

# Le Framework Spark

---

DARIO 5101A : Big Data Analytics avec Spark  
Thibaud Vienne - ESIEE Paris

# Fonctionnement du cours

- 11 heures de cours :
  - Big Data : Une introduction (2h)
  - Panorama de l'écosystème Hadoop (3h)
  - Le framework Spark (3h)
  - Machine learning avec Spark (3h)
- 15 heures de TP :
  - TP 1 : Tutorial + Comptage de mots avec Spark.
  - TP 2 : Analyse de logs Apache.
  - TP 3 : Prédiction de dates de sorties de chansons.
  - TP 4 : Classification de clics internet.
- Evaluation finale

# Sommaire

1. Le framework Spark
2. Spark par rapport au framework Hadoop
3. Concepts de base du Spark Core
4. Spark Core dans la pratique
5. Manipulation de dataframes avec Spark SQL
6. Annexes

# Le framework Spark

---

Le framework Spark  
Les fonctionnalités

# Le framework Spark

- Spark est un framework spécialisé dans l'analyse de données dans un environnement Big Data.
- Contrairement à des outils plus classiques comme Python ou R qui ne tournent que sur une seule machine, Spark est capable d'effectuer ses traitements sur un ensemble de machines. C'est donc une technologie dite Big Data.
- Le framework possède un modèle plus simple que Hadoop MapReduce ainsi que des performances bien supérieures qui le rendent un outil de choix pour les data scientists et data engineers.



# Le framework Spark : Historique

- **2009** : Naissance de Spark à l'AMPLab de l'université Berkeley (USA).
- **2013**: passage sous licence Apache.
- **2014**: Spark rejoint la liste des « *top-projects* » de la fondation Apache. La communauté s'agrandit et passe à plus de 460 développeurs partout dans le monde. Sortie de la version 1.0.
- **2015**: Intégration à Databricks.
- **2016** : Version 2.0



Matheo Zaharia (co-créateur de Spark)

# Le framework Spark : Objectifs

- Le framework Spark a pour objectif de faire suite à Hadoop MapReduce en proposant des traitements de 10 à 100x plus rapides.
- Pour atteindre ce niveau de performance, il part du principe que la mémoire RAM est très peu chère aujourd’hui. Il propose ainsi d’utiliser cette mémoire au maximum pour y stocker des données contrairement à HDFS/MapReduce qui les stocke sur disque.
- Il a également pour but d’être interopérable avec le framework Hadoop en fonctionnant avec HDFS et YARN.
- Il a pour but d’être transparent pour l’utilisateur.
- Au niveau de la scalabilité, Spark se révèle efficient jusqu’à des clusters de 8000 machines.
- Il a également pour but d’assurer la tolérance aux pannes.

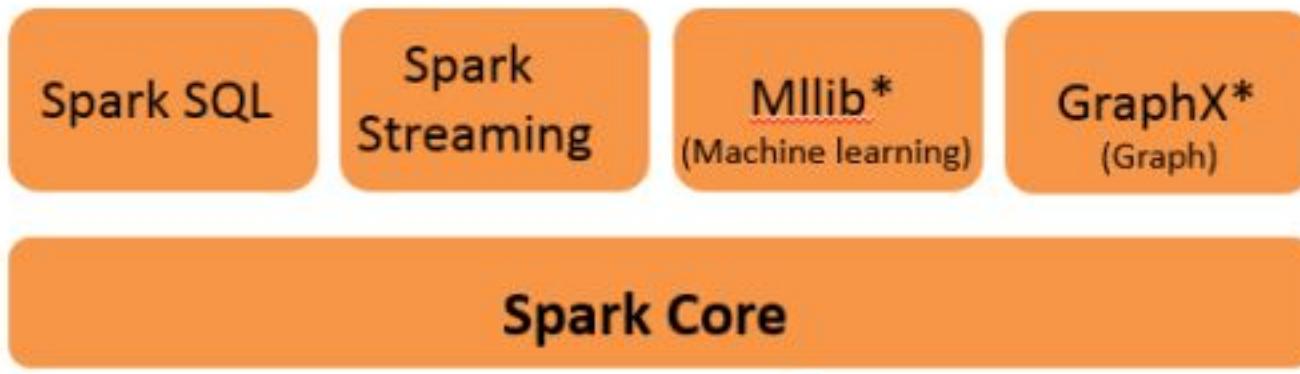
# Le framework Spark : les APIs

- Dans les dernières versions, il est possible d'interagir avec Spark au travers de 4 APIs.
- **Scala**: Langage avec lequel a été écrit Spark.
  - L'API la plus utilisée.
  - Possède l'ensemble des fonctionnalités.
  - Shell + Notebook
- **Python** : Le plus souvent utilisée par les data scientists.
  - Possède un bon nombre de fonctionnalités
  - Shell + Notebook
- **Java** : la classique, proche de Hadoop.
  - Possède un bon nombre de fonctionnalités mais peu utilisé.
- **R** : La plus récente.
  - Fonctionnalités pas encore toutes présentes.
  - Shell + Notebook



# Les fonctionnalités

- Le framework Spark possède plusieurs fonctionnalités établies selon des modules.
- **Le Spark Core:** c'est le système central de Spark. C'est une brique dédiée au traitement distribué des données (comme Hadoop MapReduce).
- **Les modules avancés:** Ils sont développés au-dessus de Spark Core et permettent de faire des traitements complexes (Streaming, machine learning, SQL...)



# Les fonctionnalités

- **Spark Core** : cette brique de traitement distribuée se base sur des algorithmes MapReduce. Elle est cependant plus performante que Hadoop MapReduce sur plusieurs critères:
  - Les traitements sont bien plus rapides que Hadoop MapReduce grâce à l'utilisation optimale de la mémoire.
  - La programmation est plus simple et plus fluide que sur Hadoop MapReduce.
- **Spark Streaming** : Cette brique profite de la puissance du Spark Core pour faire des traitements temps-réel et streaming.
- **Spark SQL**: Permet l'exploitation de dataframes ainsi que l'utilisation de requêtes SQL.
- **Spark GraphX** : brique dédiée au traitement et à la parallélisation de graphes.
- **Spark MLLib / ML** : implémentation puissante de Machine Learning basée sur le Spark Core / Spark SQL. Utilisé très fréquemment par les Data Scientists.

# Spark par rapport au framework Hadoop

---

Spark par rapport à Hadoop  
Spark vs Hadoop MapReduce  
Déploiement de Spark

# Spark par rapport à Hadoop

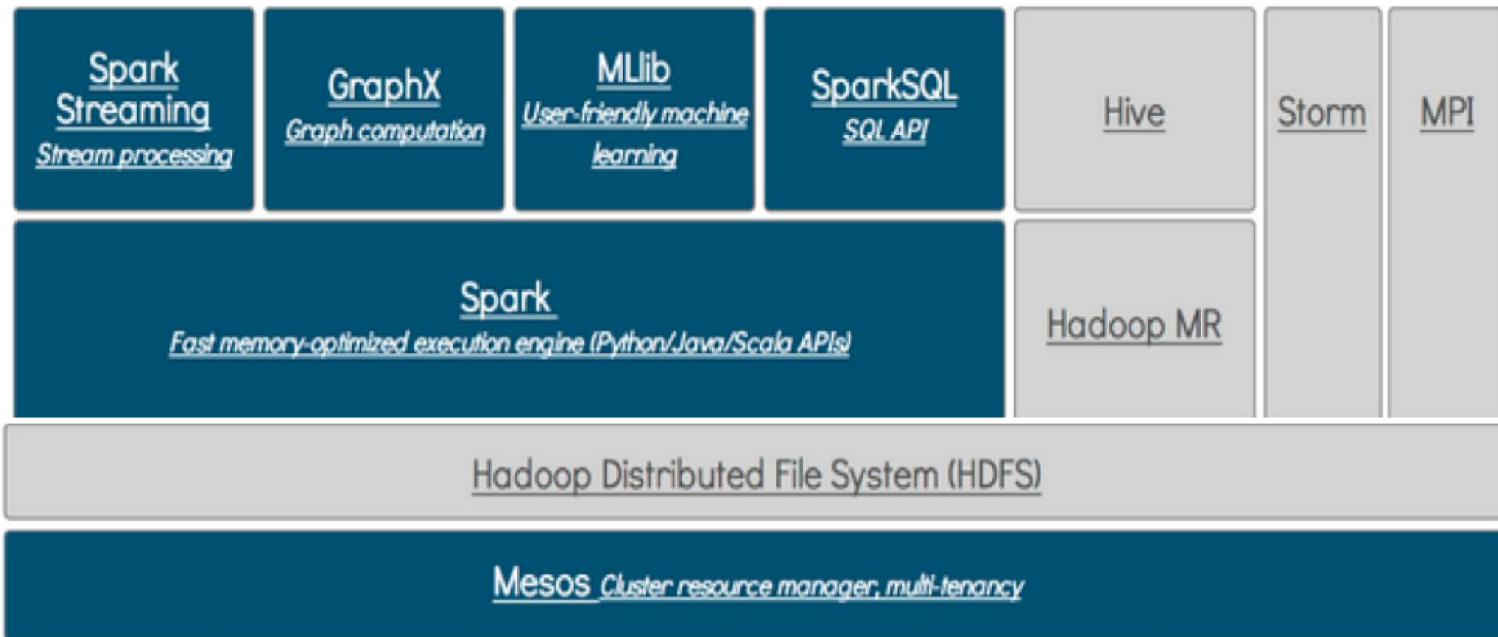
- Spark est un framework qui peut s'articuler avec le framework Hadoop. Il a évolué en même temps que Hadoop.
- **Hadoop version 1.0** : Spark se plaçait au-dessus de MapReduce qu'il utilisait alors pour réaliser des traitements sur les données.
- **Hadoop version 2.0** : Spark peut maintenant être placé à la place de MapReduce, en utilisant conjointement HDFS et la nouvelle brique d'allocations de ressources YARN.



# Spark par rapport à Hadoop

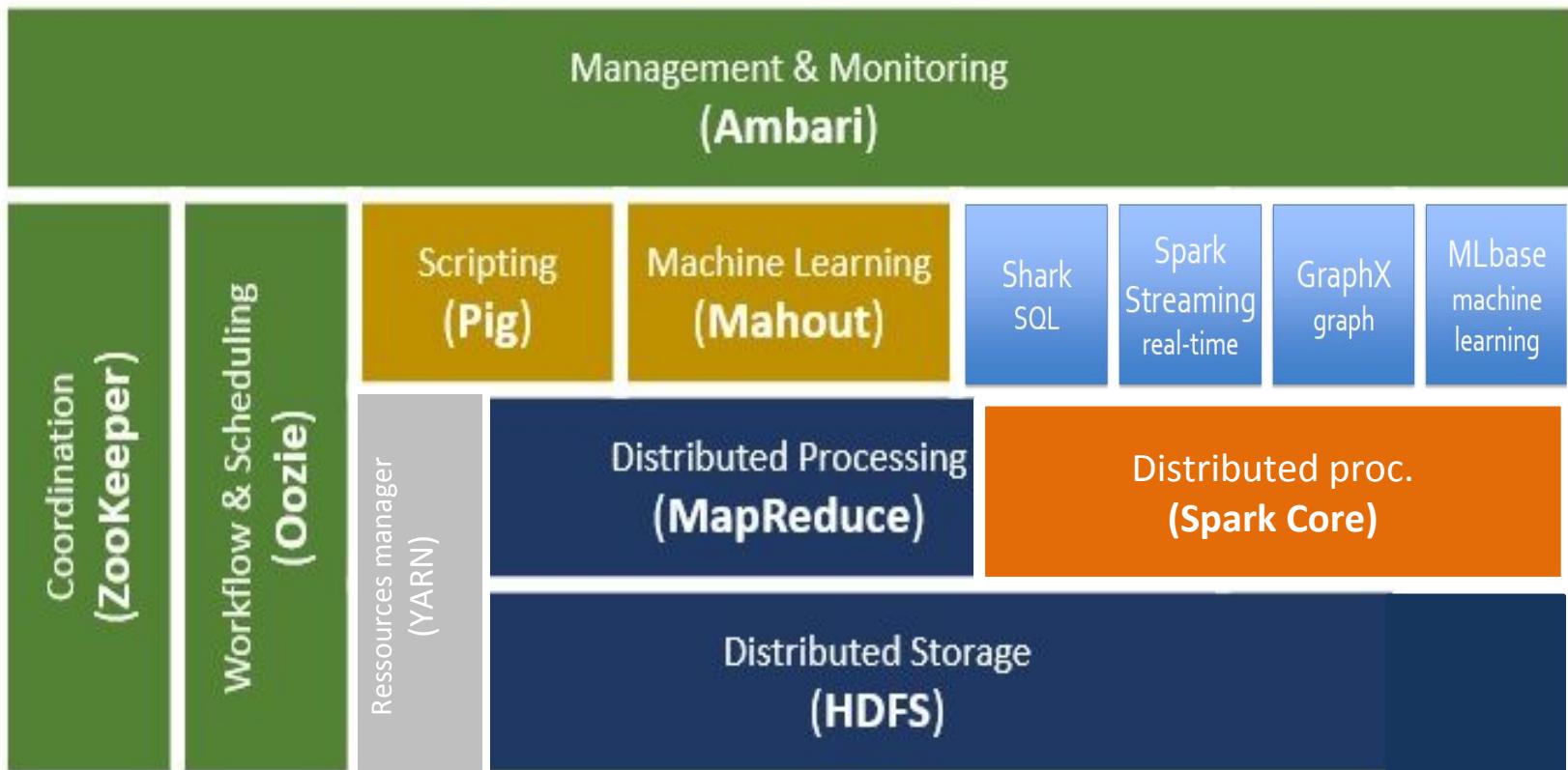
- Spark peut s'articuler parfaitement dans l'écosystème Hadoop.
- Pour fonctionner dans Hadoop, il a seulement besoin d'être associé à:
  - Un système de stockage distribué (HDFS, BigTable...)
  - Un gestionnaire de ressources (YARN, Mesos...)
- Avec l'arrivée de YARN dans Hadoop version 2, Spark est maintenant indépendant de Hadoop MapReduce pour la demande d'allocation des ressources. Il est donc maintenant alternatif à celui-ci.

# Spark par rapport à Hadoop



# Spark par rapport à Hadoop

- Certaines technologies (Hive, Mahout...) sont également capables de fonctionner au-dessus de Spark Core.
- **Attention:** Ce n'est pas le cas pour tous les modules (actuellement...).



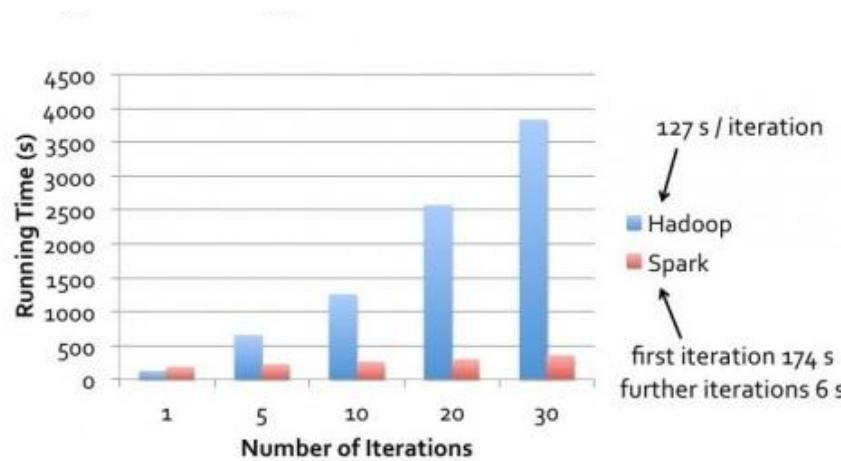
# Spark vs Hadoop MapReduce

- De manière générale, Spark Core est beaucoup plus efficace que Hadoop MapReduce pour le traitement et l'exploitation de donnée. Ses fonctionnalités surpassent facilement celles de Hadoop MapReduce dans la plupart des domaines.
- En effet, de par ses fonctionnalités et ses optimisations, Spark est capable de traiter la donnée de 10 à 100x plus rapidement que Hadoop MapReduce pour un même traitement et un même volume de données.
- Spark offre également de nombreuses fonctionnalités (machine learning, temps-réel...) et une facilité d'utilisation qui ont fait de ce framework un top-projet Apache incontournable pour le traitement de données.



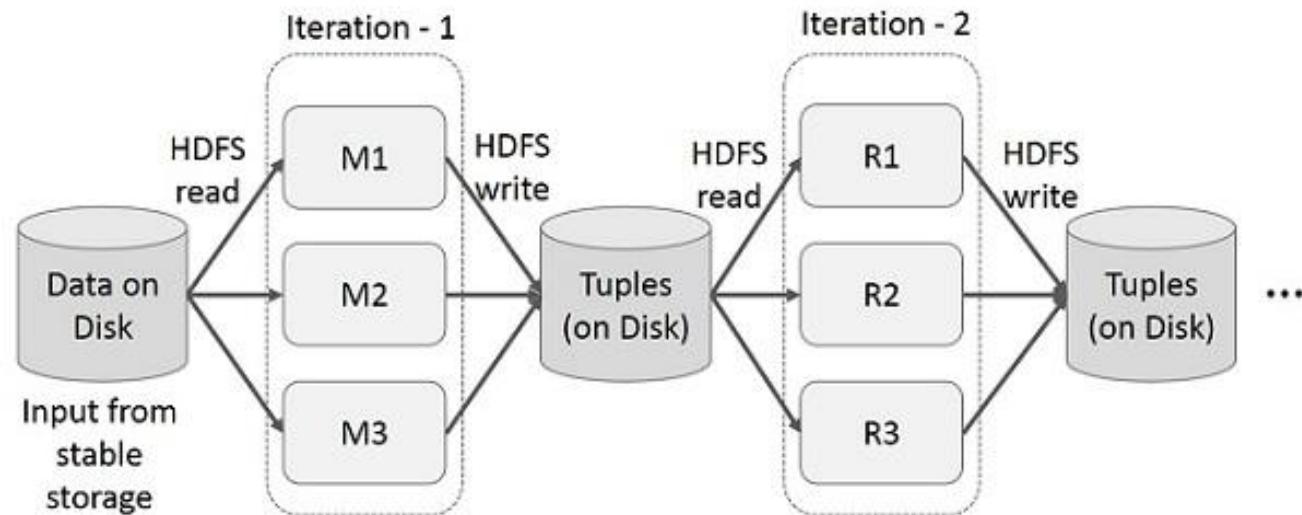
# Spark vs Hadoop MapReduce : Latence

- Concernant les temps de latence, La principale différence vient du fait que Spark est capable de garder les données en mémoire entre deux itérations de type MapReduce. A contrario, Hadoop MapReduce est obligé, entre deux itérations, d'écrire et de charger les données sur HDFS ( équivalent à un disque dur) avant de les traiter.
- Sur des calculs possédant beaucoup d'itérations, ces temps de chargement et d'écriture se révèlent être très significatifs et pénalisent fortement Hadoop MapReduce.
- Spark possède également des mécanismes d'optimisation internes appelés “lazy evaluation” qui permettent à Spark d'optimiser efficacement l'enchaînement de plusieurs opérations. Cela améliore ainsi ses performances. Voir la sous-partie “*Transformations et Actions*”.



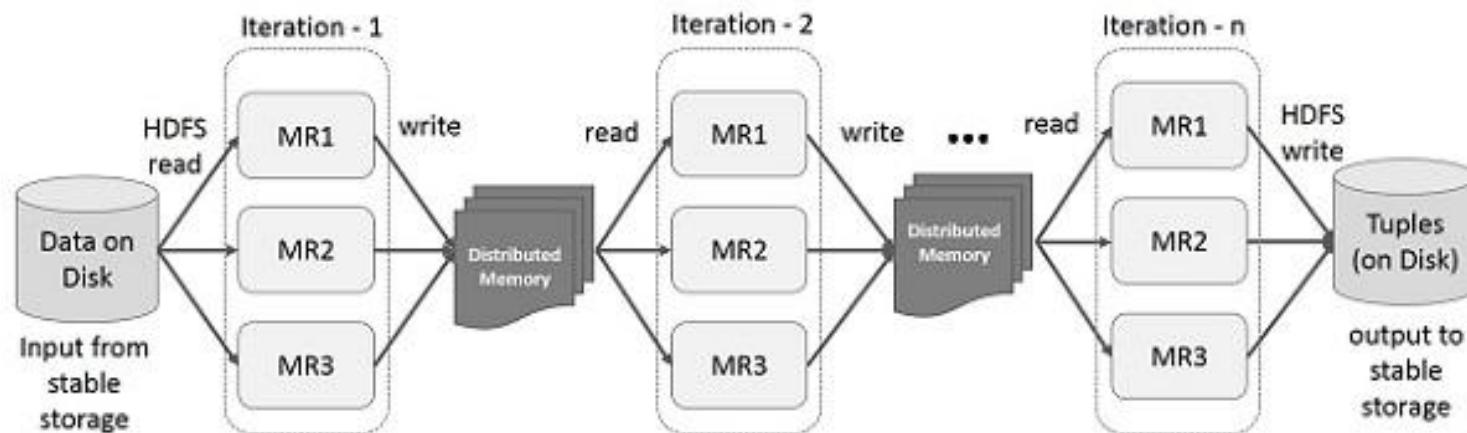
# Spark vs Hadoop MapReduce : Latence

- Sous **Hadoop MapReduce**, les données sont lues et écrites (sur disque) à chaque fois entre deux opérations de type MapReduce.
- Ces lectures et écritures dans HDFS successives rallongent significativement les temps de latence.



# Spark vs Hadoop MapReduce : Latence

- Avec **Spark**, le « stockage » en mémoire effectué entre plusieurs opérations est beaucoup plus rapide. Il est particulièrement efficace en terme de latences pour les traitements nécessitant de nombreuses itérations comme les algorithmes de machine learning.
- Dans le cas où le cluster ne possède pas de mémoire suffisante, les données sont écrites/lues sur disque de la même façon que pour Hadoop MapReduce.

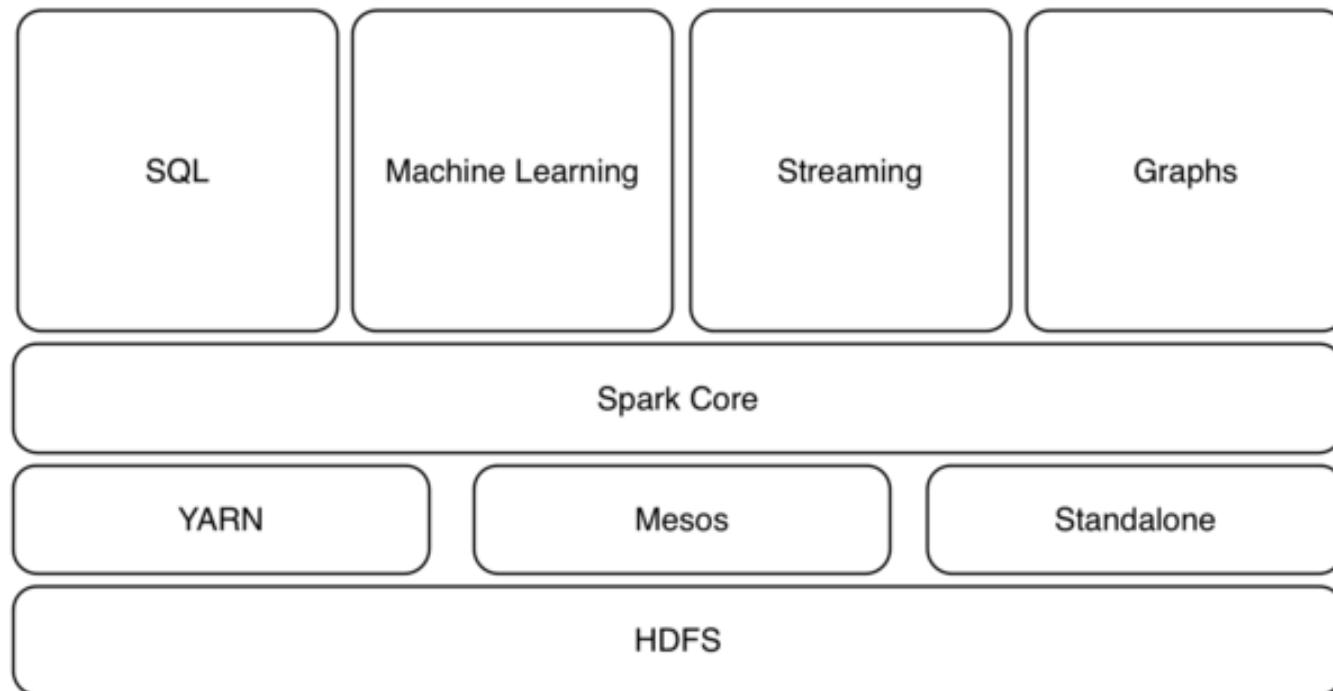


# Spark vs Hadoop MapReduce : Autres

	<b>Hadoop MapReduce</b>	<b>Spark</b>
<b>Fonctionnalités :</b>	<ul style="list-style-type: none"><li>- Traitement de données.</li><li>- “SQL” avec Hive.</li><li>- Machine learning avec Mahout.</li></ul>	<ul style="list-style-type: none"><li>- Traitement de données.</li><li>- “SQL” avec Spark SQL.</li><li>- Machine Learning avec Spark ML.</li><li>- Calcul de graphes avec Spark GraphX.</li></ul>
<b>Temps-réel :</b>	<ul style="list-style-type: none"><li>- Traitements batch.</li><li>- Pas de traitements streaming.</li></ul>	<ul style="list-style-type: none"><li>- Traitements batch.</li><li>- Traitement streaming avec Spark Streaming.</li></ul>
<b>Difficulté :</b>	<ul style="list-style-type: none"><li>- Multi-API grâce à Hadoop Streaming.</li><li>- Les algorithmes MapReduce doivent être écrits soi-même.</li></ul>	<ul style="list-style-type: none"><li>- Muti-API (Scala, Python, Java, R).</li><li>- Possède de nombreux opérateurs de haut niveau.</li></ul>
<b>Scalabilité :</b>	Très grande (plus grand cluster mondial = 14000 noeuds).	Très grande (plus grand cluster mondial = 8000 noeuds).
<b>Communauté :</b>	<ul style="list-style-type: none"><li>- Communauté passée sur Spark.</li></ul>	<ul style="list-style-type: none"><li>- Grande communauté.</li></ul>
<b>Divers :</b>	<ul style="list-style-type: none"><li>- Développé en java.</li><li>- Multi-OS.</li><li>- Licence 2 Apache.</li><li>- Haute tolérance aux pannes.</li><li>- Sécurité Kerberos (+++).</li></ul>	<ul style="list-style-type: none"><li>- Développé en Scala.</li><li>- Multi-OS.</li><li>- Licence 2 Apache.</li><li>- Hautes tolérances aux pannes.</li><li>- Sécurité Kerberos (+++).</li></ul>

# Déploiement de Spark

- Il existe plusieurs modes de déploiement pour Spark. Tant en local sur une machine que sur un cluster Hadoop.
- Dans le cas d'un déploiement sur cluster, il faut associer Spark avec:
  - Un système de stockage de fichiers distribués (HDFS, Hbase, BigTable...).
  - Un gestionnaire de ressources (YARN, mesos...).



# Déploiement de Spark

- Il existe quatre modes de déploiement de Spark sans / avec l'écosystème Hadoop.
- **Mode local** : Dans ce mode, tous les composants de Spark sont compris dans une seule JVM. Ce mode est pseudo-distribué (pas de distribution physique, mais multithread dans la JVM).
- **Mode Standalone**: c'est un mode distribué, très simple mais également peu performant. Il n'est pas adapté au cluster de plus de quelques machines.
- **Mode YARN** : Un mode utilisé pour les grands clusters. l'allocation des ressources est gérée par YARN, la brique d'allocation des ressources du framework Hadoop.
- **Mode Mesos** : Mesos est également un scheduler de ressources pour grand cluster. Crée également à Berkeley, il est maintenant sous fondation Apache.

# Concepts de base de Spark Core

---

Concepts de base  
Evaluation Lazy  
Architecture de Spark

# Concepts de base : Les RDDs

- Spark Core travaille avec des objets appelés RDDs (*Resilient Distributed Dataset*).
- Un RDD est une collection de records/observations distribués sur un ensemble de machines. Chaque bloc de données est appelée **une partition**.
- La distribution sur les différentes machines est là encore transparente.
- Les RDDs ne peuvent contenir que des objets de type python, scala ou java en fonction du langage de programmation (API) choisi par l'utilisateur.
- Les RDDs sont immutables.
- Les RDDs ne peuvent être créés que de deux façons:
  - Soit en chargeant des données depuis HDFS ou autre filesystem.
  - Soit en effectuant des transformations sur des RDDs déjà existants.

# Concepts de base : Les RDDs

- Les RDDs ne peuvent être créés que de deux façons:
  - Soit en chargeant des données distribuées depuis un système de stockage (file system).
  - Soit en effectuant des transformations sur des RDDs déjà existants.

HDFS (stockage distribué)

2016-01-03 ; 15:18 ; Paris  
2016-01-03 ; 15:18 ; Paris  
2016-05-08 ; 12:15 ; Lyon



Spark



Spark RDD

2016-01-03	15:18	Paris
2016-01-03	15:18	Paris
2016-05-08	12:15	Lyon

2016-01-01 ; 12:01 ; Paris  
2016-01-02 ; 14:08 ; Lyon  
2016-01-02 ; 15:03 ; Marseille



Spark



2016-01-01	12:01	Paris
2016-01-02	14:08	Lyon
2016-05-02	15:03	Marseille

2016-05-03 ; 12:01 ; Paris  
2016-05-02 ; 14:08 ; Lyon  
2016-06-03 ; 15:03 ; Lyon

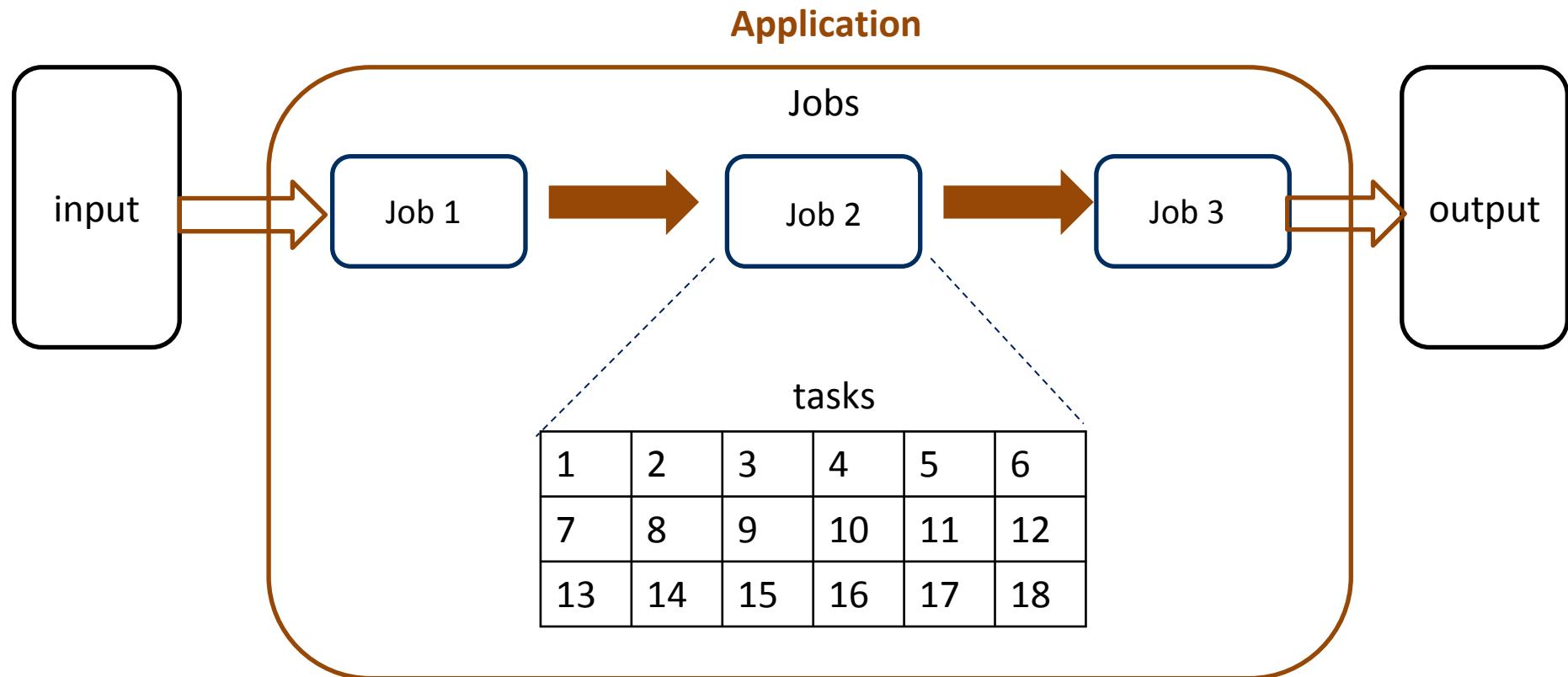


Spark



2016-05-03	12:01	Paris
2016-05-02	15:18	Lyon
2016-06-03	15:03	Lyon

# Concepts de base : Application Spark

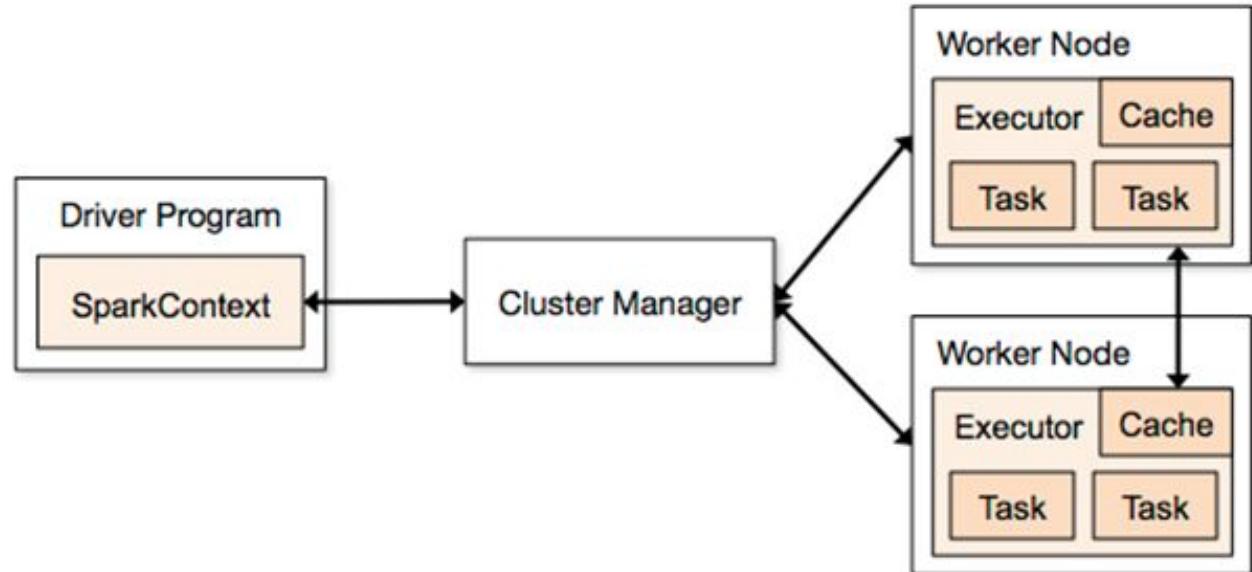


# Concepts de base : Application Spark

- **Application Spark** : une application Spark correspond à l'ensemble des Jobs effectués dans un programme. Depuis le lancement du programme (allocation des ressources) jusqu'à la fin de celui-ci (libération des ressources).
- **Job Spark** : Une application Spark est constituée d'un ensemble de jobs Spark. Chaque job Spark correspond à une “opération” réalisée dans Spark (un filtrage, un sélection, comptage de lignes, un algorithme MapReduce...). Un job Spark est lui-même la somme d'un ensemble de tâches. Lorsque toutes les tâches d'un job sont accomplies, le job est alors accompli.
- **Tâches Spark** : C'est le plus petit élément de travail pouvant et devant être exécuté par un exécuteur Spark. A chaque tâche correspond une partition du jeu de données. Dans le cas où l'on possède 50 partitions, le travail est alors découpé en 50 tâches et celles-ci sont alors traitées par parallèlement par l'ensemble des exécuteurs Spark présents dans le système.

# Architecture de Spark : Overview

- L'architecture Spark est assez analogue à l'architecture master/slave développée dans Hadoop MapReduce v1.
- Sous Spark, un programme exécuté est appelé une **application Spark**.
- L'architecture Spark est formé par 4 entités distinctes:
  - Le Driver.
  - Le master / cluster manager.
  - Les slaves / Workers
  - Les exécuteurs.

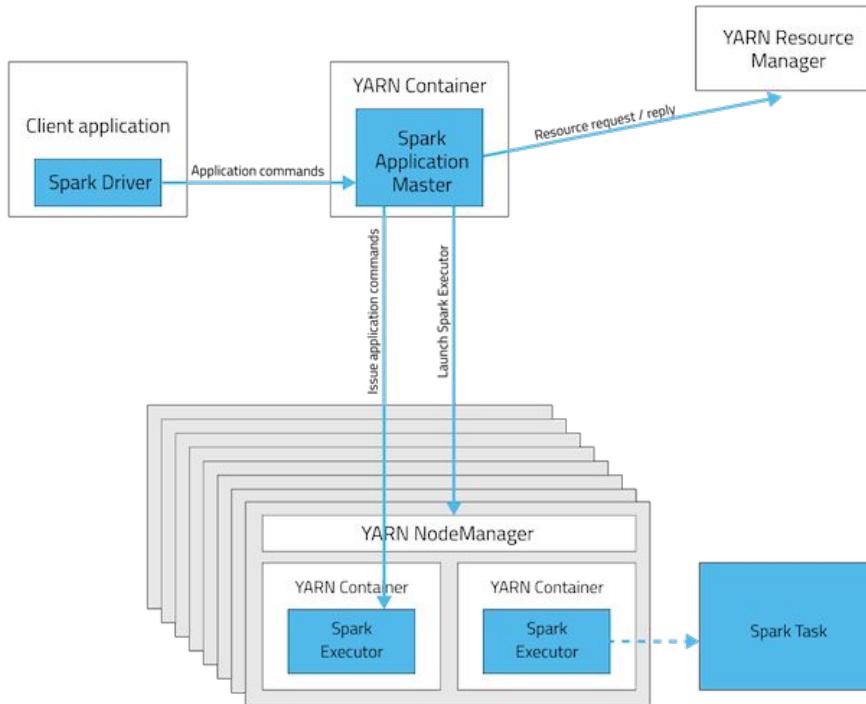


# Architecture de Spark : Driver

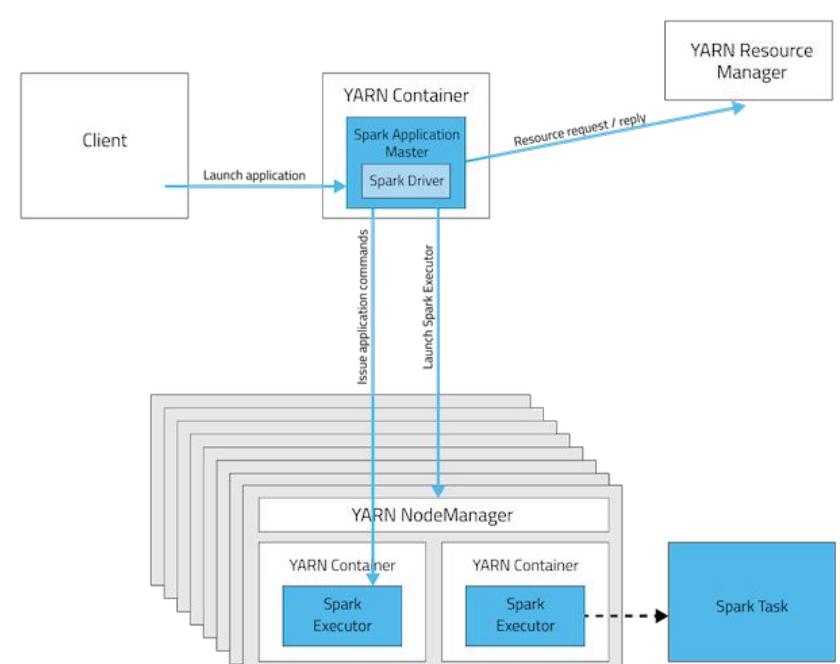
- Le Spark driver est une JVM à laquelle on a alloué des ressources (cpu, mémoire, cœurs etc...).
- Le driver est la machine qui lit le code main() et appelle les différentes opérations de l'application Spark. Il manage une application Spark.
- Le driver est la JVM qui possède un objet appelé le SparkContext. Cet objet est nécessaire au bon fonctionnement d'applications Spark.
- Il peut être déployé sur la machine du client lançant l'application (mode *yarn-client*) ou sur une machine dans le cluster (mode *yarn-cluster*). Dans ce dernier mode, celui-ci est bien souvent déployé sur le namenode.

# Architecture de Spark : mode client / cluster

Mode yarn-client



Mode yarn-cluster



# Architecture de Spark : Master

- Le Spark master, aussi appelé le cluster manager est chargé de négocier l'allocation de ressources nécessaires aux JVMs.
- Il se charge également de trouver de l'espace et des ressources pour faire tourner les workers et les exécuteurs.
- il y a un seul master par application Spark. Il peut être de l'un des trois types vus précédemment:
  - YARN.
  - Mesos.
  - Standalone.

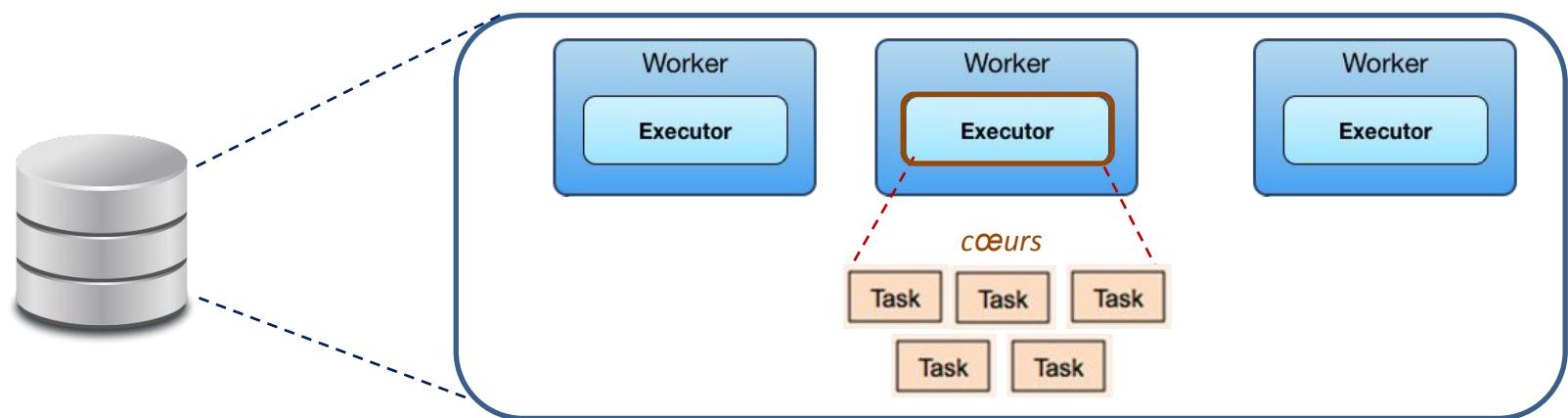
# Architecture de Spark : Worker

- Les Workers sont des JVMs auxquelles on a alloué des ressources (CPU, mémoire, coeurs etc...).
- Plus précisément, ce sont des conteneurs Spark dans lesquels vivent les exécuteurs. Sur un clusters Hadoop, les workers sont généralement présents dans les datanodes.



# Architecture de Spark : Exécuteur

- Les exéuteurs sont des agents situés dans les Workers et qui exécutent des « tasks ».
- Les exéuteurs sont dit multi-cœurs ou multithreads. Cela signifie que s'ils possèdent suffisamment de ressources, ils sont capable de se « partitionner » pour effectuer plusieurs tâches en parallèle (multithreading).
- Ces partitionnements sont appelés des « cores » (cœurs).



# Quiz

1. Nous utilisons Spark sur un cluster Hadoop afin de traiter un dataset volumineux. Un cluster Hadoop est constitué de 1 NameNode et 3 DataNodes. Dans notre programme Spark, nous allouons des ressources pour créer 12 exécuteurs de 4 cores chacun. Jusqu'à combien de tasks pouvons-nous traiter simultanément ?
2. Sachant maintenant que notre application Spark est composée de 3 jobs et que notre dataset possède 30 partitions. Au regard de la configuration décidée ci-dessus, qu'en pensez-vous / que préconisez-vous?

# Manipulation de RDDs avec Spark Core

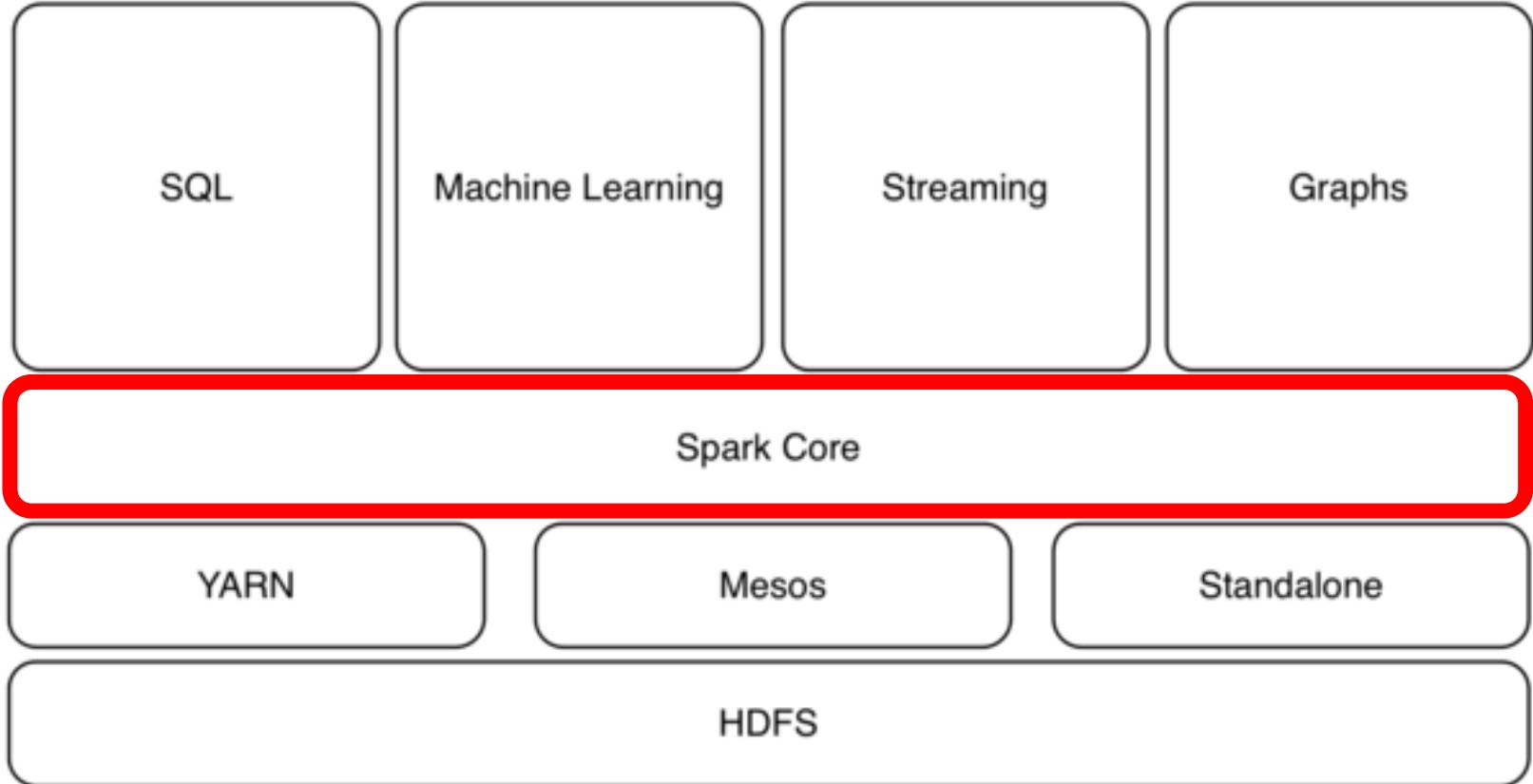
---

Création de SparkContext

Manipulations basiques

Transformations et actions

# Création de SparkContext



# Création de SparkContext

- En travaillant avec Spark Core, la première chose est de créer un objet appelé le « **SparkContext** ».
- Le SparkContext est le point d'entrée de toutes les fonctionnalités de Spark. Sans lui, aucun traitement ne peut être effectué.
- Le SparkContext permet de faire la connexion entre les différents composants qui composent le cluster ainsi que le cluster manager.
- Il est possible de créer un SparkContext de deux façons:
  - En créant un objet *SparkConf()* avec Python.
  - En l'appelant depuis le shell *PySpark*.

# Création de SparkContext

- Pour créer le SparkContext depuis Python, on construit d'abord un objet de type *SparkConf()* qui contient des informations à propos de notre application Spark.

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().set("master", "local[*]").set("spark.app.name", "DRIO")
sc = SparkContext(conf=conf)
```

- Il est possible de passer d'utiliser de nombreuses méthodes pour configurer l'objet *SparkConf()* :
  - setAppName(value)* / nom de l'application Spark
  - setMaster(value)* / L'URL du master-cluster manager
  - set("spark.executor.memory", "1g")* / mémoire allouée à chaque exécuteur en mode cluster
  - ...

# Création de SparkContext

- Pyspark est un Shell spécialement désigné pour travailler sur des programmes Spark écrits en python.
- La création du sparkContext se fait automatiquement et les informations et paramètres nécessaires sont passées en même que l'appel du shell.

```
~/spark-1.6.0-bin-hadoop2.6/bin/pyspark --master local[8] --driver-memory 12g --executor-memo  
ry 12g spark_example.py
```

# Manipulations basiques : Création des RDDs

- Il existe deux façons de charger des données dans Spark:
  - Créer soi-même ses données dans Spark et les paralléliser.
  - Charger un fichier de données depuis un *filesystem*.

```
# Création manuelle de RDD :  
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",  
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])  
  
# Depuis un fileSystem :  
rdd = sc.textFile("my_data.txt")
```

- Note** : En mode yarn-client / yarn-cluster, les données doivent bien souvent être distribuées sur HDFS afin de permettre le meilleur niveau de distribution et parallélisation des données.
- Note** : Spark prend en charge plusieurs types de fichiers dont les fichiers csv, avro, parquet, et json.

# Manipulations basiques : Collect

- Les RDDs étant des données distribuées, il n'est pas possible d'afficher ceux-ci avec des méthodes classiques comme des simple « *print* ».
- Heureusement, Spark propose des méthodes applicables aux RDDs permettant d'accéder/afficher nos données.
- Il existe pour cela plusieurs méthodes pouvant être utilisées:
  - La méthode « *.collect()* » qui rapatrie le RDD entier au user sous forme de liste python.
  - La méthode « *.take(n)* » qui récupère les « *n* » premiers éléments du RDD.

```
# Méthode .collect() :  
python_list = rdd.collect()  
print(python_list)  
  
# Méthode .take() :  
python_list_2 = rdd.take(2)  
print(python_list_2)  
  
['Paris,alimentation,19.00', 'Marseille,vetements,12.00', 'Paris,alimentation,8.00', 'Paris,vetements,15.00', 'Marsei  
lle,alimentation,20.00', 'Lyon,mediatheque,10.00']  
['Paris,alimentation,19.00', 'Marseille,vetements,12.00']
```

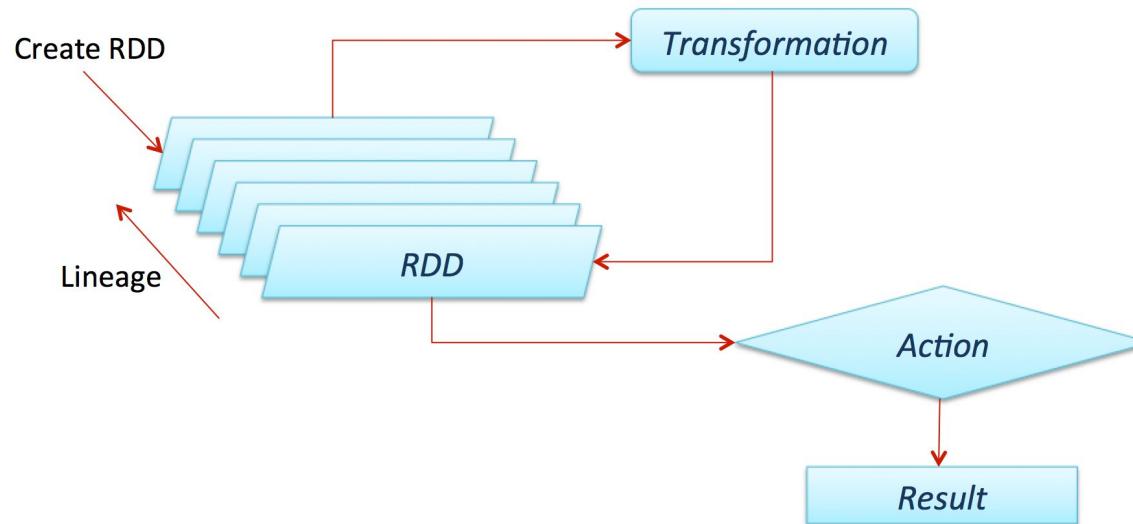
# Manipulations basiques : Sauvegarde de RDDs

- Avec Spark, les RDDs peuvent être sauvegardés sur HDFS ou sur le filesystem local.
- Dans le cas de HDFS, les fichiers sont alors distribués automatiquement entre les différents noeuds du cluster Hadoop.
- Un exemple de sauvegarde de fichier texte en local :

```
# Save dataset to text :  
rdd.saveAsTextFile("mon_rdd.txt")
```

# Transformations et actions

- Il est possible de réaliser deux types de méthodes sur les Sparks RDDS:
  - Les transformations
  - Les Actions
- Les transformations sont des méthodes qui retournent un nouveau RDD.
- Les Actions sont des méthodes qui retournent une valeur ou un résultat depuis le RDD en entrée.



# Transformations et actions

## TRANSFORMATIONS

### General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

### Math / Statistical

- sample
- randomSplit

### Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

### Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

## ACTIONS



- reduce
- collect
- aggregate
- fold
- first
- take
- foreach
- top
- treeAggregate
- treeReduce
- foreachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

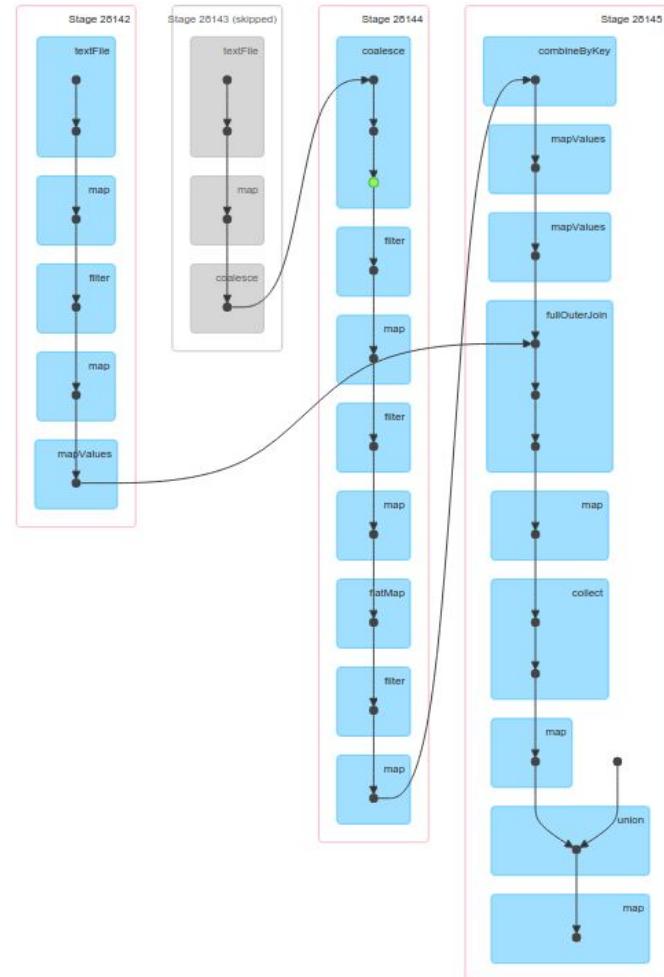
- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

# Transformations et actions : Evaluation lazy

- Les **transformations** sous Spark sont dites lazy (paresseuses). Cela signifie que lorsque Spark lit dans le code une **transformation**, il ne l'exécute pas de suite. A l'appel de chaque transformation, celle-ci est enregistrée et ajoutée à un DAG (Direct Acyclic Graph).
- Les **transformations** seront exécutées par Spark seulement lorsqu'il sera demandé un résultat par l'utilisateur. Cela se fera donc en présence d'une **action!**
- L'évaluation Lazy offre plusieurs avantages à Spark, notamment :
  - La possibilité de pouvoir étaler son code sur plusieurs lignes sans perte de performances.
  - Moins de lectures et écritures, les opérations sont effectuées en une seule fois.
  - Moins d'allers et retours d'instructions entre le driver et le cluster.
  - Réduction de la complexité et optimisation interne.
- **Note:** Dans le cas rare où on préfère qu'une transformation soit exécutée de suite, on peut utiliser la méthode « `.persist()` ».

# Transformations et actions : Evaluation lazy

- Ci-joint, un exemple de DAG (Direct Acyclic Graph) associé à un Job Spark.
- Le DAG se construit au fur et à mesure des transformations que l'on introduit dans le script Spark.
- Lors de l'appel d'une action, le graphe est alors optimisé, compilé puis envoyé au Task Scheduler de Spark.
- L'ensemble des tâches est alors exécuté et nous obtenons le résultat de notre action.



# Transformations

- Le Spark Core contient de nombreuses transformations qui prennent généralement appui sur des algorithmes de type MapReduce.
- Les transformations sont des méthodes qui retournent un nouveau RDD en sortie.
- Pour plus de lisibilité et de fluidité, on programme souvent avec des fonctions lambda.

Fonction	description
<b>map(func)</b>	Retourne un RDD dont chaque élément a été transformée par une fonction.
<b>Flatmap(func)</b>	Similaire à « map ». Cependant, les éléments du RDD sont « aplatis ».
<b>filter(func)</b>	Effectue un filtrage sur les lignes selon une condition booléenne.
<b>union()</b>	Réalise l'union de deux RDDs en un seul.
<b>groupByKey()</b>	Groupe des tuples (clé, valeur) par clés.
<b>ReduceByKey()</b>	Effectue une fonction sur les valeur d'un tuple (clé, valeur)
...	...

# Transformations : Map

- La fonction *Map()* transforme chaque ligne du RDD en une nouvelle ligne définie grâce à l'utilisation d'une fonction.
- En général, pour des mapping simples, on utilise une fonction lambda pour plus de fluidité.

```
# Creating RDD
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
print(rdd.collect())
```

```
# Mapping
new_rdd = rdd.map(lambda row: row.split(","))
```

```
# Show rdd
print(new_rdd.collect())
```

**rdd**

```
['Paris,alimentation,19.00', 'Marseille,vetements,12.00', 'Paris,alimentation,8.00', 'Paris,vetements,15.00', 'Marsei
lle,alimentation,20.00', 'Lyon,mediatheque,10.00']
```



**new\_rdd**

```
[['Paris', 'alimentation', '19.00'], ['Marseille', 'vetements', '12.00'], ['Paris', 'alimentation', '8.00'], ['Paris'
, 'vetements', '15.00'], ['Marseille', 'alimentation', '20.00'], ['Lyon', 'mediatheque', '10.00']]
```

# Transformations : FlatMap

- Le *flatmap()* est une méthode proche du mapping. Cependant, le flatmap « aplatis » les données. Il n'ajoute pas de « dimension » au RDD.
- Cela donne sur l'exemple précédent:

```
# Creating RDD
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
print(rdd.collect())

# Mapping
new_rdd = rdd.flatMap(lambda row: row.split(","))

# Show rdd
print(new_rdd.collect())
```

rdd

```
['Paris,alimentation,19.00', 'Marseille,vetements,12.00', 'Paris,alimentation,8.00', 'Paris,vetements,15.00', 'Marsei
lle,alimentation,20.00', 'Lyon,mediatheque,10.00']
```



new\_rdd

```
['Paris', 'alimentation', '19.00', 'Marseille', 'vetements', '12.00', 'Paris', 'alimentation', '8.00', 'Paris', 'vete
ments', '15.00', 'Marseille', 'alimentation', '20.00', 'Lyon', 'mediatheque', '10.00']
```

# Transformations : Filter

- La méthode « *filter()* » effectue un filtrage sur chaque élément du RDD d'après une condition booléenne.

```
# Creating RDD
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
print(rdd.collect())

# Mapping
rdd = rdd.map(lambda row: row.split(","))
new_rdd = rdd.filter(lambda row: "Paris" in row[0])

# Show rdd
print(new_rdd.collect())
```

rdd

```
['Paris,alimentation,19.00', 'Marseille,vetements,12.00', 'Paris,alimentation,8.00', 'Paris,vetements,15.00', 'Marsei
lle,alimentation,20.00', 'Lyon,mediatheque,10.00']
```



new\_rdd

```
[['Paris', 'alimentation', '19.00'], ['Paris', 'alimentation', '8.00'], ['Paris', 'vetements', '15.00']]
```

# Transformations : GroupByKey

- La méthode « groupByKey » groupe les éléments d'un RDD selon leur clé.
- Il est nécessaire d'appliquer la méthode sur un RDD sous forme d'un tuple <clé, valeur>. Il faut ensuite y associer une méthode .mapValues() afin de grouper les valeurs.

```
# Creating RDD
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
print(rdd.collect())

# Convert to <key, tuple>
rdd = rdd.map(lambda row: row.split(","))
key_tuple_rdd = rdd.map(lambda row: (row[0], row[2]))

# Group by key
new_rdd = key_tuple_rdd.groupByKey().mapValues(list)

# Show rdd
print(new_rdd.collect())
```

rdd

```
['Paris,alimentation,19.00', 'Marseille,vetements,12.00', 'Paris,alimentation,8.00', 'Paris,vetements,15.00', 'Marsei
lle,alimentation,20.00', 'Lyon,mediatheque,10.00']
```



new\_rdd

```
[('Marseille', ['12.00', '20.00']), ('Paris', ['19.00', '8.00', '15.00']), ('Lyon', ['10.00'])]
```

# Transformations : SortBy

- La méthode « SortBy » trie les éléments d'un rdd selon une “colonne” et un sens prédéfinis.

```
# Creating RDD
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
print(rdd.collect())

# Sort by
rdd = rdd.map(lambda row: row.split(","))
new_rdd = rdd.sortBy(lambda row: float(row[2]), ascending=True)

# Show rdd
print(new_rdd.collect())
```

rdd

```
['Paris,alimentation,19.00', 'Marseille,vetements,12.00', 'Paris,alimentation,8.00', 'Paris,vetements,15.00', 'Marsei
lle,alimentation,20.00', 'Lyon,mediatheque,10.00']
```



new\_rdd

```
[['Paris', 'alimentation', '8.00'], ['Lyon', 'mediatheque', '10.00'], ['Marseille', 'vetements', '12.00'], ['Paris',
'vetements', '15.00'], ['Paris', 'alimentation', '19.00'], ['Marseille', 'alimentation', '20.00']]
```

# Actions

- Les actions sont des méthodes qui retournent des résultats en sortie. Cela peut être des nombres, des listes, des tuples...
- Là encore, on utilise souvent des fonctions lambda pour programmer les actions. Cela permet plus de lisibilité lorsque de nombreuses transformations et actions sont enchainées.

Méthode	description
<b>Reduce()</b>	Agrège les éléments d'un RDD à partir d'une fonction.
<b>Collect()</b>	Retourne au driver le RDD entier sous forme de liste. Souvent utile pour vérifier un filtrage sur un petit jeu de données.
<b>Take(n)</b>	Retourne les « n » premiers éléments du RDD.
<b>Count()</b>	retourne le nombre d'éléments du RDD.
<b>saveAsTextFile()</b>	Sauvegarde d'un RDD sur un système de stockage distribué.
...	...

# Actions : Count

- L'action « count() » est une méthode qui permet de compter le nombre d'éléments dans un RDD.

```
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                      "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
# count
elements_count = rdd.count()
print(elements_count)
```

- **Question :** Combien valent « count\_1 » et « count\_2 » ?

```
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                      "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])
# count
count_1 = rdd.count()
print(count_1)

# mapping the comma
new_rdd = rdd.map(lambda row: row.split(","))
count_2 = new_rdd.count()
print(count_2)
```

# Actions : Reduce

- L'action « *Reduce* » est une méthode qui agrège les éléments d'un RDD à partir d'une fonction.
- **Note:** Le résultat est retourné en local à l'utilisateur. Ce n'est pas donc pas un RDD ou autre collection distribuée.

```
# Reducing
rdd = sc.parallelize([5, 6, 7])
rdd_sum = rdd.reduce(lambda a, b: a + b)
print(rdd_sum)
```

# Transformations et actions : Remarques

- Généralement, lorsqu'on programme avec Spark, on n'hésite pas à enchaîner les transformations et les actions au sein d'une même ligne de code.
- Cela n'a aucune incidence sur les performances de Spark. En effet, comme nous avons vu, Spark est « lazy ».
- Cela veut dire qu'il n'exécute les transformations que lorsqu'il détecte une action et donc un résultat demandé par l'utilisateur.

```
rdd = sc.parallelize(["Paris,alimentation,19.00", "Marseille,vetements,12.00", "Paris,alimentation,8.00",
                     "Paris,vetements,15.00", "Marseille,alimentation,20.00", "Lyon,mediatheque,10.00"])

results = rdd.map(lambda row: row.split(",")).map(lambda row: [row[0], float(row[2])])\ 
    .groupByKey().mapValues(list).map(lambda row: [row[0], sum(row[1])])\ 
    .filter(lambda row: "Paris" in row[0]).collect()
```

- **Questions:**
  - Quel est le type de la variable « results » ?
  - Que vaut « results » ?
  - Peut-on améliorer le code ?

# Live Coding !

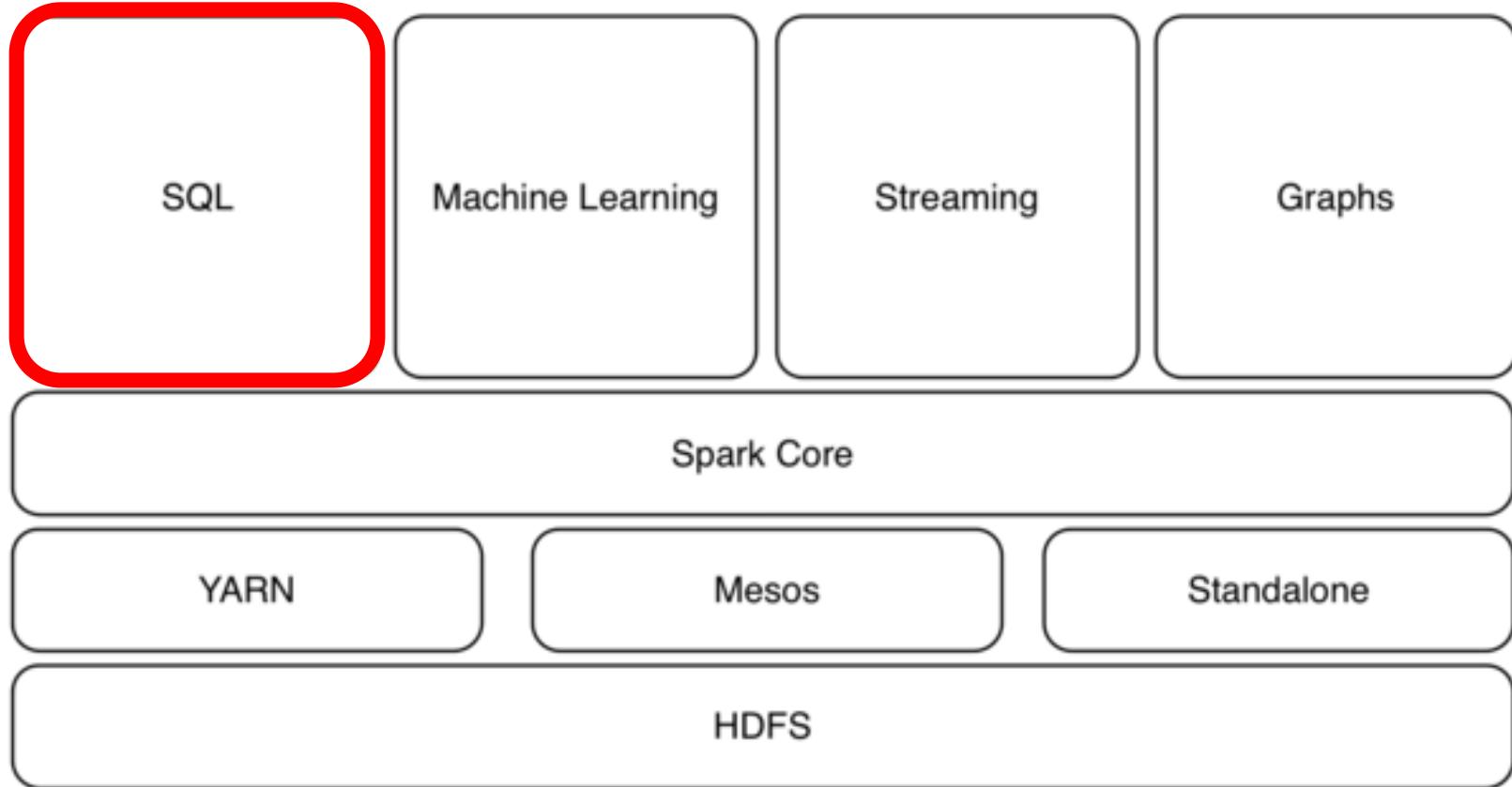
- Présentation du jeu de données et de la problématique.
- Premiers pas avec Spark Core.
  - Connection entre Spark et Jupyter Notebook.
  - Création du SparkContext.
  - Chargement des données avec Spark Core.
  - Visualiser son rdd avec .take() et .collect().
  - Compte du nombre d'observations.
- Quelques transformations et actions.
  - Nombre de mots par SMS.
  - Binarisation de la colonne ham/spam.
  - Répartition des hams et des spams.
  - Passage du SMS en minuscules.
  - Recherche de mot...

# Manipulation de dataframes avec Spark SQL

---

- Spark SQL
- Création de Dataframes
- Manipulation basique
- Manipulation avancée

# Spark SQL



# Spark SQL : Overview

- Spark SQL est un module de Spark qui se place au-dessus du Spark Core.
- Il est utilisé pour effectuer des traitements sur des données structurées appelées des dataframes.
- A ce titre, il réalise des opérations avec un niveau d'abstraction et de simplicité plus poussé que sur les RDDs notamment:
  - l'exploitation de données sur ces objets dataframes.
  - L'utilisation et l'exécution de requêtes SQL.



# Spark SQL : Concepts

- Un dataframe est un objet à deux dimensions pouvant être représenté sous forme tabulaire.
- Les dataframes sont des données structurées. En effet, elles contiennent:
  - Les « columns » (nommées) qui sont les variables du dataset.
  - Les « rows » qui sont les observations du dataset.
- La notion de dataframe se retrouve dans de nombreux langages de programmation.
  - La librairie « pandas » pour Python.
  - Les objets « data.frame » pour le langage R.
  - Les langages de type SQL.

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	X0	X1	X2	X3

# Spark SQL : SQLContext

- De façon analogue au SparkContext, le **SQLContext** est le point d'entrée de toutes les fonctions du module Spark SQL.
- Pour l'utiliser, il faut au préalable avoir configuré un objet de type SparkContext.
- On crée ensuite un objet *SQLContext()* tel que:

```
from pyspark import SQLContext
sql_ctx = SQLContext(sc)
```

Note : Depuis la version 2.0 de Spark (juin 2016), on favorise une nouvelle classe nommée `SparkSession` comme point d'entrée de Spark SQL. Cela étant, pour des raisons de comptabilité, la classe `SQLContext` reste également valable. Nous utiliserons cette première dans la suite du cours.

```
from pyspark.sql import SparkSession
conf = SparkConf().set("master", "local[*]").set("spark.app.name", "DARIO")
sql_ctx = SparkSession.builder.config(conf=conf).getOrCreate()
```

# Création de dataframes

- Il existe quatre façons de créer des dataframes avec Spark SQL:
- « **Manuellement** »: Depuis une liste de liste ou un dictionnaire.
- « **Depuis un dataframe Pandas** » : (seulement valable avec l'API Python).
- « **Depuis un Spark RDD** » : en effectuant une conversion.
- « **fichier externe** » : Spark SQL prend en charge la lecture de fichiers externes grâce à différentes méthodes.
  - Fichier JSON ( méthode `sqlContext.read.json()` )
  - Fichier parquet (méthode `sqlContext.read.parquet()`)
  - Fichier CSV (depuis Spark version 2.0)

# Création de dataframes

- On peut créer très simplement un dataframe spark SQL de façon analogue à la méthode “.parallelize()” qui existe dans Spark Core. C'est la méthode “.createDataFrame()”.
- Remarque : La création manuelle ou depuis pandas n'est pas appropriée pour le chargement de jeux de données très volumineux.

```
header = ["ville", "type", "montant"]
data = [[ "Paris", "alimentation", 19.00], [ "Marseille", "vetements", 12.00], [ "Paris", "alimentation", 8.00],
        [ "Paris", "vetements", 15.00], [ "Marseille", "alimentation", 20.00], [ "Lyon", "mediatheque",10.00]]

# manuellement :
df1 = sql_ctx.createDataFrame(data, header)

# Depuis pandas :
import pandas as pd
pandas_df = pd.DataFrame(columns=header, data=data)
df2 = sql_ctx.createDataFrame(pandas_df)

# Depuis un RDD :
rdd = sc.parallelize(data)
df3 = sql_ctx.createDataFrame(rdd, header)
```

# Création de dataframes : Fichier externe

- Il est également possible de lire des fichiers externes depuis Spark. Pour cela, il faut faire appel aux fonctionnalités de la classe “`pyspark.sql.DataFrameReader`” qui permet selon les méthodes utilisées de pouvoir lire des données depuis différentes sources. Depuis la version 2.0, les formats supportés (version 2.0) sont :
  - le format CSV.
  - le format JSON.
  - le format ORC.
  - le format parquet.
  - format texte.
  - les bases de données SQL compatibles JDBC.
  - Hive (écosystème Hadoop).
- Un exemple de lecture d'un fichier CSV depuis Spark SQL :

```
df = sql_ctx.read.csv('data.csv')
```

Remarque : De façon analogue, il est bien sûr possible d'écrire des dataframes Spark vers ces différentes sources et formats.

# Manipulation basique

- L'API Python Spark SQL propose de nombreuses fonctionnalités pour manipuler les colonnes et dataframes.
- Les commandes sont proche de l'API Python Pandas, ce qui le rend très intuitif pour le data scientist.

# Manipulation basique : Afficher le contenu

- Il existe plusieurs méthodes pour récupérer/afficher le contenu d'un dataframe:
  - La méthode « `.collect()` » qui rapatrie le dataframe au user sous forme de liste python.
  - La méthode « `.take(n)` » qui récupère les « `n` » premiers éléments du dataframe.
  - La méthode « `.show()` » qui affiche le contenu sur la sortie standard.

```
# collect()
python_list = df.collect()
print(python_list)

# take() :
first_ten = df.take(10)

# show() :
df.show()
```

La méthode « `.show()` » :

ville	type	montant
Paris	alimentation	19.0
Marseille	vetements	12.0
Paris	alimentation	8.0
Paris	vetements	15.0
Marseille	alimentation	20.0
Lyon	mediatheque	10.0

# Manipulation basique : Le schéma

- Les dataframes étant des données structurées, celles-ci possèdent un schéma.
- Il est possible d'afficher le schéma avec l'API Spark Python. Pour cela, on peut utiliser la méthode « `.printSchema()` ».

```
df.printSchema()
```

```
root
 |-- ville: string (nullable = true)
 |-- type: string (nullable = true)
 |-- montant: double (nullable = true)
```

- On peut également utiliser l'attribut « `dtypes` » d'un dataframe.

```
schema = df.dtypes
print(schema)

[('ville', 'string'), ('type', 'string'), ('montant', 'double')]
```

# Manipulation basique : Le schéma

- Avec Spark SQL, le schéma est automatiquement détecté lorsque l'on charge les données. Cependant, il est possible de le spécifier manuellement lors de la création d'un dataframe Spark. Cela se fait à partir des classes de typage que l'on retrouve dans le module "pyspark.sql.types" :

```
from pyspark.sql.types import FloatType, IntegerType, StringType, StructField, StructType
data = [["Paris", "alimentation", 19.00], ["Marseille", "vetements", 12.00], ["Paris", "alimentation", 8.00],
        ["Paris", "vetements", 15.00], ["Marseille", "alimentation", 20.00], ["Lyon", "mediatheque", 10.00]]
schema = StructType([StructField("ville", StringType(), nullable=True),
                     StructField("type", StringType(), nullable=True),
                     StructField("montant", FloatType(), nullable=True)]))

df = sql_ctx.createDataFrame(data, schema=schema)
df.printSchema()

root
 |-- ville: string (nullable = true)
 |-- type: string (nullable = true)
 |-- montant: float (nullable = true)
```

- Il est également possible de convertir les colonnes d'un type vers un autre avec la méthode `.cast()` :

```
# Converting column "montant" from Float to String :
df = df.withColumn("montant", df.montant.cast(StringType()))
df.printSchema()
```

```
root
 |-- ville: string (nullable = true)
 |-- type: string (nullable = true)
 |-- montant: string (nullable = true)
```

# Manipulation basique : Sélection de colonnes

- Il est possible de sélectionner des colonnes d'un dataframe avec la méthode « `.select()` ».
- Cette méthode retourne un dataframe.

```
df = df.select("ville", "montant")
df.show()
```

ville	montant
Paris	19.0
Marseille	12.0
Paris	8.0
Paris	15.0
Marseille	20.0
Lyon	10.0

- Pour manipuler une seul colonne (équivalent série en Python Pandas), on peut utiliser l'attribut « `column` ».

```
ville = df.ville
```

# Manipulation basique : Filtrage de lignes

- Il existe plusieurs façons de filtrer des lignes d'un dataframe.
- La plus simple et la plus intuitive est la méthode « `.filter()` » que nous avons déjà vu et qui s'applique également sur des RDDs.

```
header = ["ville", "type", "montant"]
data = [["Paris", "alimentation", 19.00], ["Marseille", "vetements", 12.00], ["Paris", "alimentation", 8.00],
        ["Paris", "vetements", 15.00], ["Marseille", "alimentation", 20.00], ["Lyon", "mediatheque", 10.00]]
df = sql_ctx.createDataFrame(data, header)

# Filtering
new_df = df.filter(df.ville == "Paris")
new_df.show()
```

ville	type	montant
Paris	alimentation	19.0
Paris	alimentation	8.0
Paris	vetements	15.0

# Manipulation basique : Sort

- Il est possible de trier les lignes d'un dataframes au travers de plusieurs méthodes dont notamment:
  - La méthode « `.sort()` »
  - La méthode « `.orderBy()` »

```
header = ["ville", "type", "montant"]
data = [["Paris", "alimentation", 19.00], ["Marseille", "vetements", 12.00], ["Paris", "alimentation", 8.00],
        ["Paris", "vetements", 15.00], ["Marseille", "alimentation", 20.00], ["Lyon", "mediatheque", 10.00]]
df = sql_ctx.createDataFrame(data, header)
```

```
# Order By
new_df = df.orderBy(df.montant.asc())
new_df.show()
```

ville	type	montant
Paris	alimentation	8.0
Lyon	mediatheque	10.0
Marseille	vetements	12.0
Paris	vetements	15.0
Paris	alimentation	19.0
Marseille	alimentation	20.0

# Manipulation basique : Création de colonnes

- Il est possible d'ajouter une colonne à un dataframe avec la méthode « .withColumn() ». Les données étant distribuées en partitions, il n'est possible de créer des colonnes que de deux façons:
  - Créer une colonne avec des valeurs par défaut.
  - En dérivant une autre colonne déjà existante.
  - en utilisant des fonctions prédéfinies.
  - En utilisant des UDFs (User Defined Functions).

```
from pyspark.sql.functions import lit, rand

# Avec une constante :
df = df.withColumn("constant_6", lit(6))

# en transformant une colonne :
df = df.withColumn("less50percent", df.montant / 2)

# En utilisant une fonction prédéfinie :
df = df.withColumn("rand_number", rand())
df.show()
```

ville	type	montant	constant_6	less50percent	rand_number
Paris	alimentation	19.0	6	9.5	0.5057885756248387
Marseille	vetements	12.0	6	6.0	0.5823445224727232
Paris	alimentation	8.0	6	4.0	0.9893496354401767
Paris	vetements	15.0	6	7.5	0.4017801321096276
Marseille	alimentation	20.0	6	10.0	0.33322622659212664
Lyon	mediatheque	10.0	6	5.0	0.7155794474696288

# Manipulation basique : Gestion de colonnes

- Il est également possible de renommer une colonne avec la méthode « `.withColumnRenamed()` »:

```
new_df = df.withColumnRenamed("ville", "metropole")
```

- Pour supprimer une colonne, on utilise la méthode “`.drop()`” :

```
new_df = new_df.drop("rand_number")
new_df.show()
```

metropole	type	montant	constant_6	less50percent
Paris	alimentation	19.0	6	9.5
Marseille	vetements	12.0	6	6.0
Paris	alimentation	8.0	6	4.0
Paris	vetements	15.0	6	7.5
Marseille	alimentation	20.0	6	10.0
Lyon	mediatheque	10.0	6	5.0

# Manipulation avancée : fonctions SparkSQL

- En manipulant des dataframes, il est très courant de manipuler et / ou appliquer des fonctions directement sur les colonnes.
- Afin de pouvoir répondre à la plupart des situations, PySpark possède déjà un certain nombre de fonctions pré-implémentées et présentes dans le module “*pyspark.sql.functions*”.
- Les fonctions sont nombreuses et de tous types. Il ne faut pas hésiter à s'y référer.
  - Fonctions mathématiques (cos, sin, tan, abs, exp, log...).
  - Manipulation de dataframes (concaténation, répartition, manipulation de structures...).
  - Statistiques (moyenne, variance, corrélations, covariances...)
  - Manipulation et calcul de dates.
  - Opérateurs d'algèbre relationnel.

```
from pyspark.sql.types import IntegerType, FloatType, StringType
from pyspark.sql.functions import cos

# Create data + Apply cos
rdd = sc.parallelize([[1, 2], [2, 3], [3, 4], [4, 5]])
df = sqlContext.createDataFrame(data=rdd, schema=["x1", "x2"])
new_df = df.withColumn("cos_x", cos(df.x1 + df.x2))
new_df.show()
```

x1	x2	cos_x
1	2	-0.9899924966004454
2	3	0.28366218546322625
3	4	0.7539022543433046
4	5	-0.9111302618846769

# Manipulation avancée : fonctions SparkSQL

- pyspark.sql.functions module

- |                         |                     |                      |
|-------------------------|---------------------|----------------------|
| ▪ abs                   | ▪ column            | ▪ desc               |
| ▪ acos                  | ▪ concat            | ▪ encode             |
| ▪ add_months            | ▪ concat_ws         | ▪ exp                |
| ▪ approxCountDistinct D | ▪ conv              | ▪ explode            |
| ▪ approx_count_distinct | ▪ corr              | ▪ exprm1             |
| ▪ array                 | ▪ cos               | ▪ expr               |
| ▪ array_contains        | ▪ cosh              | ▪ factorial          |
| ▪ asc                   | ▪ count             | ▪ first              |
| ▪ ascii                 | ▪ countDistinct     | ▪ floor              |
| ▪ asin                  | ▪ covar_pop         | ▪ format_number      |
| ▪ atan                  | ▪ covar_samp        | ▪ format_string      |
| ▪ atan2                 | ▪ crc32             | ▪ from_json          |
| ▪ avg                   | ▪ create_map        | ▪ from_unixtime      |
| ▪ base64                | ▪ cume_dist         | ▪ from_utc_timestamp |
| ▪ bin                   | ▪ current_date      | ▪ get_json_object    |
| ▪ bitwiseNOT            | ▪ current_timestamp | ▪ greatest           |
| ▪ broadcast             | ▪ date_add          | ▪ grouping           |
| ▪ bround                | ▪ date_format       | ▪ grouping_id        |
| ▪ cbrt                  | ▪ date_sub          | ▪ hash               |
| ▪ ceil                  | ▪ datediff          | ▪ hex                |
| ▪ coalesce              | ▪ dayofmonth        | ▪ hour               |
| ▪ col                   | ▪ dayofyear         | ▪ hypot              |
| ▪ collect_list          | ▪ decode            | ▪ initcap            |
| ▪ collect_set           | ▪ degrees           | ▪ input_file_name    |
|                         | ▪ dense_rank        | ▪ instr              |

(...)

# Manipulation avancée : UDF

- Dans le cas où une fonction Spark SQL ne serait pas définie, il est également possible de créer ses propres fonctions à partir d'objets Spark dits “UDF”.
- Les UDF (*User Defined Function*) sont des fonctions Spark SQL qui permettent d'appliquer une fonction définie par l'utilisateur sur l'ensemble des lignes d'un dataframe.
- Cette méthode est équivalente à la méthode “`.apply()`” que l'on peut retrouver dans Python Pandas.
- Voici ci-dessous un exemple simple d'UDF appliqué sur un dataframe :

x1	x2
1	2
2	3
3	4
4	5

$$y = \text{UDF}(x1) = x1 + 3$$

x1	x2	y
1	2	4
2	3	5
3	4	6
4	5	7

# Manipulation avancée : UDF

- Pour définir une UDF, il faut commencer par créer une fonction classique python, qui, une fois les entrées spécifiées, est capable de retourner le résultat attendu.
- Une fois la fonction créée, on l'encapsule alors dans l'instanciation d'un objet UDF (`pyspark.sql.functions`). Ne pas oublier de spécifier le typage de la colonne de sortie.
- On applique alors l'UDF au travers d'un méthode “`.withColumn()`”.

The diagram illustrates the application of a User-Defined Function (UDF) to transform a DataFrame. On the left, an input DataFrame has columns `x1` and `x2`, containing the values:

x1	x2
1	2
2	3
3	4
4	5

In the center, the formula  $y = \text{UDF}(x1) = x1 + 3$  is shown above a large arrow pointing to the right. On the right, the resulting DataFrame has columns `x1`, `x2`, and `y`, containing the values:

x1	x2	y
1	2	4
2	3	5
3	4	6
4	5	7

# Manipulation avancée : UDF

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType, FloatType, StringType

# Create UDF
def add_3(x):
    return x + 3

my_udf = udf(lambda col: add_3(col), IntegerType())

# Create data + apply UDF
rdd = sc.parallelize([[1, 2], [2, 3], [3, 4], [4, 5]])
df = sqlContext.createDataFrame(data=rdd, schema=["x1", "x2"])
new_df = df.withColumn("y", my_udf("x1"))
new_df.show()
```

x1	x2	y
1	2	4
2	3	5
3	4	6
4	5	7

# Manipulation avancée : UDF

- **Note :** Il est également possible de pouvoir passer plusieurs colonnes au sein d'une seule UDF.
- **Note :** Il est également possible de retourner plusieurs colonnes après une même UDF en retournant un objet de type “*StructType*”.
- **Note :** Il est également possible de passer des objets extérieurs dans une UDF, notamment en créant une fonction “*wrapper*”. La manipulation est un peu plus complexe et ne sera pas nécessairement montrée dans le cours.

x1	x2		
1	2		
2	3		
3	4		
4	5		

$y1, y2 = \text{UDF}(x1, x2)$

$y1 = x1 + x2$

$y2 = x1 - x2$

The diagram illustrates a data transformation process. On the left, there is a 4x2 input table with columns labeled 'x1' and 'x2'. The rows contain the values (1, 2), (2, 3), (3, 4), and (4, 5). An arrow points from this table to a larger 4x4 output table on the right. The output table has columns labeled 'x1', 'x2', 'y1', and 'y2'. The 'x1' and 'x2' columns correspond to the input table. The 'y1' column contains the sum of the input 'x1' and 'x2' values (3, 5, 7, 9). The 'y2' column contains the difference between the input 'x1' and 'x2' values (-1, -1, -1, -1). This visualizes how a single UDF can be applied to multiple columns to produce multiple output columns.

x1	x2	y1	y2
1	2	3	-1
2	3	5	-1
3	4	7	-1
4	5	9	-1

# Manipulation avancée : UDF

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType, FloatType, StringType, StructType, StructField

# Create UDF
def add_xx(x1, x2):
    y1 = x1 + x2
    y2 = x1 - x2
    return y1, y2

result_struct = StructType([StructField("y1", IntegerType(), True), StructField("y2", IntegerType(), True)])
my_udf = udf(lambda col1, col2: add_xx(col1, col2), result_struct)

# Create data + apply UDF
rdd = sc.parallelize([[1, 2], [2, 3], [3, 4], [4, 5]])
df = sqlContext.createDataFrame(data=rdd, schema=["x1", "x2"])
new_df = df.withColumn("result_struct", my_udf("x1", "x2"))
new_df.printSchema()

# Explode the structure
new_df = new_df.withColumn("y1", new_df.result_struct.y1)
new_df = new_df.withColumn("y2", new_df.result_struct.y2)
new_df.printSchema()
new_df.show()
```

```
root
|-- x1: long (nullable = true)
|-- x2: long (nullable = true)
|-- result_struct: struct (nullable = true)
|   |-- y1: integer (nullable = true)
|   |-- y2: integer (nullable = true)
|-- y1: integer (nullable = true)
|-- y2: integer (nullable = true)
```

x1	x2	result_struct	y1	y2
1	2	[3,-1]	3	-1
2	3	[5,-1]	5	-1
3	4	[7,-1]	7	-1
4	5	[9,-1]	9	-1

# Manipulation avancée : Requêtes SQL

- Enfin, une autre force de Spark SQL est qu'il permet de mener des requêtes d'algèbre relationnel de type SQL-92 afin d'accéder ces données.
- L'algèbre relationnel est en effet particulièrement adapté pour accéder à nos données après un traitement complexe.
- Pour cela, on peut utiliser la méthode « `.sql()` » qui s'utilise sur l'objet `SQLContext`:

```
header = ["ville", "type", "montant"]
data = [[ "Paris", "alimentation", 19.00], [ "Marseille", "vetements", 12.00], [ "Paris", "alimentation", 8.00],
        [ "Paris", "vetements", 15.00], [ "Marseille", "alimentation", 20.00], [ "Lyon", "mediatheque", 10.00]]
df = sql_ctx.createDataFrame(data, header)

# Requêtes SQL :
sql_ctx.registerDataFrameAsTable(df, "achats")
sql_df = sql_ctx.sql("SELECT * FROM achats WHERE montant > 11.0")
sql_df.show()
```

ville	type	montant
Paris	alimentation	19.0
Marseille	vetements	12.0
Paris	vetements	15.0
Marseille	alimentation	20.0

# Live Coding !

- Premiers pas avec Spark SQL :
  - Connection en Spark et Jupyter Notebook.
  - Création du SparkContext et du SQLContext.
  - Chargement des données CSV.
  - Affichage des données.
  - Nommage des colonnes.
- Nettoyage des données :
  - Création de la colonne label.
  - Passage en minuscules.
  - Suppression de la ponctuation et des nombres.
  - Stemming.
- Création de features supplémentaires :
  - Compte de mots.
  - Compte de monnaies.
- Un peu d'exploration :
  - Tableau croisé entre le compte de monnaies et la possibilité de spam.
  - corrélation entre ces deux variables.

# Conclusion

- Spark est un framework très puissant et très utilisé par les data scientists travaillant en contexte Big Data.
- Spark s'articule parfaitement avec le framework Hadoop. Pour cela, il a seulement besoin d'être associé avec un système de stockage distribué (HDFS, Cassandra...) et un système de gestion d'allocation de ressources (YARN, Mesos...)
- Par ailleurs, il remplace de plus en plus Hadoop MapReduce. En effet, Spark possède de bien meilleures performances grâce à sa technique de stockage de données en mémoire entre deux itérations.
- **Spark Core** est la brique de base de Spark. Il travaille sur des RDDs qui sont des collections de données distribuées sur plusieurs machines ( $\approx$  listes de listes en Python).
- **Spark SQL** est une brique construite au-dessus du Spark Core. Elle travaille sur des dataframes et possède une API assez proche de Python Pandas.

# Annexes

---

Spark for Linux / Windows

# Spark for linux/windows - python 2.7 (1)

1/ Télécharger Spark : se rendre sur le site de Spark dans la section “downloads” :

<https://spark.apache.org/downloads.html>

2/ Télécharger Spark au format tgz avec les paramètres suivants :

The screenshot shows the Apache Spark website at <https://spark.apache.org/downloads.html>. The page features the Apache Spark logo and the tagline "Lightning-fast cluster computing". A blue navigation bar at the top includes links for Download, Libraries, Documentation, Examples, Community, and Developers. Below the bar, a section titled "Download Apache Spark™" lists five steps for downloading the software:

- Choose a Spark release: 2.2.0 (Jul 11 2017)
- Choose a package type: Pre-built for Apache Hadoop 2.7 and later
- Choose a download type: Direct Download
- Download Spark: spark-2.2.0-bin-hadoop2.7.tgz
- Verify this release using the 2.2.0 signatures and checksums and project release KEYS.

A note at the bottom states: "Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support."

# Spark for linux/windows - python 2.7 (1)

3/ Ouvrir un terminal / explorateur OS et se rendre dans l'espace “Downloads”.

```
[MBP-de-Thibaud:~ Thibaud$ cd Downloads/  
[MBP-de-Thibaud:Downloads Thibaud$ ls
```

4/ Repérer le fichier Spark téléchargé et le dézipper :

```
[MBP-de-Thibaud:Downloads Thibaud$ tar -xvzf spark-2.2.0-bin-hadoop2.7.tgz  
x spark-2.2.0-bin-hadoop2.7/  
x spark-2.2.0-bin-hadoop2.7/NOTICE
```

5/ Déplacer le répertoire spark dézippé dans un répertoire plus approprié:

```
[MBP-de-Thibaud:Downloads Thibaud$ mv spark-2.2.0-bin-hadoop2.7 /opt/]
```

# Spark for linux/windows - python 2.7 (1)

6/ Ouvrir un notebook jupyter (ou tout autre IDE) et entrez le code suivant dans une cellule.

```
import os
import sys
spark_path = "/opt/spark-2.2.0-bin-hadoop2.7/"
python_path = sys.executable # == /Users/Thibaud/anaconda/bin/python/
os.environ["SPARK_HOME"] = spark_path
os.environ["HADOOP_HOME"] = spark_path
os.environ["PYSPARK_PYTHON"] = python_path
os.environ["PYSPARK_DRIVER_PYTHON"] = python_path
sys.path.append(spark_path + "python/lib/pyspark.zip")
sys.path.append(spark_path + "python/lib/py4j-0.10.4-src.zip")

from pyspark import SparkConf, SparkContext
conf = SparkConf().set("master", "local[*]").set("spark.app.name", "DARIO")
sc = SparkContext(conf=conf)
```

7/ Modifier les variables spark\_path et python\_path en fonction de :

- spark\_path : répertoire où a été déplacé Spark (étape 5)
- python\_path : l'emplacement de python. Par défaut, la commande “sys.executable” y mène directement.

8 / Exécuter le code. Si aucune erreur ne s'affiche, Spark est maintenant opérationnel!