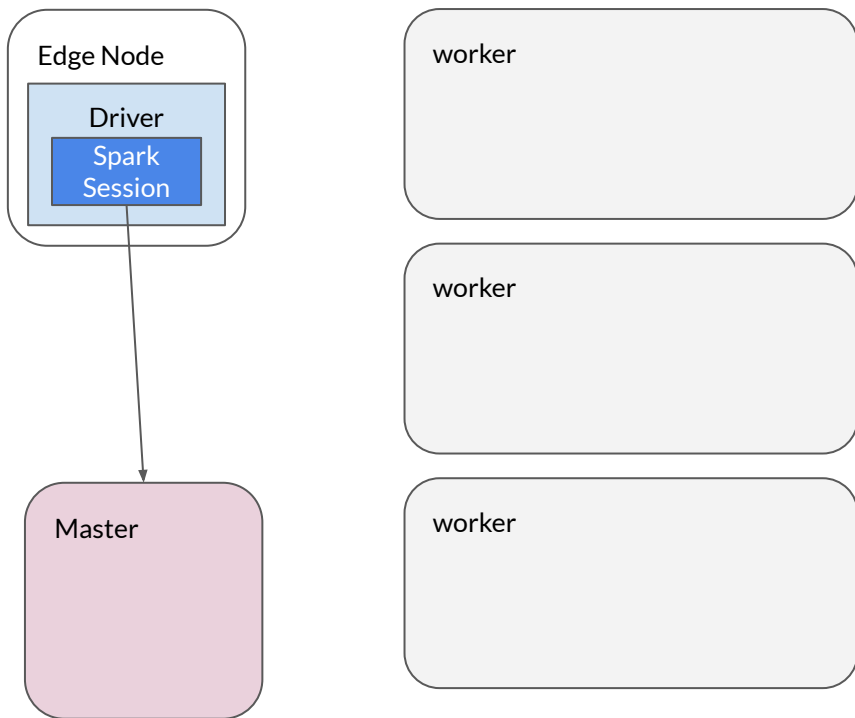
YARN - Client mode



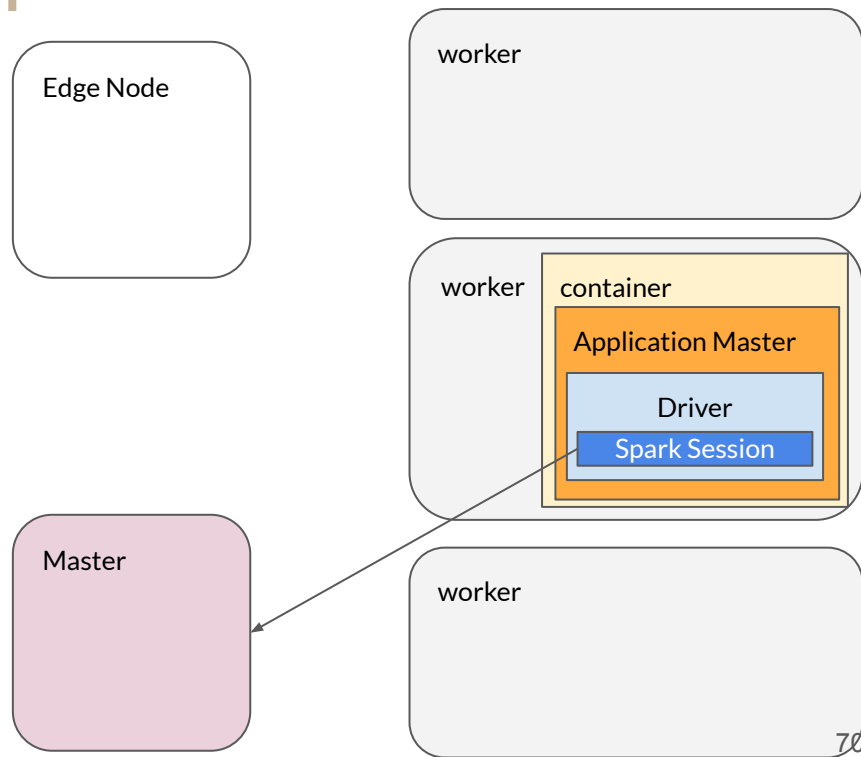
YARN - Cluster mode



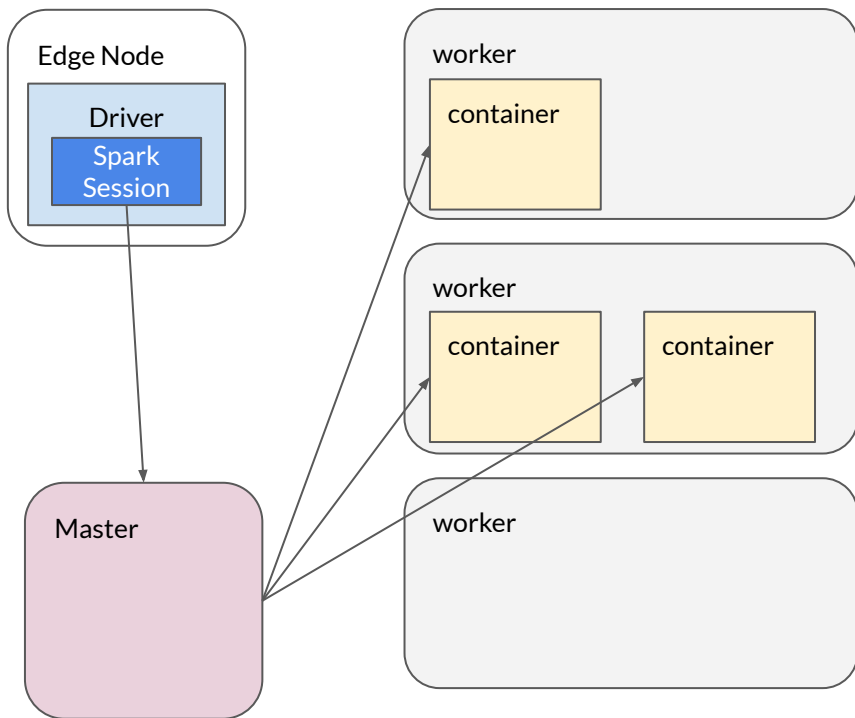
YARN - Client mode



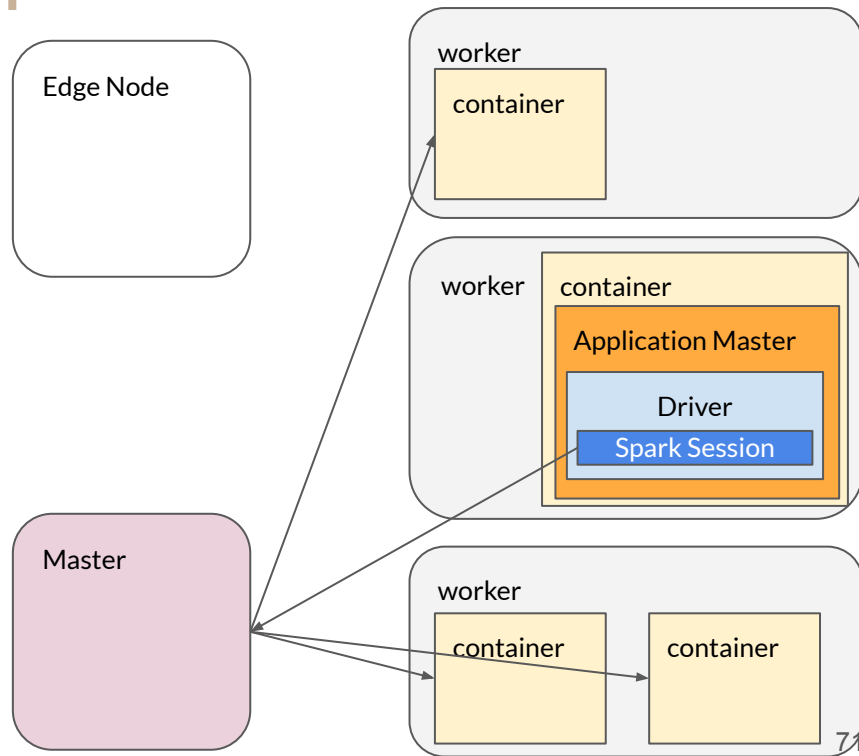
YARN - Cluster mode



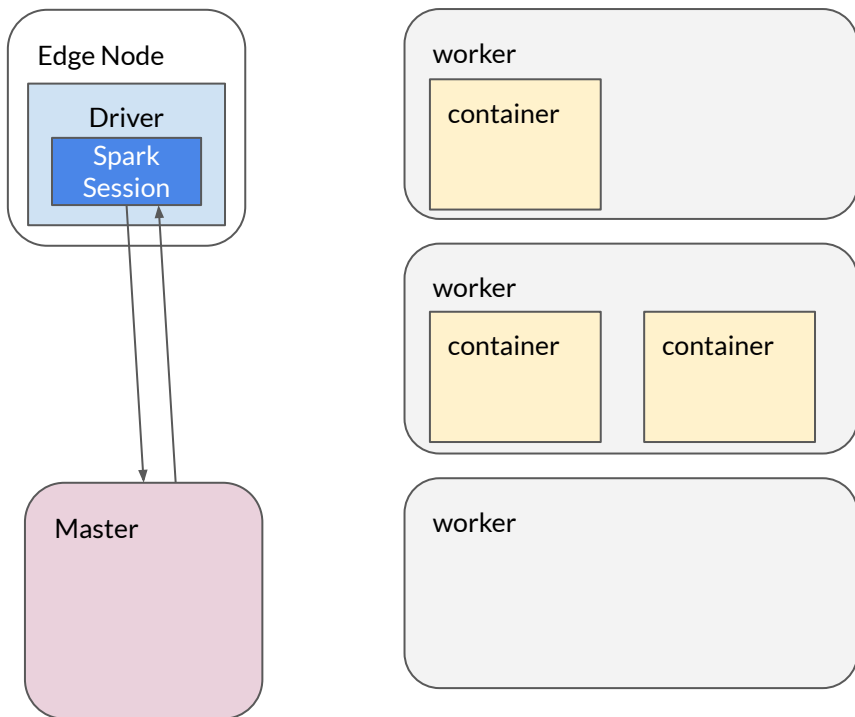
YARN - Client mode



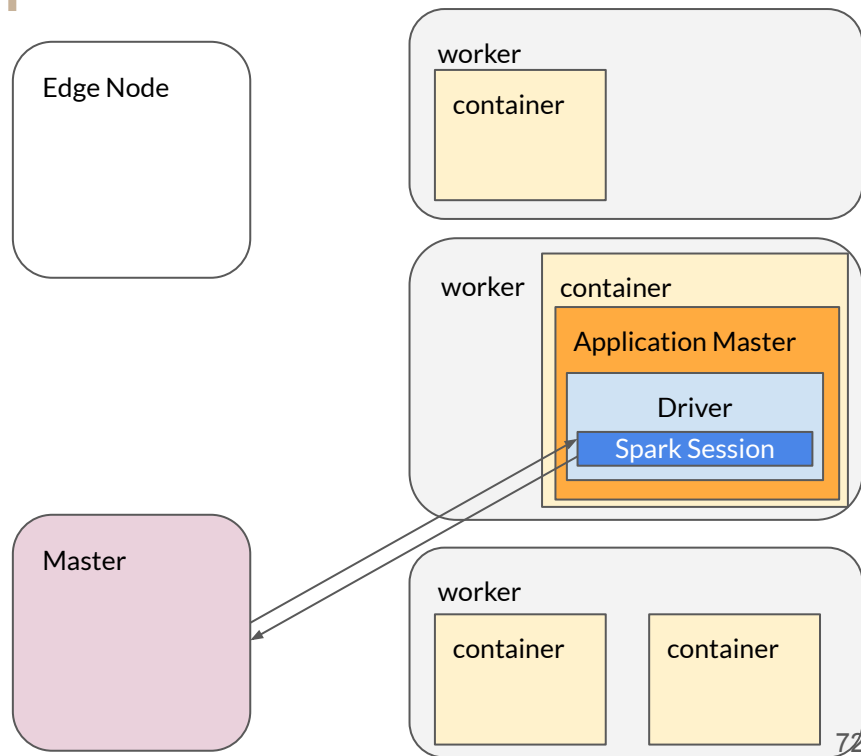
YARN - Cluster mode



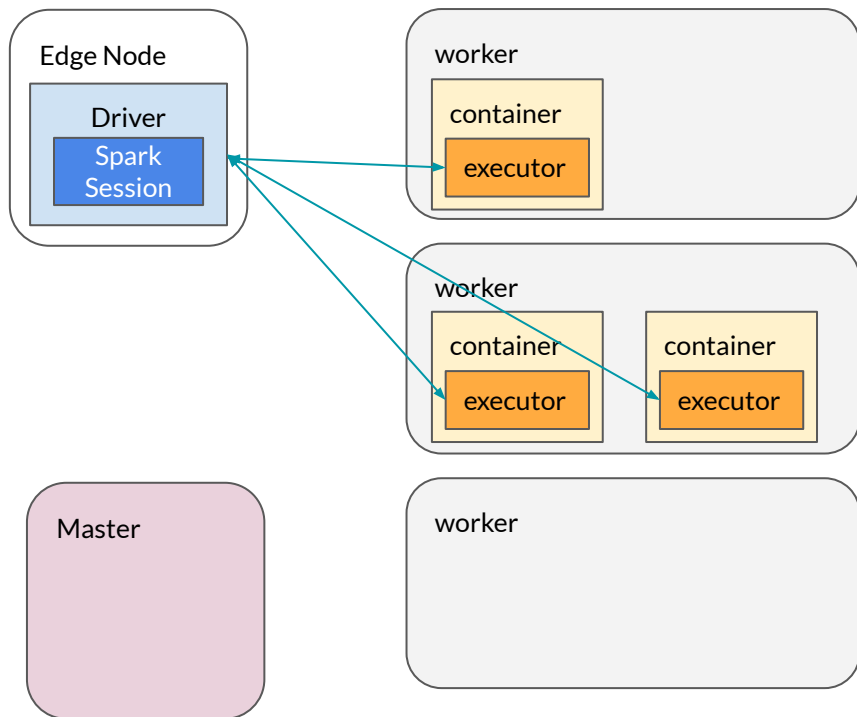
YARN - Client mode



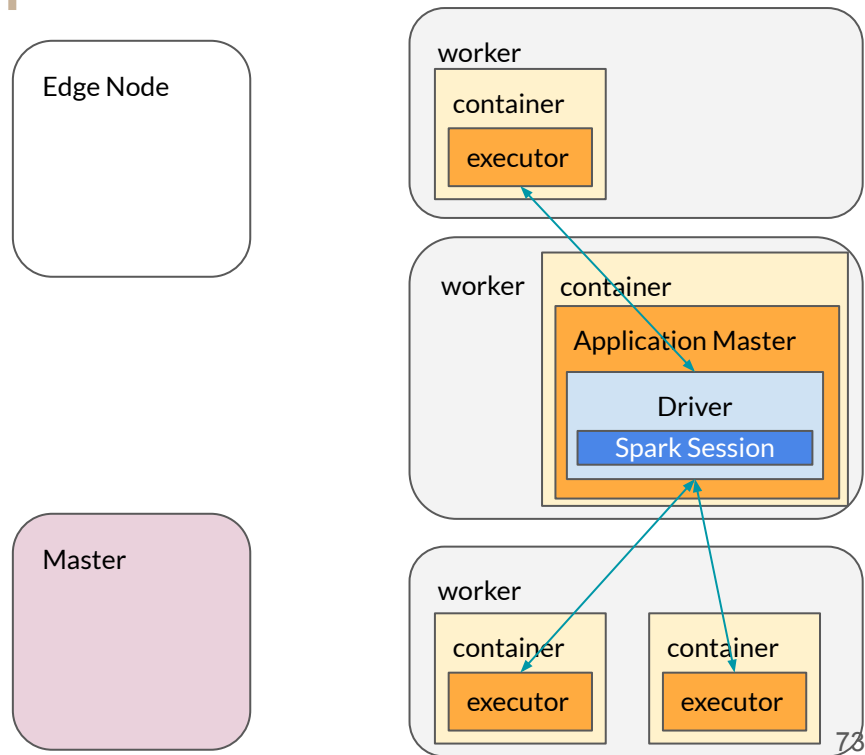
YARN - Cluster mode



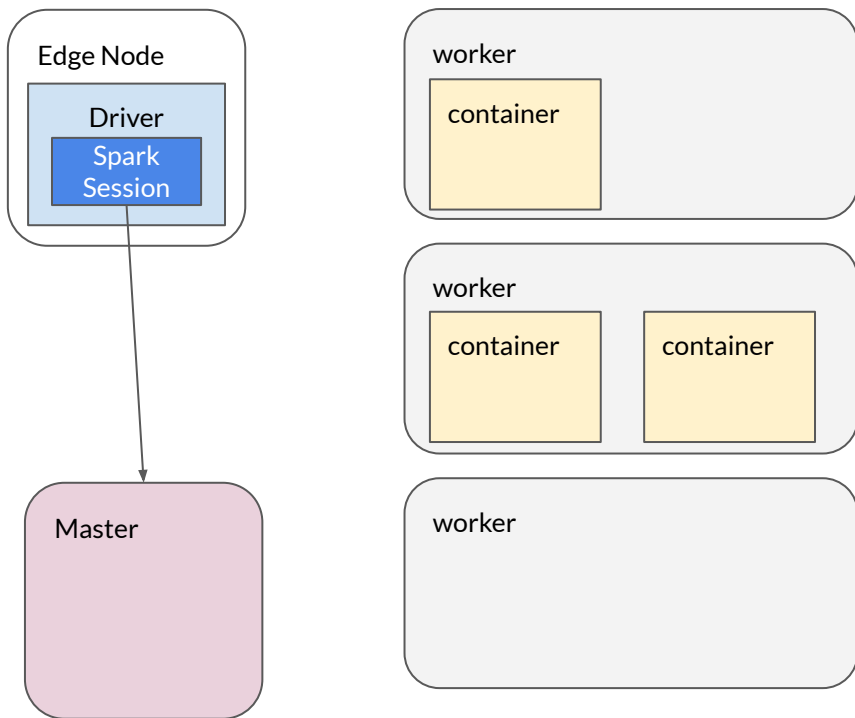
YARN - Client mode



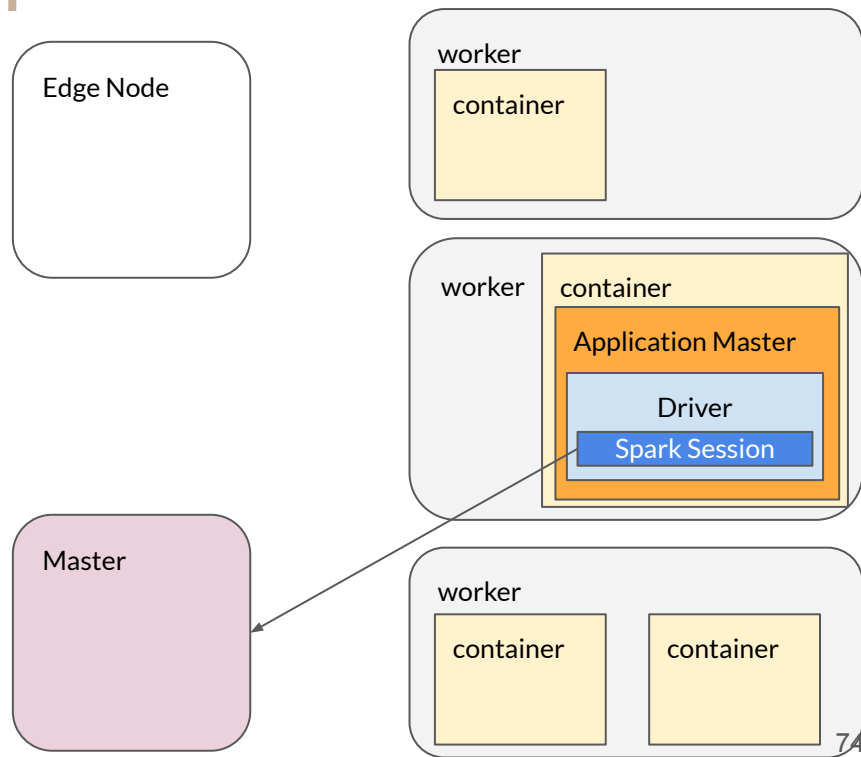
YARN - Cluster mode



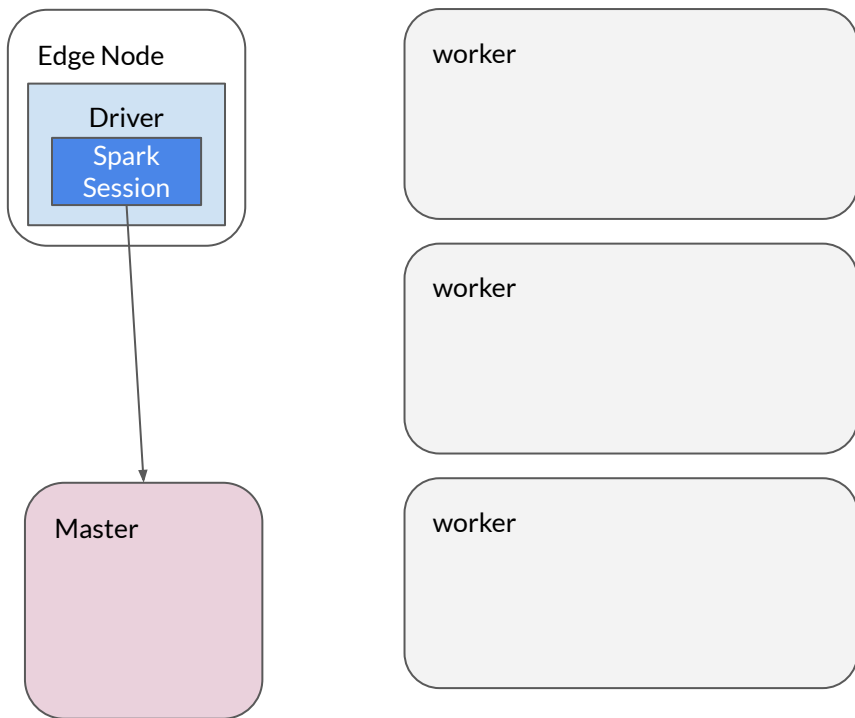
YARN - Client mode



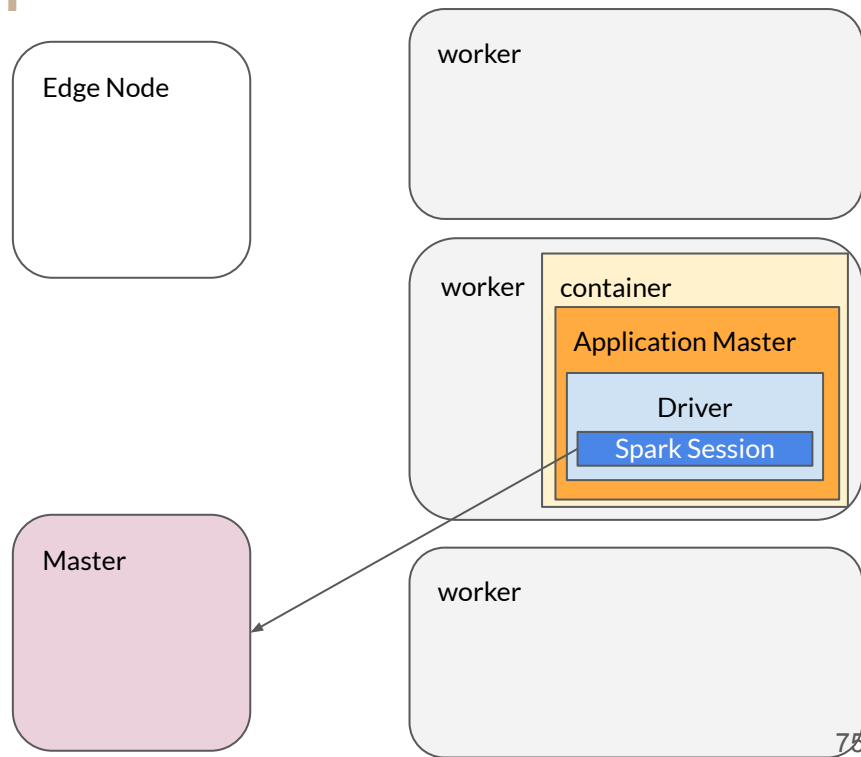
YARN - Cluster mode



YARN - Client mode



YARN - Cluster mode



YARN - Client mode

Edge Node

worker

worker

Master

worker

YARN - Cluster mode

Edge Node

worker

worker

Master

worker

| Vocabulaire - Spark Application

- ▼ Une application Spark se compose d'un processus **driver** et d'un ensemble de processus **executors**
- ▼ **Driver**
 - ▼ exécute la fonction *main()*
 - ▼ garde les informations sur l'application
 - ▼ analyse, distribue et planifie le travail des *executors*
- ▼ **Executors**
 - ▼ exécutent *de facto* le travail qui leur est assigné par le *driver*
 - ▼ reportent l'état (success or failure) et le résultat de l'exécution au *driver*

- ▼ Les *executors* exécutent toujours du code Spark, le *driver* peut être en différents langages mis à disposition grâce à 4 APIs :
 - ▼ Scala: langage dans lequel Spark est écrit
 - ▼ Java
 - ▼ Python
 - ▼ R
- ▼ Java est cependant moins utilisé. On privilégiera Python et Scala.

| Comment interagir avec Spark

- ▼ Soumission d'une application avec spark-submit

```
spark-submit \  
  --master yarn \  
  --deploy-mode cluster \  
  examples/src/main/python/pi.py \ 1000
```

- ▼ Démarrage d'un shell interactif

```
pyspark --master local[*]
```

```
spark-shell --master local[*]
```

- ▼ Démarrage d'un notebook ipython

```
PYSPARK_DRIVER_PYTHON=jupyter \  
PYSPARK_DRIVER_PYTHON_OPTS="notebook" \  
pyspark --master yarn
```

Spark

API

Spark

API

- ▼ Comprendre la notion de **Session** dans Spark
- ▼ Comment configurer Spark
- ▼ Découvrir la structure de données
- ▼ Distinguer les concepts de **Transformations** et d'**Actions**

| SparkSession

- ▼ Point d'entrée aux fonctionnalités de Spark
- ▼ correspondance 1:1 entre SparkSession et une application Spark
 - ▼ Son arrêt (avec la méthode `.stop`) entraîne l'arrêt de l'application
 - ▼ créé automatiquement au lancement du spark-shell
 - ▼ autrement en spécifiant dans le code :

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .getOrCreate()
```

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
    .builder
    .getOrCreate()
```

| La configuration de session : SparkConf

- ▼ L'objet *SparkConf* est un accès programmable à la configuration d'une application spark. Les configurations par défauts sont dans le fichier *conf/spark-defaults.conf*. Seule `spark.master` & `spark.app.name` sont des clefs de configuration obligatoires.

```
from pyspark.sql import SparkSession
from pyspark import SparkConf

conf = SparkConf() \
    .setAppName("weblogs processing") \
    .set("spark.executor.memory", "8g")

spark = SparkSession \
    .builder \
    .config(conf=conf) \
    .getOrCreate()
```

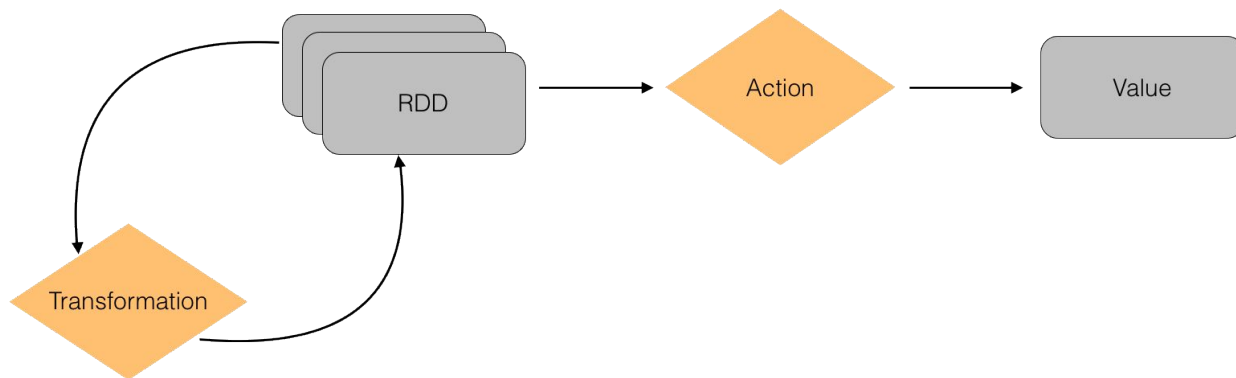
```
import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkConf

val conf = new SparkConf()
    .setAppName("weblogs processing")
    .set("spark.executor.memory", "8g")

val spark = SparkSession
    .builder
    .config(conf)
    .getOrCreate()
```

Lazy evaluation

- ▼ Dans Spark les structures de données sont **immuables**
- ▼ Pour modifier les données il faut appliquer des **transformations**
 - ▼ Elles ne sont exécutées que si une **action** les déclenche
- ▼ Une action donne l'ordre à Spark de calculer un résultat selon la liste de transformations



| Les actions

- ▼ 3 catégories d'action :
 - ▼ pour visualiser les données dans la console
 - ▼ pour récupérer les données comme objets du langage choisi
 - ▼ pour écrire les données

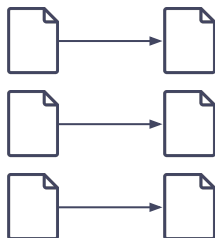
```
print rdd.collect()
```

```
> [('Alice', '22'), ('Bob', '30')]
```

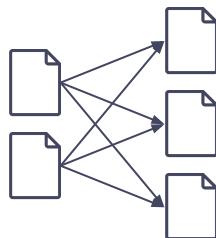
Les transformations

- ▼ 2 catégories de transformation
 - ▼ **narrow** : transformation qui est exécutable sur une partition indépendamment des autres partitions
 - ▼ **wide** : transformation qui a besoin d'accéder à plusieurs partitions pour être exécutée
- ▼ La présence d'une transformation de type *wide* engage un *shuffle* quand l'action est appelée
 - ▼ **shuffle** : Spark échange des partitions à travers le cluster
 - ▼ en terme de performance le *shuffle* est coûteux

Narrow transformation
1:1



Wide transformation
(shuffles) 1:n



Action

- ▼ collect
- ▼ take
- ▼ show
- ▼ count
- ▼ save
- ▼ foreach

Transformation

- ▼ map
- ▼ select
- ▼ filter
- ▼ where
- ▼ group by
- ▼ join
- ▼ reduceByKey

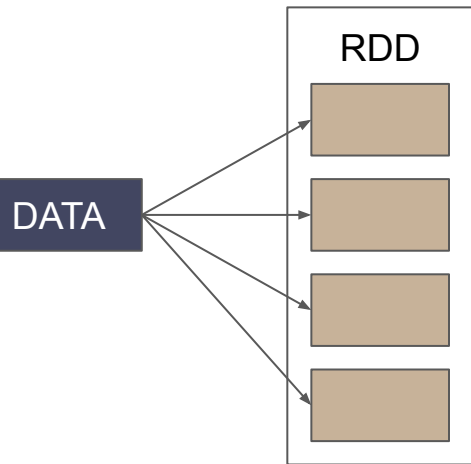
| Partition

- ▼ Les données dans un RDD sont partitionné et répartie sur les executor (d'ou Distribué)
- ▼ Avec les Dataset/DataFrame le partitionnement est automatique
- ▼ Avec les RDD on peut contrôler le partitionnement
- ▼ Plus de partitions = Plus de parallélisme



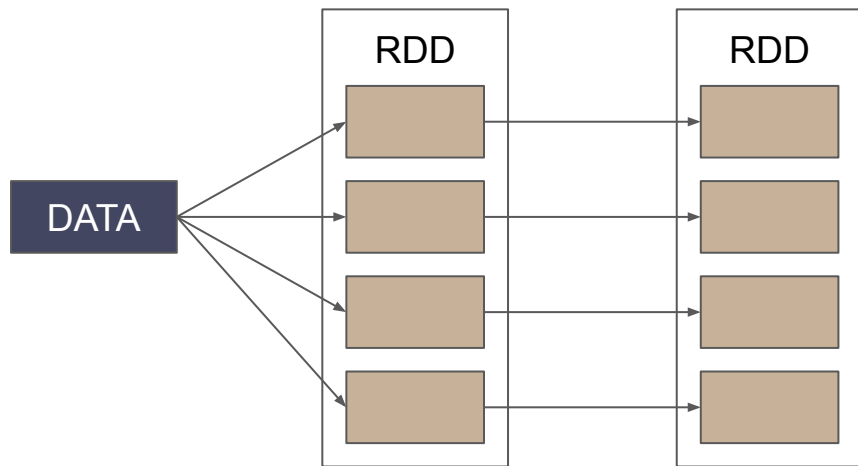
| Task and Stage

```
val wc = sc.textFile(myData)
```



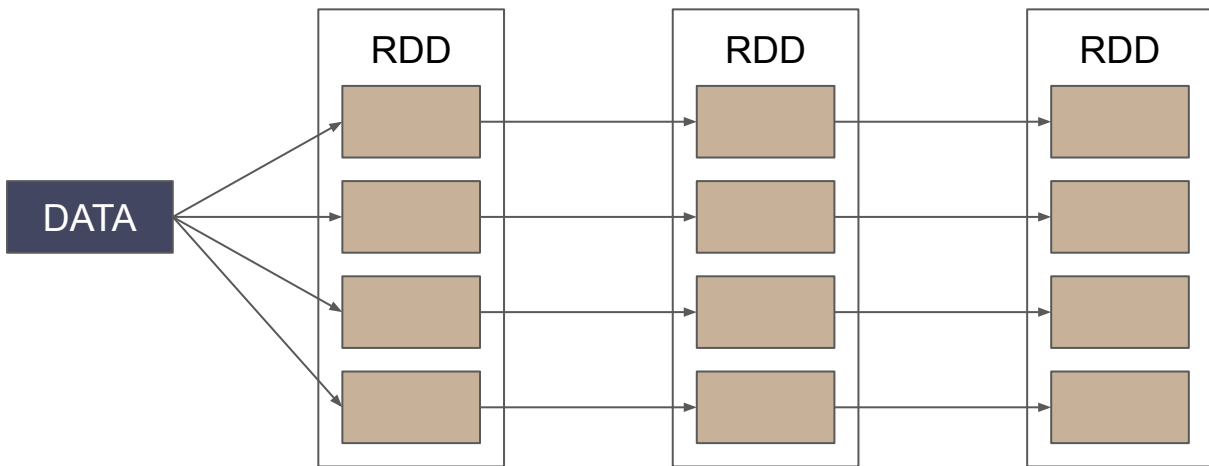
| Task and Stage

```
val wc = sc.textFile(myData)  
    .map(_ .split(","))
```



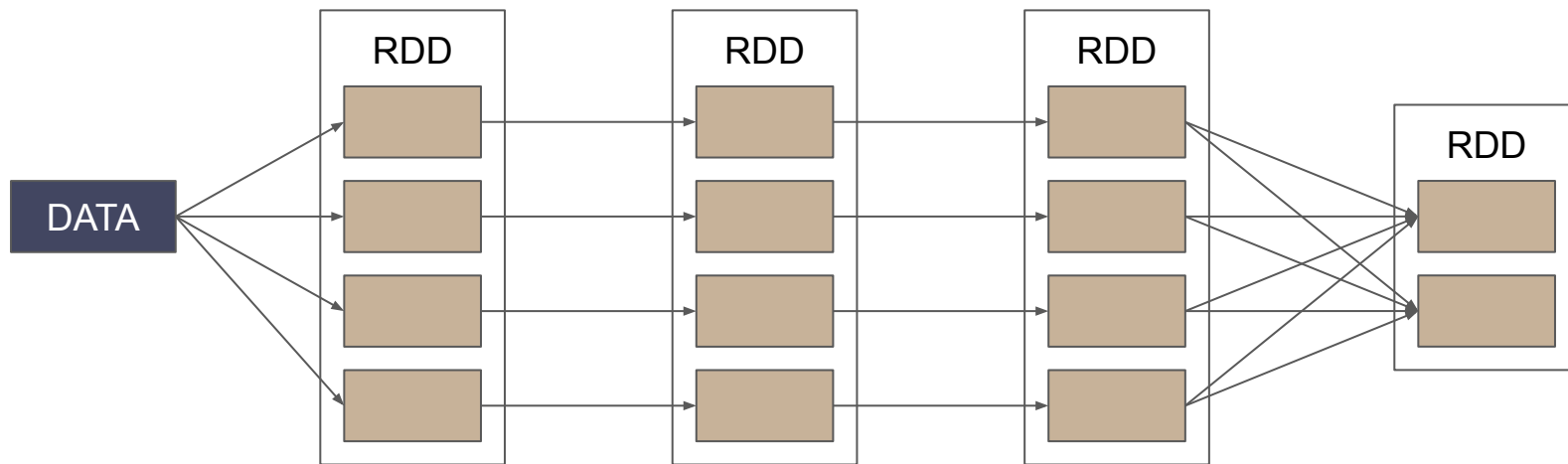
| Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
```



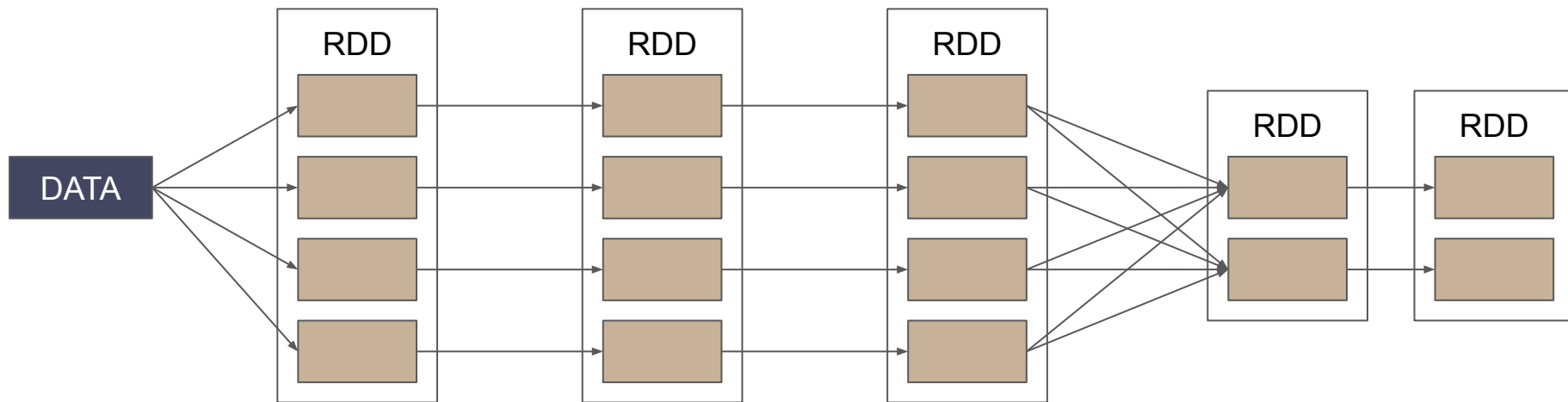
Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
```



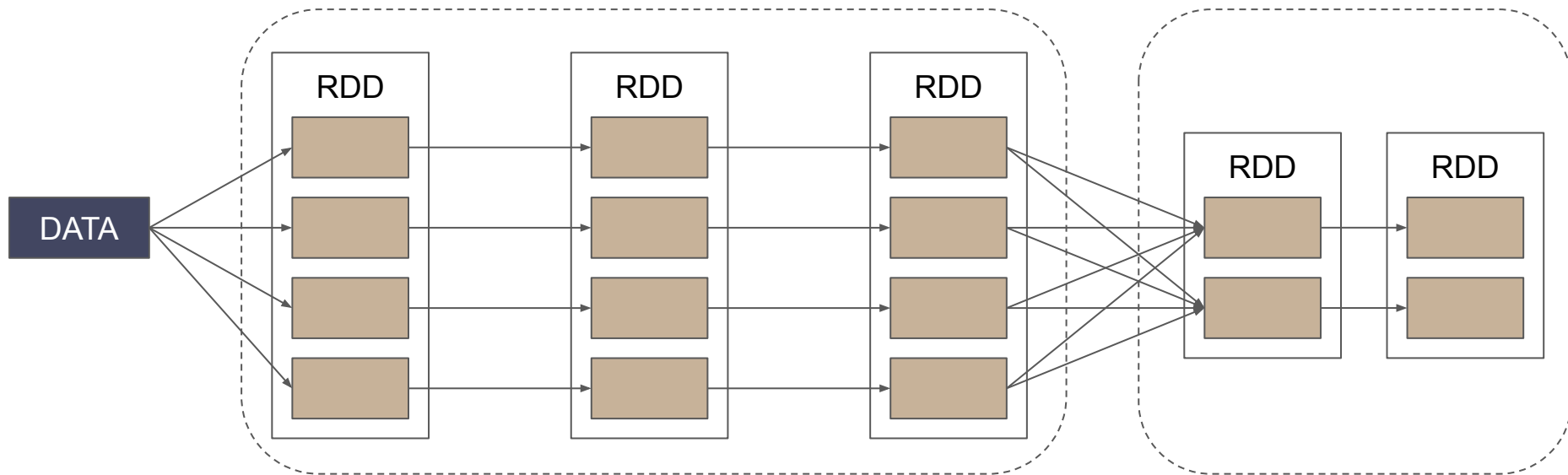
Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
  .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



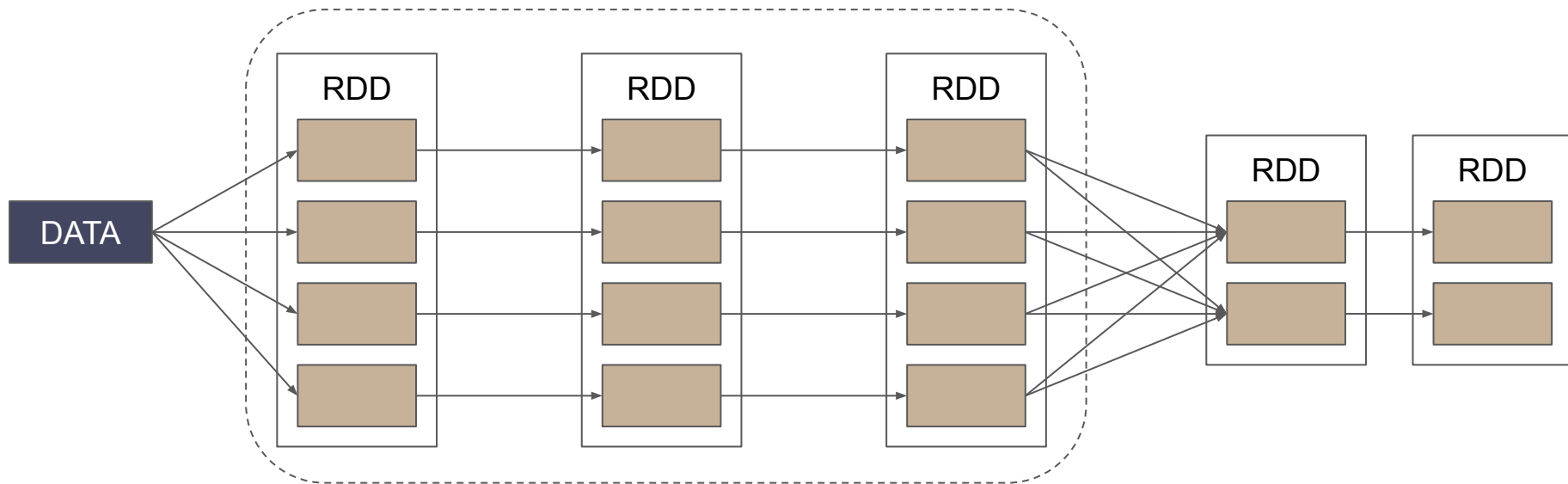
Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
  .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



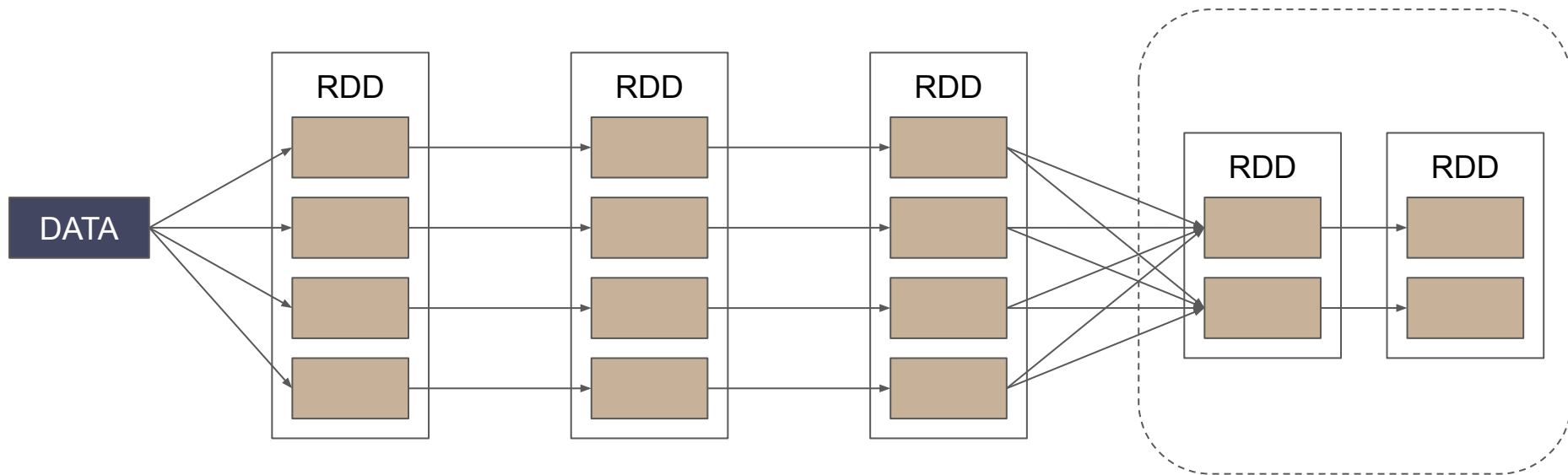
Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
  .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



Task and Stage

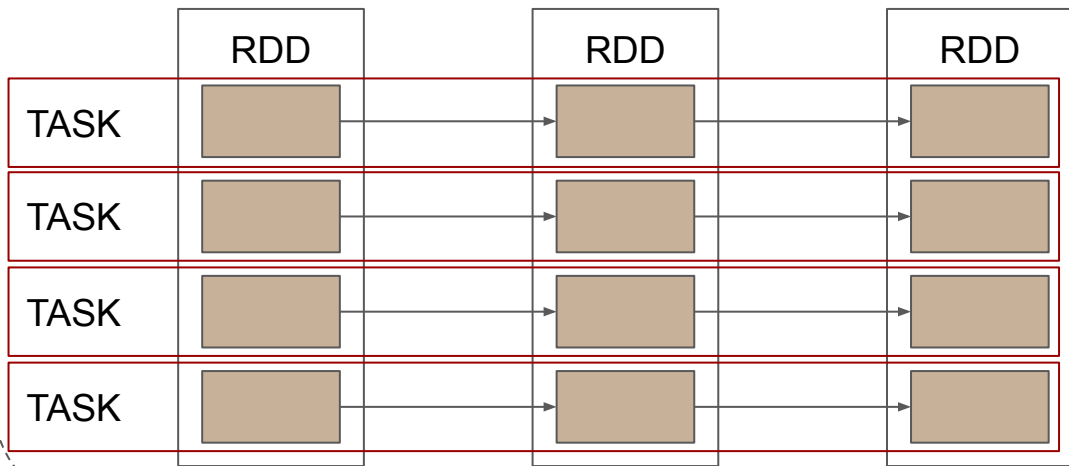
```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
  .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



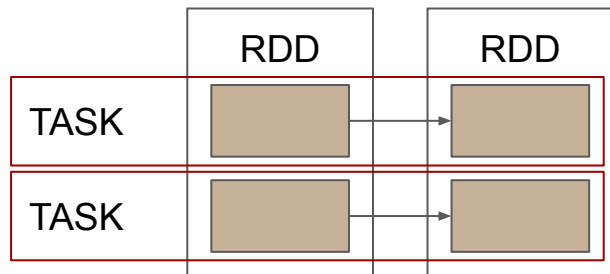
Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
  .mapValues { case (dataSum, cpt) => dataSum/cpt }
```

STAGE 0

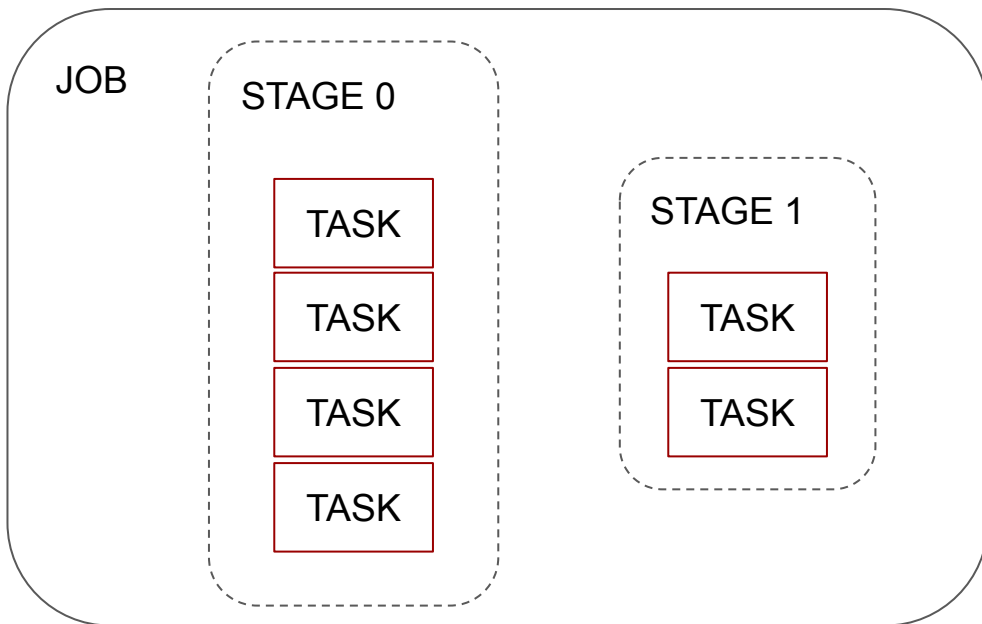


STAGE 1



Task and Stage

```
val wc = sc.textFile(myData)
  .map(_.split(","))
  .map(row => (row(0), (row(14), 1)))
  .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
  .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



| Vocabulaire

▼ Application

- ▼ Instance d'une SparkSession (correspondance 1:1)
- ▼ Ensemble de *jobs*

▼ Job

- ▼ Ensemble de transformations déclenchées par une action (point vue logique)
- ▼ Ensemble de stages (point vue pratique, selon le nombre de *shuffles* nécessaires)

| Vocabulaire

▼ Stage

- ▼ Un ensemble de Tasks qui sont exécutées en parallèle
- ▼ Un *shuffle* coupe un job en plusieurs *stages*
- ▼ Spark commence un stage après chaque *shuffle* et garde l'ordre d'exécution des stages pour arriver au résultat final (envoyé au *driver*)

▼ Task

- ▼ Traitement unitaire effectué sur un *exécuteur* sur une unité de données (partition)
- ▼ Le nombre de partition détermine le niveau de parallélisation

Spark UI: Jobs

[Jobs](#)[Stages](#)[Storage](#)[Environment](#)[Executors](#)[SQL](#)

Zeppelin application UI

Spark Jobs (?)

User: sagean

Total Uptime: 5,2 min

Scheduling Mode: FIFO

Completed Jobs: 1

▶ [Event Timeline](#)

Completed Jobs (1)

Job Id (Job Group) ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0 (zeppelin-2E18ZAS6F-20181228-173254_1752340159)	Started by: admin saveAsTextFile at <console>:28	2019/01/21 16:09:03	1 s	2/2	4/4

Spark UI: Stage

[Jobs](#)[Stages](#)[Storage](#)[Environment](#)[Executors](#)[SQL](#)

Zeppelin application UI

Details for Job 0

Status: SUCCEEDED

Job Group: zeppelin-2E18ZAS6F-20181228-173254_1752340159

Completed Stages: 2

▶ [Event Timeline](#)

▶ [DAG Visualization](#)

Completed Stages (2)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	Started by: admin saveAsTextFile at <console>:28 +details	2019/01/21 16:09:05	0,2 s	<div>2/2</div>		11.3 KB	11.3 KB	
0	Started by: admin map at <console>:25 +details	2019/01/21 16:09:04	1,0 s	<div>2/2</div>	14.7 KB			11.3 KB

Spark UI: Stage DAG

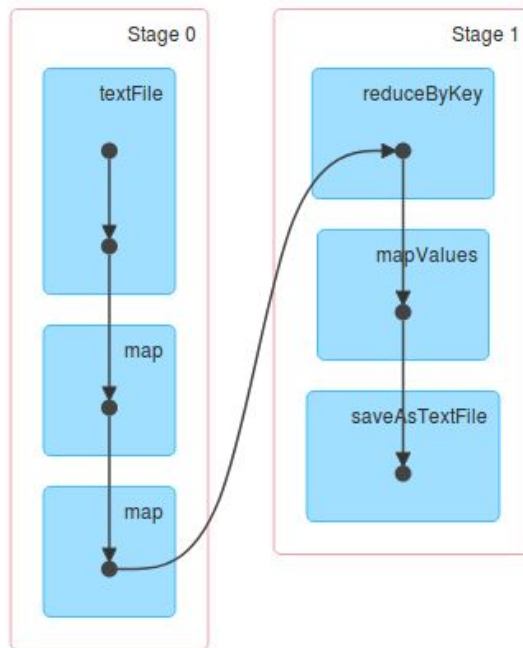
Details for Job 0

Status: SUCCEEDED

Job Group: zeppelin-2E18ZAS6F-20181228-173254_1752340159

Completed Stages: 2

- ▶ Event Timeline
- ▼ DAG Visualization



Spark UI: Tasks

 Jobs Stages Storage Environment Executors SQL

Zeppelin application UI

Details for Stage 1 (Attempt 0)

Total Time Across All Tasks: 0,3 s

Locality Level Summary: Any: 2

Output: 11.3 KB / 957

Shuffle Read: 11.3 KB / 977

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▶ Event Timeline

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0,1 s	0,1 s	0,2 s	0,2 s	0,2 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Output Size / Records	5.5 KB / 462	5.5 KB / 462	5.8 KB / 495	5.8 KB / 495	5.8 KB / 495
Shuffle Read Size / Records	5.5 KB / 470	5.5 KB / 470	5.8 KB / 507	5.8 KB / 507	5.8 KB / 507

▼ Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Output Size / Records	Shuffle Read Size / Records
driver	10.7.14.164:36674	0,4 s	2	0	0	2	11.3 KB / 957	11.3 KB / 977

Tasks (2)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Output Size / Records	Shuffle Read Size / Records	Errors
0	2	0	SUCCESS	ANY	driver / localhost	2019/01/21 16:09:05	0,1 s		5.8 KB / 495	5.8 KB / 507	
1	3	0	SUCCESS	ANY	driver / localhost	2019/01/21 16:09:05	0,2 s		5.5 KB / 462	5.5 KB / 470	

| Les points à retenir

- ▼ Spark est un **framework de calcul performant** qui tire parti d'une utilisation intensive de la mémoire
- ▼ La **SparkSession** est le point d'entrée unique aux fonctionnalités de Spark
- ▼ Les **Resilient Distributed Dataset (RDD)** sont les éléments de travail de base sous Spark
- ▼ On distingue deux types d'opérations avec des RDDs: les **transformations** et les **actions**

Spark DataFrame

| Création, Chargement & Sauvegarde
à partir de différentes sources

Spark DataFrame

Création, Chargement & Sauvegarde à partir de différentes sources

- ▼ Comprendre ce qu'est une **DataFrame**
- ▼ Créer des DataFrames à partir de données existantes
- ▼ Charger et sauvegarder des DataFrames dans différents formats
- ▼ Comprendre ce qu'est le format **parquet** et ses avantages
- ▼ Créer une DataFrame à partir de tables **Hive** et de **Bases de Données Relationnelles**

| DataFrame - Définition générale

- ▼ Structure de données immutable à deux dimensions avec des colonnes nommées
- ▼ Les colonnes peuvent être de types différents
- ▼ Peut être visualisée comme une feuille de calcul ou une table SQL
- ▼ Structure très fréquente dans de nombreux langages d'analyse de données
 - ▼ R -> data.frame
 - ▼ Python -> librairie pandas
 - ▼ Spark SQL -> DataFrame

ID	Name	Age
001	Alice	34
002	Bob	25
003	Chris	53

| Spark SQL

- ▼ Module de Spark pour le **processing de données structurées**
- ▼ Deux moyens
 - ▼ Via son abstraction appelée **DataFrame**
 - ▼ Via des **requêtes SQL** qui seront distribuées

| DataFrame (Spark SQL)

- ▼ Collection de données organisée en colonnes nommées
 - ▼ RDD de Rows organisé avec un Schema
- ▼ Abstraction et API inspirées de R et pandas (Python) pour manipuler des données structurées

| Création de DataFrame

- ▼ **createDataFrame** permet de créer une DataFrame depuis un RDD de tuple/list, une list ou une DataFrame Pandas
- ▼ Le paramètre ***schema*** permet d'associer un schéma aux données
- ▼ Si le schéma n'est pas fourni, la méthode tente d'inférer le schéma depuis les données
- ▼ **samplingRatio** permet de définir le ratio de lignes utilisées pour inférer le schéma

| Création à partir de RDDs (Python)

La création de DataFrame se fait très simplement si l'on possède un RDD de *list* et que l'on connaît le schéma à appliquer

```
rdd = spark.sparkContext.parallelize([('Alice', 22), ('Bob', 30)])  
df1 = spark.createDataFrame(rdd, ['name', 'age'])  
  
df1.collect()  
df1.dtypes
```

```
[Row(name=u'Alice', age=22), Row(name=u'Bob', age=30)]  
[('name', 'string'), ('age', 'bigint')]
```


| Création à partir de RDDs (Scala)

Plusieurs méthodes disponibles. ex: à partir d'un "Product"

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

val rdd = spark.sparkContext.parallelize(Seq(Person("Alice", 22), Person("Bob", 30)))

val df = spark.createDataFrame(rdd)
```

| Création à partir de RDDs (Python)

- ▼ On peut passer par un RDD de *Rows*, qui vient de l'API SchemaRDD

```
from pyspark.sql import Row

Person = Row('name', 'age')
person = rdd.map(lambda r: Person(*r))
df2 = spark.createDataFrame(person)
df2.collect()
```

```
[Row(name=u'Alice', age=22), Row(name=u'Bob', age=30)]

[('name', 'string'), ('age', 'bigint')]
```

| Création à partir de RDDs (Scala)

▼ ex: à partir d'un RDD de Row

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

val schema = StructType(Seq(
  StructField("name", StringType),
  StructField("age", IntegerType)
))

val rdd = spark.sparkContext.parallelize(Seq(Row("Alice", 22), Row("Bob", 30)))
val df = spark.createDataFrame(rdd, schema)
```

I Depuis Pandas

- ▼ Spark permet également de créer une DataFrame Spark depuis une DataFrame pandas

```
pandas_df = pd.DataFrame([('Alice', 22), ('Bob', 30)], columns=['name', 'age'])  
df3 = spark.createDataFrame(pandas_df)
```

- ▼ De la même manière on peut récupérer, en local, dans une DataFrame pandas le contenu d'un DataFrame Spark

```
pandas_df = df3.toPandas()
```

| Depuis Pandas

- ▼ Attention au volume de données à échanger
 - ▼ Une DataFrame Spark peut notamment être liée à des données à très forte volumétrie stockées sur HDFS
 - ▼ Une DataFrame Pandas doit tenir en mémoire (sur système de stockage non distribué)
 - ▼ Bien s'assurer en amont que le rapatriement vers le *driver* est réalisable

| Fichiers Parquet

- ▼ **Parquet** est un format de sérialisation dit “*Columnar*” open source
- ▼ Très bien intégré dans l'écosystème Hadoop et Spark
- ▼ Source de donnée par défaut pour Spark SQL
 - ▽ Peut être modifié en configurant *spark.sql.sources.default*

Parquet - Format "Columnar"

Représentation logique
des données

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

Stockage "ligne"

a1	b1	c1	a2	b2	c2	a3	b3	c3
----	----	----	----	----	----	----	----	----

Stockage "colonne"

a1	a2	a3	b1	b2	b3	c1	c2	c3
----	----	----	----	----	----	----	----	----

| Fichiers Parquet

- ▼ Les fichiers parquet portent **leur propre schéma**
 - ▼ On peut créer une DataFrame sans définir le schéma
- ▼ Optimisé pour la performance
 - ▼ Permet de **compresser** les données et de les traiter de manière efficace
 - ▼ Permet de lire uniquement une partie des données
 - “filter push-down”

Fichiers Parquet

- ▼ La lecture d'un fichier parquet se fait à l'aide de la méthode globale **read** et de son module **parquet**

```
df = spark.read.parquet('/data/titanic/')
```

```
val df = spark.read.parquet("/data/titanic/")
```

- ▼ L'écriture au format parquet se fait de manière similaire avec la méthode **write**

```
df.write.parquet('/data/titanic/')
```

```
df.write.parquet("/data/titanic/")
```

| Mode d'écriture

- ▼ On peut spécifier lors de l'écriture le comportement à adopter, si le répertoire de sortie existe déjà
 - ▼ **"error"** : Comportement par défaut dans le monde Hadoop. Le job échoue
 - ▼ **"overwrite"** : On remplace le répertoire existant
 - ▼ **"append"** : On ajoute les fichiers au répertoire
 - ▼ **"ignore"** : On ne sauvegarde pas les données

```
df.write.mode('append').parquet('/data/titanic/')
```

```
df.write.mode("append").parquet("/data/titanic/")
```

| Fichiers JSON

- ▼ Les méthodes **read** et **write** peuvent s'adapter à d'autres formats, tels que le JSON, qui est un format venant avec un schéma

```
df = spark.read.json('/data/titanic.json')  
df.write.json('/data/titanic/')
```

- ▼ La méthode **toJSON** permet de passer d'une DataFrame à un RDD de string

```
df.toJSON().collect()
```

```
[u'{"name":"Alice","age":22}',  
u'{"name":"Bob","age":30}']
```

| Fichiers JSON

- ▼ Attention, chaque ligne du fichier JSON doit contenir un objet JSON bien identifié
 - ▼ Un fichier JSON multi-ligne verra sa lecture échouer le plus souvent

Fichiers CSV

- ▼ Le reader csv est directement inclus à partir de la version 2.0 de Spark
 - ▼ Avant 2.0 : reader csv externe disponible sur le github de databricks

```
# SPARK 2.x
df = (spark
      .read
      .csv('my.csv', header=True, sep=";"))
```

```
# SPARK 2.x
df = spark
      .read
      .option("header", "true")
      .option("sep", ";")
      .csv("my.csv")
```

| Bases de Données Relationnelles

- ▼ Spark SQL permet la lecture de données depuis d'autres bases de données utilisant **JDBC**
- ▼ Il faut au préalable ajouter le driver JDBC au *classpath* Spark
- ▼ On dispose des options suivantes pour charger les tables
 - ▼ *url*: L'URL JDBC à laquelle se connecter
 - ▼ *dbtable*: La table JDBC à lire
 - ▼ *driver*: Le nom de la classe du driver JDBC
 - ▼ Autres options pour le partitionnement

| Autre Reader

- ▼ D'autres readers externes existent et peuvent être utilisés
 - ▼ Notamment
 - Elasticsearch
 - Cassandra
 - Kudu
 - ...

| Les points à retenir

- ▼ La méthode **createDataFrame** permet de créer des DataFrames à partir de différents types de données
- ▼ On peut passer facilement d'un **RDD** à une **DataFrame** si l'on connaît le schéma des données
- ▼ Le format **parquet** est particulièrement adapté aux DataFrames
- ▼ Les méthodes **read** et **write** permettent de charger et sauvegarder des DataFrames

Spark DataFrame

Prise en main

Spark DataFrame

Prise en main

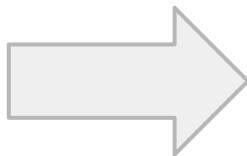
- ▼ Manipuler le **schéma** et les **colonnes** d'une DataFrame
- ▼ Manipuler les données grâce à des **Opérateurs** et des **Expressions**

Afficher le schéma d'une DataFrame

- ▼ L'API Dataframe permet d'afficher le schéma d'une DataFrame
 - ▼ Après chargement d'un fichier CSV

```
PassengerId,Survived,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,Fare,Cabin,Embarked  
1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S  
2,1,1,"Cumings, Mrs. John Bradley",female,38,1,0,PC 17599,71.2833,C85,C
```

```
df.printSchema()
```



```
root  
|-- PassengerId: string (nullable = true)  
|-- Survived: string (nullable = true)  
|-- Pclass: string (nullable = true)  
|-- Name: string (nullable = true)  
|-- Sex: string (nullable = true)  
|-- Age: string (nullable = true)  
|-- SibSp: string (nullable = true)  
|-- Parch: string (nullable = true)  
|-- Ticket: string (nullable = true)  
|-- Fare: string (nullable = true)  
|-- Cabin: string (nullable = true)  
|-- Embarked: string (nullable = true)
```

Récupérer le contenu d'une DataFrame

Spark permet de récupérer la totalité ou un sous ensemble d'une DataFrame en mémoire du driver dans une collection "classique".

```
# Ces méthodes retournent une liste de pyspark.sql.Row  
  
df.collect()  
df.take(10) #10 premiers éléments
```

```
import org.apache.spark.sql.Row  
  
val allContent: Array[Row] = df.collect()  
val firstTenRows: Array[Row] = df.take(10)
```

| Afficher le contenu d'une DataFrame

Spark permet d'afficher un sous ensemble d'une DataFrame

```
# Affiche les 10 premiers éléments  
df.show(10)  
  
# Par défaut affiche les 20 premiers éléments  
df.show()
```

| Projection d'un ensemble de colonnes

- ▼ Les méthodes **select** et **selectExpr** permettent de manipuler les colonnes d'une DataFrame
 - ▼ Pour filtrer les colonnes

```
df.select('PassengerId', 'Survived')
```

- ▼ Ou en créer de nouvelle à l'aide d'expression

```
df.selectExpr('Age < 18')
```

| Manipulation de colonnes

- ▼ L'API DataFrame permet une manipulation aisée des colonnes
 - ▼ Avec la classe **Column** (que l'on privilégie)
 - ▼ Ou sous forme d'expression textuelle
- ▼ On peut récupérer une instance de **Column** depuis une **DataFrame**

```
df.PassengerId  
df['PassengerId']
```

```
df("PassengerId")  
df.col("PassengerId")
```

| Manipulation de colonnes

- ▼ Il est possible de créer une colonne sans directement faire référence à une DataFrame.

```
from pyspark.sql.functions import col  
  
col('PassengerId')
```

```
import org.apache.spark.sql.functions.col  
import spark.implicits._  
  
col("PassengerId")  
$"PassengerId"
```


| Manipulation de colonnes

- ▼ On peut manipuler les colonnes à l'aide d'un ensemble de fonctions
 - ▼ de **comparaison** (like, rlike, isNull, isNotNull, between, when otherwise ...)
 - ▼ de **transformation de colonnes** (substr, pow...)
 - ▼ d'**expression** >, <, +, *, ...
 - ▼ de **coercion de type** (cast)
 - ▼ de **changement de nom** (alias)

| Manipulation de colonnes

- ▼ On peut utiliser ces différentes fonctions en entrée des différents opérateurs relationnels de l'API DataFrame

```
df.select(df.Age.cast('int').between(0, 17).alias('under_age'))
```

```
df.select(df("Age").cast(IntegerType).between(0, 17).alias("under_age"))
```

Spark SQL - Types simples

Spark	Python	Scala	
ByteType	int ou long	Byte	8 bits signés (-127 à 127)
ShortType	int ou long	Short	16 bits signés
IntegerType	int ou long	Int	32 bits signés
LongType	long	Long	64 bits signés
FloatType	float	Float	flotant 32 bits
DoubleType	float	Double	flotant 64 bits
DecimalType	decimal.Decimal	java.math.BigDecimal	
StringType	string	String	
BinaryType	bytearray	Array[Byte]	
BooleanType	Boolean	Boolean	
TimestampType	datetime.datetime	java.sql.Timestamp	
DateType	datetime.date	java.sql.Date	

Spark	Python	Scala	
ArrayType	list, tuple, or array	scala.collection.Seq	
MapType	dict	scala.collection.Map	
StructType	list or tuple	org.apache.spark.Row	

| Manipuler le schéma et les colonnes

- ▼ La modification du schéma d'une DataFrame se fait à travers un ensemble de fonctions permettant d'ajouter ou de supprimer des colonnes
- ▼ Ces fonctions **ne modifient pas les données**, elles doivent être adaptées au nouveau schéma

| Ajouter une colonne

- ▼ La fonction **withColumn** permet d'ajouter une colonne
- ▼ L'expression fournie à la fonction permet de lui donner une valeur par défaut ou de dériver sa valeur à partir d'une ou plusieurs autres colonnes
- ▼ **Rappel:** *Une DataFrame est immutable. L'ajout d'une colonne correspond en réalité à la création d'une nouvelle DataFrame avec une colonne supplémentaire*

| Ajouter une colonne

```
from pyspark.sql.functions import lit

df.withColumn('name', lit("defaultValue"))
df.withColumn('amount', lit(10000))
df.withColumn('tax', df.amount * 0.2)
df.withColumn('fromUdf', complexFunc(df.amount, df.tax))
```

```
import org.apache.spark.sql.functions.lit

df.withColumn("name", lit("defaultValue"))
df.withColumn("amount", lit(10000))
df.withColumn("tax", df("amount") * 0.2)
df.withColumn("fromUdf", complexFunc(df("amount"), df("tax")))
```

| Supprimer une colonne

- ▼ La fonction drop permet de supprimer une colonne

- ▼ Python

```
df.drop('colName')  
df.drop(df.colName)
```

- ▼ Scala

```
df.drop(colName = "colName")  
df.drop(df("colName"))
```


I Modifier le nom ou le type

- ▼ La fonction *alias* permet de modifier le nom
- ▼ La fonction *cast* permet de modifier le type
- ▼ La fonction *withColumnRenamed* permet de renommer une colonne

```
df = df.withColumnRenamed('colName', 'newName')  
df.select(df.Pclass.cast('int').alias('Classe'))
```

```
import spark.implicits._  
val newDf = df.withColumnRenamed("colName", "newName")  
newDf.select($"Pclass".cast(IntegerType).alias("Classe"))
```

| Filtrage des lignes

- ▼ La méthode **filter** permet de créer une nouvelle DataFrame ne contenant que les lignes pour lesquelles l'expression évaluée vaut **True**

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen ...	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. Joh...	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. ...	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	Futrelle, Mrs. Ja...	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. Willia...	male	35	0	0	373450	8.05		S

Filtrage des lignes

▼ Python

```
df.filter((df.Survived == 1) & (df.Embarked == 'C')).show()
```

▼ Scala

```
df.filter(($"Survived" === 1) and ($"Embarked" === 'C')).show()
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|PassengerId|Survived|Pclass|          Name|    Sex|Age|SibSp|Parch|          Ticket|    Fare|Cabin|Embarked|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|          2|        1|        1|Cumings, Mrs. Joh...|female| 38|    1|    0|          PC 17599|71.2833|  C85|        C|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

| Tri des données

- ▼ Les fonctions **orderBy** et **sort** permettent de trier les données d'une DataFrame en fonction d'une ou plusieurs colonnes

- ▼ Python

```
df.orderBy([df.Age, df.Fare], ascending=[0,1]).show(4)
```

- ▼ Scala

```
df.orderBy($"Age".desc, $"Fare".asc).show(4)
```

| Autres fonctions sur les colonnes

▼ Fonctions sur des colonnes de date

- ▼ `add_months(start, months)`
- ▼ `current_date()`
- ▼ `date_add(start, days)`
- ▼ `date_format(date, format)`
- ▼ `date_add(start, days), date_sub(start, days)`
- ▼ `datediff(end, start)`
- ▼ `dayofmonth(col)`
- ▼ `dayofyear(col)`
- ▼ `months_between(date1, date2)`
- ▼ `next_day(date, dayOfWeek)`
- ▼ `from_utc_timestamp(timestamp, tz)`
- ▼ `year(col), quarter(col), month(col), hour(col), minute(col), second(col)`

| Autres fonctions sur les colonnes

▼ Fonctions mathématiques

- ▼ `abs`, `avg`, `mean`, `min`, `max`, `stddev`, `variance`, `sqrt`, `cbrout`, `ceil`, `floor`, `exp`, `log`, `log10`, `log2`, `factorial`, `pow`
- ▼ `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`

▼ Fonctions de manipulation de collections

- ▼ `array(*cols)`, `array_contains(col, value)`, `sort_array(col)`
- ▼ `create_map(*cols)`, `size(col)`

▼ Fonctions sur des colonnes de strings

- ▼ `concat(*cols)`, `concat_ws(sep, *cols)`
- ▼ `format_string(format, *cols)`, `initcap(col)`
- ▼ `instr(col)`, `length(col)`, `substring(str, pos, len)`, `upper(col)`

| Les points à retenir

- ▼ Une DataFrame est une **collection de Rows** accompagnées d'un **Schema**
- ▼ Une DataFrame est sauvegardée de manière à être optimisée pour travailler sur les **colonnes**
- ▼ Les DataFrames se manipulent à l'aide d'**opérateurs relationnels** tels que *select*, *filter*, *groupBy* auxquels on fournit des **expressions**
- ▼ Il existe de nombreuses fonctions applicables aux colonnes d'une DataFrame
 - ▼ [pyspark.sql.functions](#)
 - ▼ [org.spark.sql.functions](#)

Spark DataFrame

Agrégations et jointures

Spark DataFrame

Agrégations et jointures

- ▼ Utiliser les différentes **fonctions d'agrégations** de DataFrames
- ▼ Utiliser les différents types de **jointures** entre DataFrames
- ▼ Faire l'**union** et le **split** de DataFrames
- ▼ Comprendre et utiliser les **Window Aggregations**

| Agrégations

- ▼ Les **agrégations** sont des opérations très courantes lorsque l'on souhaite manipuler la donnée
- ▼ La méthode **groupBy** permet de regrouper les données selon les différentes modalités de la colonne sélectionnée.
- ▼ Il convient ensuite d'appliquer d'autres fonctions sur ces agrégats qui sont du type **GroupedData**, telles que *count*, *sum* ou *avg*

| Agrégations

```
count_by_class = df.groupBy('Pclass').count()

count_by_class.printSchema()
count_by_class.show()
```

```
root
 |-- Pclass: string (nullable = true)
 |-- count: long (nullable = false)

+-----+-----+
|Pclass|count|
+-----+-----+
|      1|  216|
|      2|  184|
|      3|  491|
+-----+-----+
```

| Fonction d'agrégation

- ▼ La méthode `agg` permet de définir les opérations à effectuer sur les colonnes après un `groupBy`

```
import pyspark.sql.functions as F
agg_by_class = (df
    .groupBy("Pclass")
    .agg(
        F.count(df.Age).alias("count_by_age"),
        F.min(df.Age.cast("int")).alias("min_age"),
        F.max(df.Age.cast("int")).alias("max_age"),
        F.avg(df.Age.cast("int")).alias("avg_age")
    )
)
agg_by_class.show()
```

	Pclass	count_by_age	min_age	max_age	avg_age
	1	216	0	80	38.225806451612904
	2	184	0	70	29.85549132947977
	3	491	0	74	25.11549295774648

| Fonction d'agrégation

```
import org.apache.spark.types._
import org.apache.spark.sql.functions._

val agg_by_class = df.groupBy(df("Pclass"))
  .agg(
    count(df("Age")).alias("count_by_age"),
    min(df("Age").cast(IntegerType)).alias("min_age"),
    max(df("Age").cast(IntegerType)).alias("max_age"),
    avg(df("Age").cast(IntegerType)).alias("avg_age")
  )

agg_by_class.show()
```

| Fonctions d'agrégations

function	description
<code>avg(col), mean(col)</code>	Calcul de la valeur moyenne d'un groupe
<code>count(col)</code>	Compte le nombre d'item non null d'un groupe
<code>countDistinct(col, *cols)</code>	Compte le nombre d'item unique sur une ou plusieurs colonnes
<code>first(col), last(col)</code>	Première ou dernière valeur
<code>min(col), max(col)</code>	Valeur minimum ou maximum
<code>sum(col)</code>	Somme des valeurs d'un groupe
<code>sumDistinct(col)</code>	Somme des valeurs distinctes d'un groupe

| Combiner des datasets

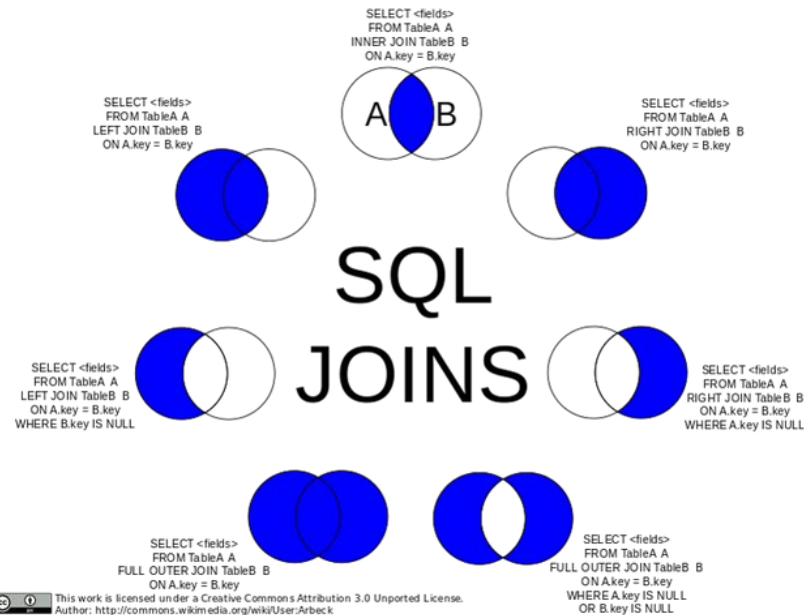
- ▼ Spark DataFrame fournit plusieurs outils permettant de combiner ou de séparer une ou plusieurs DataFrame
- ▼ Des **jointures** du même type que le SQL
- ▼ Manipulation de plusieurs DataFrame avec les **Union** et **Split**

Jointures

- Les types de jointures sont proches de celles existantes en SQL

- inner
- full_outer (synonymes : full, outer)
- left_outer (synonyme : left)
- right_outer (synonyme : right)
- left_semi
- left_anti

- On ne peut faire des jointures qu'entre deux DataFrames



Joitures

```
# Left outer join
df_with_agg = df.join(
    agg_by_class,
    df.Pclass == agg_by_class.Pclass,
    'left_outer'
)
df_with_agg.show(3)
```

```
val df_with_agg = df.join(
    agg_by_class,
    df("Pclass") === agg_by_class("Pclass"),
    "left_outer"
)
df_with_agg.show(3)
```

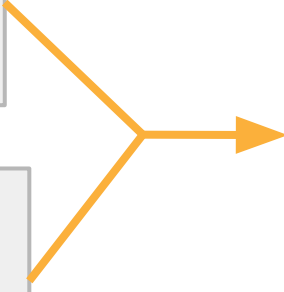
Pclass	Name	Sex	Age	Pclass	count_by_age	min_age	max_age	avg_age
1.0	Cumings, Mrs. Joh...	female	38.0	1.0	186	0	80	38.225806451612904
1.0	Futrelle, Mrs. Ja...	female	35.0	1.0	186	0	80	38.225806451612904
1.0	McCarthy, Mr. Tim...	male	54.0	1.0	186	0	80	38.225806451612904

| Jointures

- ▼ Par défaut, le schéma de la DataFrame résultant d'une jointure est la concaténation des deux schémas des DataFrames initiales
 - ▼ Les deux clés de la jointure sont présentes

```
root
|-- Pclass: double (nullable = true)
|-- Name: string (nullable = true)
|-- Sex: string (nullable = true)
|-- Age: double (nullable = true)
```

```
root
|-- Pclass: double (nullable = true)
|-- count_by_age: long (nullable = false)
|-- min_age: integer (nullable = true)
|-- max_age: integer (nullable = true)
|-- avg_age: double (nullable = true)
```



```
root
|-- Pclass: double (nullable = true)
|-- Name: string (nullable = true)
|-- Sex: string (nullable = true)
|-- Age: double (nullable = true)
|-- Pclass: double (nullable = true)
|-- count_by_age: long (nullable = false)
|-- min_age: integer (nullable = true)
|-- max_age: integer (nullable = true)
|-- avg_age: double (nullable = true)
```

| Jointures

- ▼ La DataFrame résultant garde cependant en mémoire l'origine de chaque colonne
- ▼ On peut alors supprimer une colonne en stipulant l'une ou l'autre DataFrame initiale

```
df_with_agg = df_with_agg.drop(agg_by_class.Pclass)
```

```
df_with_agg.show(3)
```

Name	Sex	Age	Pclass	count_by_age	min_age	max_age	avg_age
Cumings, Mrs. Joh...	female	38.0	1.0	186	0	80	38.225806451612904
Futrelle, Mrs. Ja...	female	35.0	1.0	186	0	80	38.225806451612904
McCarthy, Mr. Tim...	male	54.0	1.0	186	0	80	38.225806451612904

Raccourcis pour les inner join

- ▼ Dans le cas d'un inner join pour lequel le nom de la colonne de jointure est le même pour les deux DataFrames, l'écriture peut être allégée
 - ▼ Drop automatique de la colonne de la 2ème DataFrame
 - ▼ Inner implicitement appelé

```
df_with_agg = df.join(agg_by_class, "Pclass")  
  
df_with_agg.show(3)
```

Name	Sex	Age	Pclass	count_by_age	min_age	max_age	avg_age
Cumings, Mrs. Joh...	female	38.0	1.0	186	0	80	38.225806451612904
Futrelle, Mrs. Ja...	female	35.0	1.0	186	0	80	38.225806451612904
McCarthy, Mr. Tim...	male	54.0	1.0	186	0	80	38.225806451612904

| Jointures sur plusieurs colonnes

- ▼ Dans le cas d'une jointure sur plusieurs colonnes, il faut spécifier les expressions sur les colonnes sous la forme d'une expression globale

- ▼ Encapsuler le tout dans des parenthèses

```
# Left outer join
df1.join(
  df2,
  ((df1.col1 == df2.col1) & (df1.col2 == df2.col2)),
  'left_outer'
)
```

```
df1.join(
  df2,
  (df1("col1") === df2("col1")) and df1("col2") === df2("col2"),
  "left_outer"
)
```

Union

- ▼ **union** permet de retourner une DataFrame contenant l'**union des lignes** de deux DataFrames
- ▼ Les deux DataFrames doivent avoir le même nombre de colonnes.
- ▼ Si les noms de colonnes sont différents, ce sont les colonnes de la première DataFrame qui sont conservées

```
rdd = spark.sparkContext.parallelize([('Alice', 22), ('Bob', 30)])  
df1 = spark.createDataFrame(rdd, ['name', 'age'])  
  
rdd2 = spark.sparkContext.parallelize([('Chris', 31), ('David', 86)])  
df2 = spark.createDataFrame(rdd2, ['name_bis', 'age_bis'])  
  
df1.union(df2).show()
```

```
+-----+-----+  
|  name|age|  
+-----+-----+  
|Alice| 22|  
|  Bob| 30|  
|Chris| 31|  
|David| 86|  
+-----+-----+
```

| Split

- ▼ La méthode `randomSplit` permet de séparer une `DataFrame` en plusieurs `DataFrames` selon les proportions souhaitées
- ▼ L'assignation des Row à une `DataFrame` se fait aléatoirement

```
(df1, df2) = df.randomSplit([0.67, 0.33])
```

```
df1.show()
```

```
df2.show()
```

```
+-----+----+  
|  name|age|  
+-----+----+  
|Alice| 22|  
|  Bob| 30|  
|David| 86|  
+-----+----+
```

```
+-----+----+  
|  name|age|  
+-----+----+  
|Chris| 31|  
+-----+----+
```

| Les points à retenir

- ▼ Les agrégations se font à l'aide de la méthode **groupBy**
- ▼ La fonction **agg**, couplée à un **groupBy**, permet d'utiliser de nombreuses fonctions d'agrégations
- ▼ Plusieurs types de jointures sont disponibles sur les DataFrames, au sein de la fonction **join**
- ▼ Les **Window functions** permettent d'utiliser des fonctions d'agrégation et de les appliquer à

Datasets

Pour créer et manipuler les Datasets :

```
import fr.mycompany.travel

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

val ds: Dataset[Person] = spark.read.parquet("/path/to/file").as[Person]

ds.collect()
```

```
Array(Person('Alice',22), Person('Bob', 30), ...)
```

Spark DataFrame

User Defined Functions &
SQL sur Spark

Spark DataFrame

User Defined Functions & SQL sur Spark

- ▼ Créer et utiliser des **User Defined Functions** sur des DataFrames
- ▼ Comprendre les impacts de l'utilisation d'UDFs sur les performances

| UDF

- ▼ Spark permet de créer ses propres fonctions utilisables en tant qu'expression sur les colonnes utilisables:
 - ▼ avec l'API DataFrame (select, filter, withColumn, ...)
 - ▼ dans les requêtes SparkSQL

UDF (Python)

Il est important de spécifier le type de sortie.

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

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

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

```
+-----+-----+
|           Name|title|
+-----+-----+
|Braund, Mr. Owen ...| Mr.|
|Cumings, Mrs. Joh...| Mrs.|
|Heikkinen, Miss. ...|Miss.|
|Futrelle, Mrs. Ja...| Mrs.|
+-----+-----+
```

UDF (Python)

```
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_title = df.withColumn('title', extract_title_udf(df.Name))
df_title.select('Name', 'title').show(4)
```

```
+-----+-----+
|           Name|title|
+-----+-----+
|Braund, Mr. Owen ...| Mr.|
|Cumings, Mrs. Joh...| Mrs.|
|Heikkinen, Miss. ...|Miss.|
|Futrelle, Mrs. Ja...| Mrs.|
+-----+-----+
```

UDF (Python)

```
import pyspark.sql.functions as F

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

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

```
+-----+-----+
|              Name|title|
+-----+-----+
|Braund, Mr. Owen ...|  Mr.|
|Cumings, Mrs. Joh...| Mrs.|
|Heikkinen, Miss. ...|Miss.|
|Futrelle, Mrs. Ja...| Mrs.|
+-----+-----+
```

| Pandas UDF

- ▼ Depuis Spark 2.3
- ▼ Utilise Apache Arrow
- ▼ Deux types : Scalar et Grouped Map
- ▼ Plus rapide que les UDF car travail sur des partitions
- ▼ Reste plus lente que des UDF Scala

| Pandas UDF : Scalar

- ▼ Les séries d'entrées et de sorties doivent avoir la même taille
- ▼ Le DataFrame est découpé en plusieurs séries pandas
- ▼ La transformation doit être indépendante du découpage

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

# Use pandas_udf to define a Pandas UDF
@pandas_udf('double', PandasUDFType.SCALAR)
# Input/output are both a pandas.Series of doubles

def pandas_plus_one(v):
    return v + 1

df.withColumn('v2', pandas_plus_one(df.v))
```

| Pandas UDF : Grouped Map

- ▼ Applique la transformation à un groupe de donnée (Group By)
- ▼ L'entrée peut avoir une taille différente de la sortie
- ▼ L'entrée est découpée en plusieurs DataFrame Pandas (un pour chaque groupe)

```
from pyspark.sql.functions import pandas_udf, PandasUDFType


# Input/output are both a pandas.DataFrame


@pandas_udf(df.schema, PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
    return pdf.assign(v=pdf.v - pdf.v.mean())

df.groupby('id').apply(subtract_mean)
```

| UDF (Scala)

```
import org.apache.spark.sql.functions.udf
import df.sparkSession.implicits._

val title: String => String = _.split(" ")(1)
val extractTitle = udf(title)

df.select($"Name", extractTitle($"Name").alias("title")).show(4)
```

```
+-----+-----+
|           Name|title|
+-----+-----+
|Braund, Mr. Owen ...| Mr.|
|Cumings, Mrs. Joh...| Mrs.|
|Heikkinen, Miss. ...|Miss.|
|Futrelle, Mrs. Ja...| Mrs.|
+-----+-----+
```

| UDF et performance

- ▼ Les UDFs permettent de créer des fonctions custom non présentes dans l'API de base
- ▼ Cependant, en Python, leur utilisation dégrade les performances des calculs
 - ▼ Chaque appel d'une fonction Python nécessite d'échanger les données avec un **interpréteur Python externe à la JVM**
- ▼ Les performances des DataFrames sont quasi-équivalentes quel que soit les langages
 - ▼ Lors de l'utilisation d'UDFs, ce n'est plus le cas

| Les points à retenir

- ▼ Il est possible de créer des **UDFs** applicables sur des DataFrames ou utilisables lors de requêtes SQL
- ▼ (Python) Leur création est utile mais dégrade les performances de calcul

Spark DataFrame

Caching & Partition Discovery

- ▼ Sauvegarder des résultat intermédiaire en mémoire
- ▼ Utiliser le Partition Discovery
- ▼ Optimiser les performances d'une application

| Caching des données

- ▼ Afin d'optimiser les performances , une possibilité est de conserver des données en mémoire
 - ▼ Utile si la DataFrame est utilisé à plusieurs reprises
- ▼ La fonction **persist()** permet de spécifier différents niveaux de stockage
 - ▼ **cache()** est un alias de **persist()**
- ▼ La méthode **unpersist()** libère les données mises en cache
- ▼ Le cache est seulement une suggestion
- ▼ S'il n'y a plus assez de mémoire, les données cachées utilisées en dernier sont supprimées du cache.

| Cache : Storage Level

▼ MEMORY_ONLY

▼ MEMORY_AND_DISK (by default)

si la partition ne rentre pas en mémoire, elle est écrite sur le disque

▼ MEMORY_ONLY_SER, MEMORY_AND_DISK_SER

les données sérialisées prennent moins de place, mais augmentent le temps de calcul

▼ DISK_ONLY

stock les données sur le disque de la machine qui a fait le calcul

▼ MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc

réplication des données sur une autre exécuteur

data.parquet



name	town	age
		.



name	age

Exemple

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

data.parquet



name	town	age



name	age

age >= 35



name	age

age < 35



name	age

Exemple

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

data.parquet



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



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

age >= 35



name	age
hopper	42
...	...



472876

age < 35



name	age

Exemple

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

data.parquet



name	town	age



name	age

age >= 35



name	age

age < 35



name	age

Exemple

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

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

Exemple

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

data.parquet



name	town	age



name	age

Exemple

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

data.parquet



name	town	age



name	age

age >= 35



name	age

age < 35



name	age

Exemple

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

data.parquet



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



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

age >= 35



name	age
hopper	42
...	...



472876

age < 35



name	age

Exemple

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


data.parquet



name	town	age



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

age >= 35



name	age

age < 35



name	age
turing	21
...	...



563543

Exemple

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

Cache : Spark 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 : Bonne Pratique

- ▼ Le DataFrame est utilisé plus d'une fois
- ▼ Le calcul du DataFrame est coûteux (read + filter is not)
- ▼ Peut résoudre certains problèmes liés à catalyst
- ▼ Faire un `unpersist` seulement après une action et que le DataFrame n'est pas réutilisé

Partition Discovery

- ▼ Le **partitionnement de table** est une optimisation courante
 - ▼ Les données sont sauvegardées dans différents répertoires
 - ▼ Les valeurs des colonnes partitionnées sont encodées dans le chemin d'accès

```
path
└─ to
    └─ table
        ├── gender=male
        │   ├── ...
        │   ├── country=US
        │   │   └─ data.parquet
        │   ├── country=CN
        │   │   └─ data.parquet
        │   └─ ...
        └─ gender=female
            ├── ...
            ├── country=US
            │   └─ data.parquet
            ├── country=CN
            │   └─ data.parquet
            └─ ...
```

| Partition Discovery

- ▼ Le format de données **parquet** permet de découvrir et d'inférer l'information partitionnée automatiquement
- ▼ Les types des colonnes sont aussi automatiquement inférés
- ▼ Il suffit de donner le path/to/table à la fonction de chargement pour extraire automatiquement le partitionnement

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

Spark UI: Tasks

 Jobs Stages Storage Environment Executors SQL

Zeppelin application UI

Details for Stage 1 (Attempt 0)

Total Time Across All Tasks: 0,3 s

Locality Level Summary: Any: 2

Output: 11.3 KB / 957

Shuffle Read: 11.3 KB / 977

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▶ Event Timeline

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0,1 s	0,1 s	0,2 s	0,2 s	0,2 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Output Size / Records	5.5 KB / 462	5.5 KB / 462	5.8 KB / 495	5.8 KB / 495	5.8 KB / 495
Shuffle Read Size / Records	5.5 KB / 470	5.5 KB / 470	5.8 KB / 507	5.8 KB / 507	5.8 KB / 507

▼ Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Output Size / Records	Shuffle Read Size / Records
driver	10.7.14.164:36674	0,4 s	2	0	0	2	11.3 KB / 957	11.3 KB / 977

Tasks (2)

Index	▲ ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Output Size / Records	Shuffle Read Size / Records	Errors
0	2	0	SUCCESS	ANY	driver / localhost	2019/01/21 16:09:05	0,1 s		5.8 KB / 495	5.8 KB / 507	
1	3	0	SUCCESS	ANY	driver / localhost	2019/01/21 16:09:05	0,2 s		5.5 KB / 462	5.5 KB / 470	

| Les points à retenir

- ▼ L'utilisation du **cache** peut améliorer grandement les performances
- ▼ **spark** permet de faire du **partition discovery** en lecture de données
- ▼ La Spark UI fourni de nombreuse information pour détecter les problèmes de performance

| Les points à retenir

- ▼ L'API **DataFrame** de Spark permet de faire de nombreuses opérations sur des données structurées
- ▼ Son fonctionnement se rapproche des librairies Pandas de Python et des DataFrames de R
- ▼ Elle permet de gérer, en lecture et en écriture, de **nombreux formats de données**
- ▼ Son usage est relativement intuitif, avec la possibilité de travailler directement sur les **méthodes associées** ou bien d'utiliser des **requêtes SQL**
- ▼ Elle intègre aujourd'hui une bonne part des fonctionnalités du SQL et des DataFrames Pandas