DATUM ACADEMY

Hadoop, Spark & Map/Reduce

Benjamin Renaut
Mis à jour par Sergio Simonian

Plan

Module 1 :

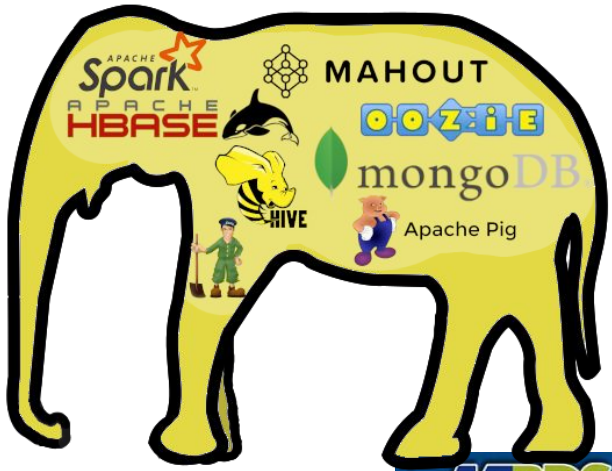
- Big Data et le calcul distribué
- Le paradigme Map/Reduce
- Introduction à Hadoop

Module 2 :

- Programmation Hadoop

Module 3 :

- Spark
- Écosystème autour d'Hadoop



DATUM ACADEMY

SPARK

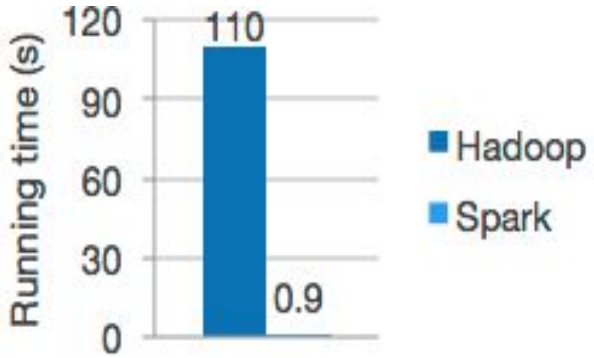
Présentation - Architecture - Usage - API Python

Présentation

- Framework de conception et d'exécution MapReduce.
- Originellement (2014) un projet de l'université de Berkeley en Californie, désormais un logiciel libre de la fondation Apache.
- Site officiel: https://spark.apache.org/

Présentation

- ➔ **Sensiblement plus rapide que Hadoop.**
  Notamment pour les tâches impliquant de multiples exécutions de Map et/ou Reduce.
  Évite les lectures et écritures répétées sur HDFS.
- ➔ **Beaucoup plus flexible que Hadoop.**
  Pas de cadre rigide de type "map+shuffle+reduce".
  Bien adapté au Machine Learning.
- ➔ **Facile à utiliser et à développer.**
- ➔ **Très intégrable avec d'autres solutions**.
  Peut très facilement lire des données depuis de nombreuses sources



Présentation

Développé en **Scala** (langage orienté objet dérivé de Java et incluant de nombreux aspects des langages fonctionnels).

Quatre langages supportés :

- Scala
- Java
- Python (PySpark)
- R (SparkR)

Performances et fonctionnalités plus ou moins équivalentes pour les 4.

Présentation

Fournit de divers bibliothèques :

- ➔ **Spark Core** : cadre principal de Spark.
- ➔ **Spark SQL** : composant de données structurées ; accès et manipulation unifiés des données.
- ➔ **Spark MLlib** : composant de Machine Learning distribué.
- ➔ **Spark GraphX** : composant de calcul de graphes distribués.
- ➔ **Spark Streaming** : composant de traitement du streaming (continu/quasi temps réel).

Le présent module est consacré à **Spark Core**.

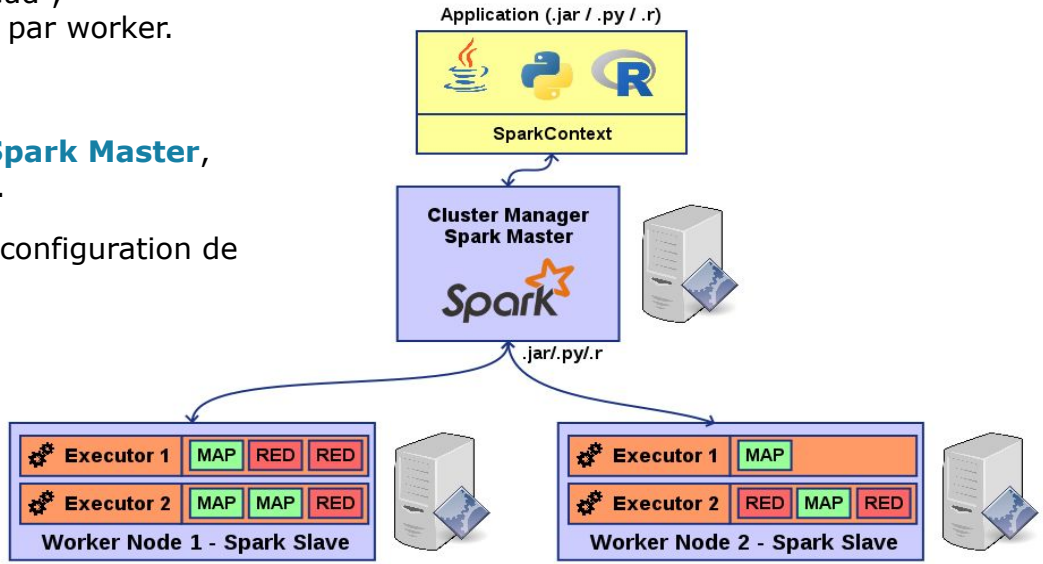
Présentation

Quatre modes d'exécution :

- Cluster Spark natif.
- Hadoop (YARN).
- Mesos (Spark natif + scheduler Mesos).
- Kubernetes (depuis décembre 2019 / Spark 2.3)

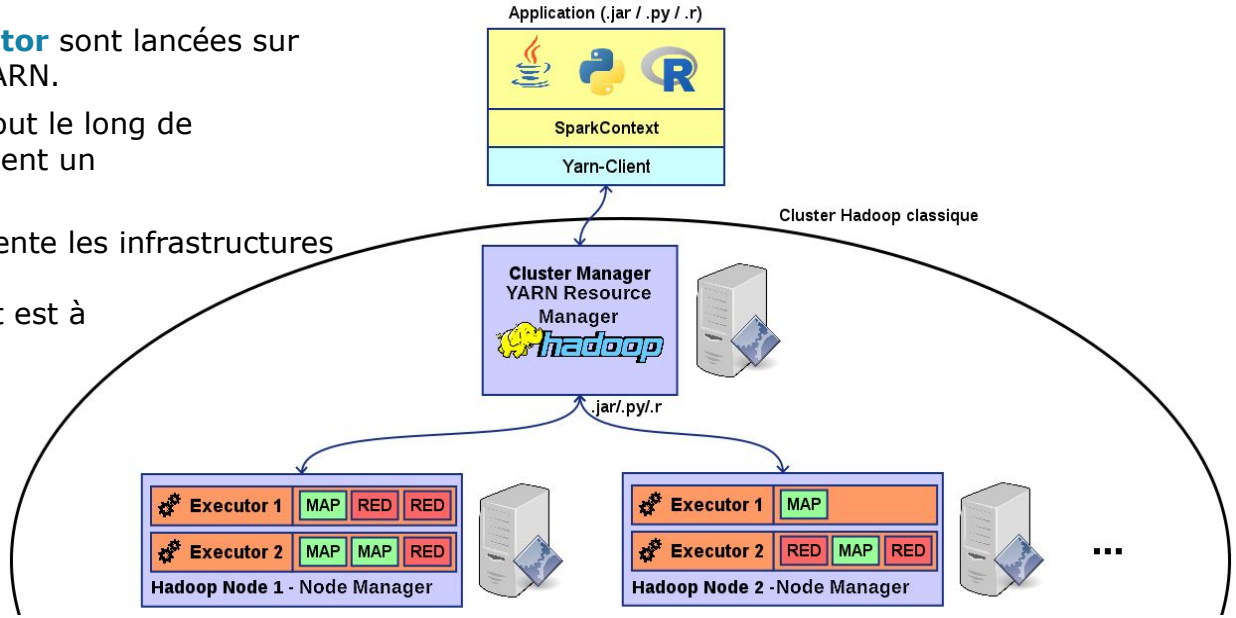
Architecture – Haut niveau - Mode Spark natif

- Un service **Spark Worker** par nœud ;
  N créneaux d'exécution **Executor** par worker.
- Ils disposent tous d'un **cache**.
- L'unique gestionnaire du cluster, **Spark Master**, est un point de défaillance unique.
- Nécessité de mettre en place une configuration de haute disponibilité.



Architecture – Haut niveau - Mode YARN

- Des tâches Spark **Executor** sont lancées sur les **NodeManager** de YARN.
- Elles resteront lancées tout le long de l'exécution, et maintiennent un cache similaire.
- Spark souvent complémente les infrastructures Hadoop existantes.
  Ce mode de déploiement est à l'heure actuelle le plus courant en production.

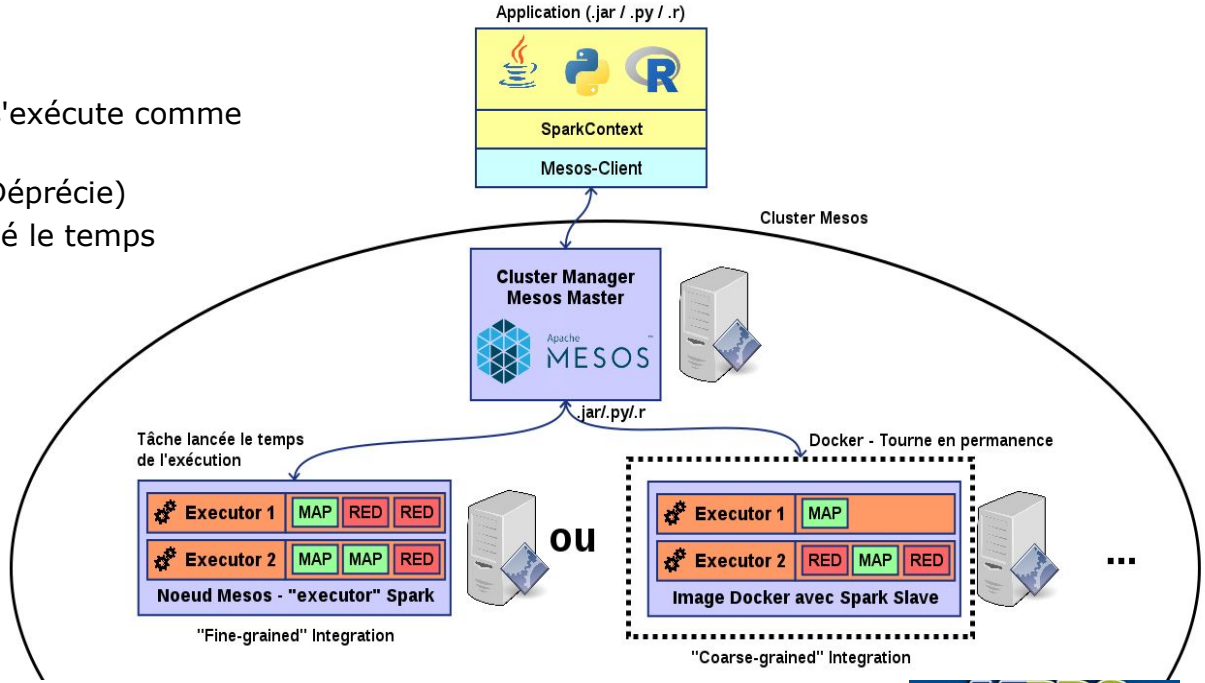


Architecture – Haut niveau - Mode Mesos

Deux intégrations possibles:

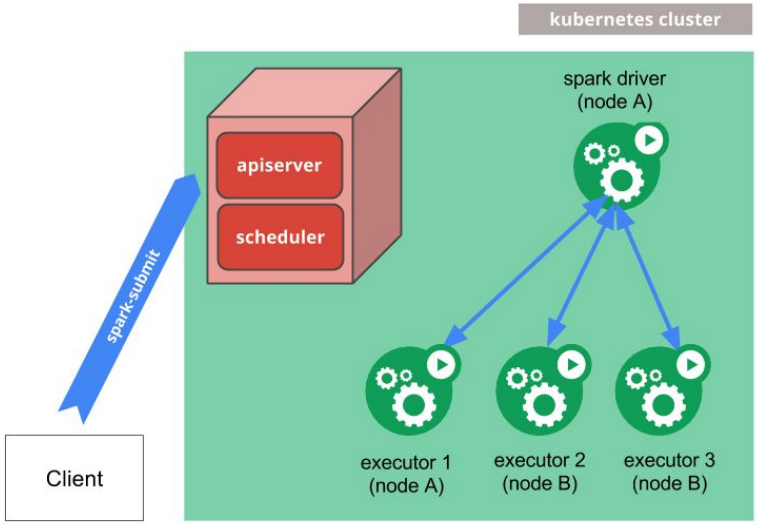
- → « **Coarse-grained** » (chaque **Spark Worker** s'exécute comme une seule tâche Mesos)
- → Tâche Mesos - « Fine » (Déprécie)

Là aussi, un cache est conservé le temps de l'exécution.



Architecture – Haut niveau - mode Kubernetes

- Très similaire au mode « Coarse-grained » de Mesos.
- Le gestionnaire de cluster est Kubernetes Master (kube-scheduler+kube-controller-manager).
- Plus facile à déployer que Mesos.



Architecture – Haut niveau

- Architectures similaires - un gestionnaire des ressources par cluster, au moins un service ‘exécuteur’ sur chaque nœud.
- Le code principal d’un programme Spark est appelé **Driver**.
- Il y a deux modes d'exécution :
  - Mode **Client** : le Driver s'exécute à l'endroit où le programme est initialement lancé.
  - Mode **Cluster** : le Driver s'exécute dans un nœud sur le cluster.

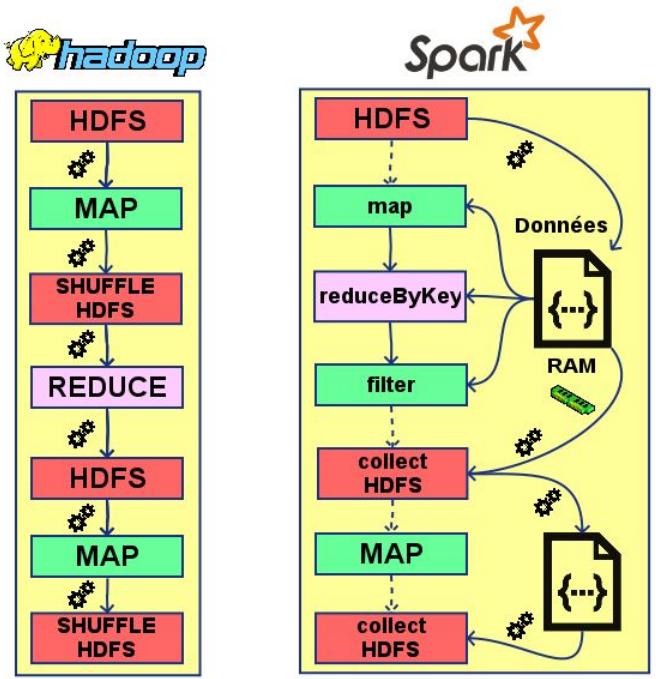
Architecture – Haut niveau

- Le **Driver** communique avec le gestionnaire de cluster ; demande des ressources et des créneaux d'exécution.
- La tolérance aux pannes est gérée par le gestionnaire de cluster.
- Aucun stockage distribué persistant dédié ; peut utiliser HDFS, ou de nombreuses autres alternatives (autres systèmes de fichiers distribués, bases de données, etc.).

Architecture – Haut niveau

- **Hadoop** a besoin de sérialiser / désérialiser les données avant/pendant et après chaque exécution MapReduce.
- L'écriture sur HDFS est une opération lente.
- De nombreux programmes ont besoin de faire plusieurs exécutions itératives de MapReduce.

- **Spark** garde les données en RAM et est capable de déterminer quand il aura besoin de sérialiser les données / les réorganiser; et ne le fait que quand c'est nécessaire.
- On peut explicitement lui demander de conserver des données en RAM, quand elles seront nécessaires entre plusieurs étapes d'écriture.



Architecture – Haut niveau - Remarques

- Spark **consomme beaucoup plus de mémoire vive** que Hadoop. Les machines du cluster nécessitent ainsi plus de RAM.
- Le cluster manager (« Spark Master ») est **un point de défaillance unique** et nécessite la mise en place d'une architecture de haute disponibilité.
- **Spark est préférée pour :**
  - Des tâches complexes qui nécessitent plusieurs itérations de MapReduce.
  - Des traitements des données en temps réel/en continu
- **Hadoop est préférée pour :**
  - Des tâches simples, nécessitant une seule itération de MapReduce.
  - La création des clusters à partir de matériel hétérogène de bas de gamme (coût plus faible).

Architecture – Les RDDs

La principale abstraction de données dans Spark : la notion de **RDD (Resilient Distributed Datasets)**

Il s'agit de larges collections des divers éléments.

Ils sont :

- ➔ **Distribués** (pour permettre à plusieurs nœuds de traiter les données)
- ➔ **Partitionnés** (chaque nœud possède une ou plusieurs parties du RDD).
- ➔ **Tolérant aux pannes** (pour limiter le risque de perte de données, les RDDs sont capables de recalculer les partitions manquantes ou endommagées suite à des défaillances de nœuds).
- ➔ **En lecture seule** (un traitement appliqué à un RDD donne lieu à la création d'un nouveau RDD).

Architecture – Les RDDs

Deux types d'opérations possibles sur les RDDs :

- **Transformation** :
  - Crée un nouveau RDD modifié.
  - Évaluation paresseuse : ne s'exécute que quand il est nécessaire d'accéder aux données.
  - Un traitement sur un RDD est une transformation.
- **Action** :
  - Accède aux données d'un RDD.
  - Cause son évaluation (application de toutes les transformations l'une après l'autre).
  - La lecture d'un RDD est une action.

Architecture – Les RDDs

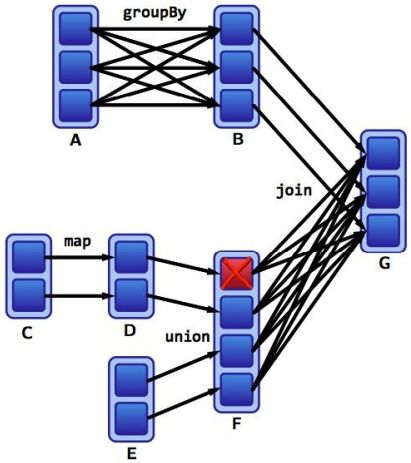
Parmi les opérations on distingue des opérations **larges** et **étroites** :

→ Une **opération étroite** :
- Effectue une opération sur le RDD "isolé par partition". Ne change pas son partitionnement.
- Est plutôt rapide car ne nécessite pas le déplacement des données sur le réseau du cluster.

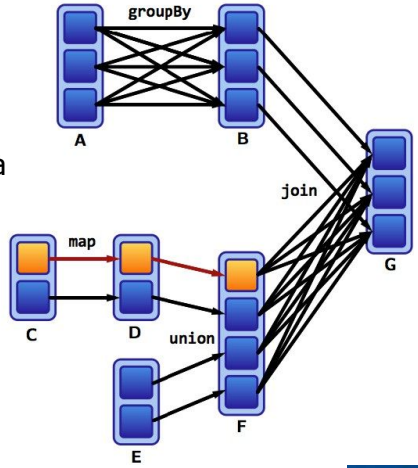
→ Une **opération large** :
- Effectue une opération sur le RDD qui nécessite l'accès à plus d'une partition du RDD.
- Nécessitent un "**shuffle**" (échange) partiel ou complet des données entre les nœuds.
- Est plus lente que les transformations étroites.

Architecture – Les RDDs

- Spark maintient une **chronologie des opérations** pour les partitions des RDDs ; il peut les évaluer, si nécessaire plusieurs fois pour la tolérance aux pannes.
- Pas de redondance nécessaire (mais possible) : Spark **recalcule à nouveau les données perdues** à partir de la chronologie des opérations depuis la source.

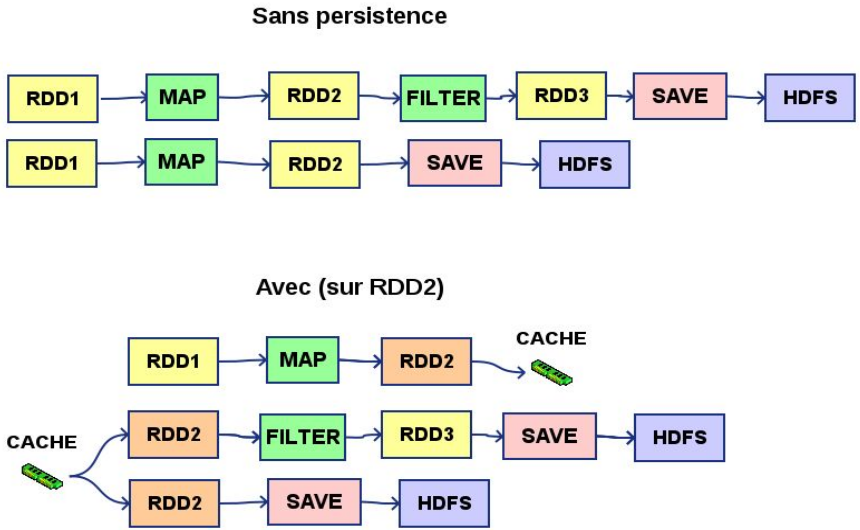


Spark recalcule à nouveau **F** à partir de la partition d'origine **C**



Architecture – Les RDDs

- Deux tâches différentes appliquant une opération à un même RDD causent l'exécution de cette transformation **deux fois**.
- Pour éviter cela, on peut persister un RDD en mémoire, pour qu'il ne soit calculé qu'une fois.
- Lorsqu'un RDD est persisté, il est gardé en mémoire **après la première évaluation**.

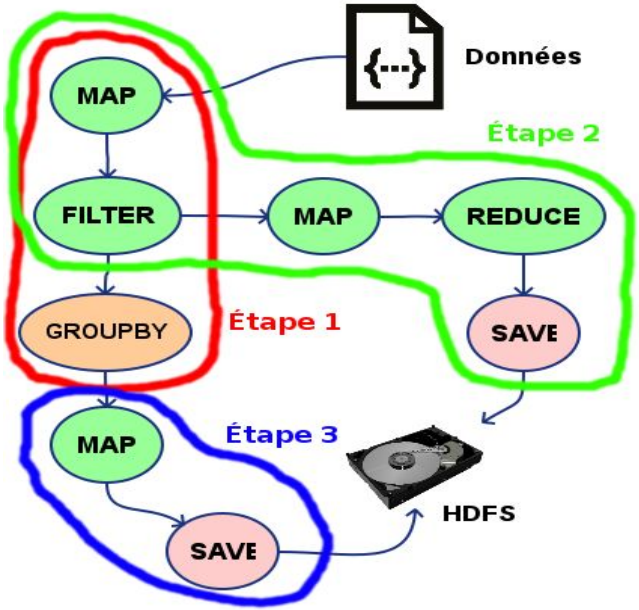


Exécution

- Tout programme Spark est composé d'un **Driver** (programme contenant un objet **SparkContext**).
- Le **SparkContext** interagit avec le cluster. Il a deux composants principaux : le **Tasks Scheduler** et le **DAG Scheduler**.
- Lors de l'exécution d'un programme :
  - Le **DAG Scheduler** construit un Graphe Orienté Acyclique (DAG, Directed Acyclic Graph) avec les actions et transformations définis dans le programme et découpe ensuite le graphe en étapes (stages), chacun contenant un certain nombre de tâches : des actions ou des transformations.
  - Le **Tasks Scheduler** soumet ensuite les tâches au cluster manager.

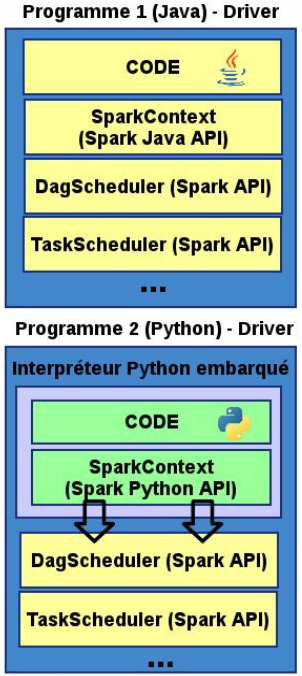
Exécution

- Les étapes sont découpées par :
  - Des actions.
  - Des transformations larges.
- Le DAG permet à Spark de savoir quand il devrait persister sur le disque.



Exécution

- Les programmes utilisant l'API Spark diffèrent de Java/Scala ont des performances proches de l'exécution native (avec Scala/Java).
- Spark exécute le code Python/R via un interpréteur correspondant embarqué.



API Python

- L'API Spark est globalement unifiée: les fonctions et leurs synopsis sont globalement identiques d'un langage à l'autre.
- Le plus important est donc de connaître les différentes opérations.
- L'API présentée ici est celle de **Python**; mais les APIs Scala, Java et (dans une moindre mesure) R sont similaires.
- La plupart des opérations prennent en paramètre une fonction indépendante de l'environnement (**closure**); dans les faits, cela se traduira souvent dans le code par l'usage de **fonctions lambda**.

Rappel – Les fonctions lambda

- Syntaxe d'une fonction lambda (closure) en python :

```
lambda ARG1, ARG2, ARG3, ...: RETURN_VALUE
```

- Exemples :

```
lambda x: x + 1
lambda a, b: a + b
lambda mot: (mot, 1)
```

Rappel – Les fonctions lambda

- Exemple en Java (possible depuis Java 8) :

```
(x) -> return(x+1)
(a, b) -> { return(a+b)}
(mot) → { System.out.println(mot); return(Arrays.asList(mot, 1)) }
```

- Exemple en Scala :

```
(x:Int) => (x+1)
(_:Int)+(_:Int)
(mot:String) => { println(mot); return(Seq(mot, "1")) }
```

API Python

- La classe principale permettant d'accéder à l'API Spark est **SparkContext**.
- Elle est contenu dans le module **pyspark**.
- On commencera ainsi généralement un programme Spark par :

```
from pyspark import SparkContext
```

- Remarque : si vous utilisez le Shell interactif '**pyspark**', cet import est déjà fait et vous disposez déjà d'une instance SparkContext avec la variable '**sc**'.

API Python

- Pour instancier le **SparkContext**, la syntaxe de base est :

```
SparkContext(master=MASTER_URL, appName=DESCRIPTION)
```

- L'option **master** indique le gestionnaire de cluster à contacter pour soumettre les tâches à exécuter.
- L'option **appName** fournit une description textuelle de l'application.
- Exemples :

```
# Spark en mode YARN :
sc = SparkContext(master="yarn", appName="WordCount")
# Spark en mode Natif :
sc = SparkContext("spark://10.0.0.1/")
# Spark en mode local (simule la présence d'un cluster avec N=2 nœuds) :
sc = SparkContext("local[2]", "WordCount")
# Spark en mode Mesos :
sc = SparkContext("mesos://192.168.1.1:5050", "WordCount")
# Spark en mode Kubernetes :
sc = SparkContext("k8s://https://192.168.1.254", "WordCount")
```

API Python

→ Remarque: le **SparkContext** ne peut être instancié qu'une fois par JVM.
Alternativement, nous pouvons récupérer une instance de **SparkContext** via l'API **DataFrame** de Spark:

```
from pyspark.sql import SparkSession
spark = (
        SparkSession.builder
        .master("local[*]")
        .appName("tp1")
        .getOrCreate()
)
sc = spark.sparkContext
```

API Python – Lecture des données

- SparkContext a plusieurs fonctions pour lire des données et les charger au sein d'un premier RDD.
- Ces fonctions renvoient toutes un (pointeur vers un) RDD.

Une liste non exhaustive (à appeler sur l'objet SparkContext) :

- **textFile(URL)**
  Charge un ou plusieurs fichiers texte; Le RDD résultant aura un élément par ligne.
- **binaryFiles(URL)**
  Charge un ou plusieurs fichiers binaires; Le RDD résultant aura un élément par fichier.
- **sequenceFile(URL)**
  Charge un ou plusieurs fichiers de séquence. Le RDD résultant aura un élément par objet dans le fichier.
- **wholeTextFiles(URL)**
  Charger des fichiers textes entiers. Le RDD résultant aura un élément par fichier.
- **pickleFile(URL)**
  Charge un ou plusieurs fichiers sérialisés Python 'pickle' contenant un RDD.

API Python – Lecture des données

- L'argument **URL** peut prendre les formes suivantes, par exemple :

```
hdfs:///input/poeme.txt
file:///home/john/data.txt
hdfs:///results/part-r-*
```

- Remarque : toutes les fonctions disposent d'un argument optionnel en plus – '**numPartitions**', permettant d'indiquer la quantité minimale de partitions qu'on souhaite avoir au sein du RDD.
- Certaines des ses fonctions ont également d'autres arguments non décrits ici pour ajuster leur fonctionnement.

API Python – Créer des données

- Nous pouvons aussi créer un RDD à l'aide de la méthode **parallelize** de l'objet **SparkContext**.

```
parallelize(LIST, numSlices=NUM_PARTITIONS)
```

- La méthode convertit la liste passée en paramètre en RDD (en effectuant le partitionnement), et renvoie un pointeur vers ce RDD.
- Par exemple :

```
rdd = sc.parallelize([0, 1, 2, 3, 4], numSlices=5)
rdd = sc.parallelize([f"item-#{x}" for x in range(0, 100)])
```

- Particulièrement utile pour le développement et les tests rapides.

API Python – Sauvegarde des données

- Les **RDD**s ont plusieurs fonctions pour leur sauvegarde sur HDFS ou disque.
- Le contenu d'un RDD serait stocké dans un **répertoire** en plusieurs fichiers - **un par partition** (nommés **part-XXXXX**, où XXXXX est un numéro incrémental).

Une liste non exhaustives de fonctions permettant la sauvegarde (à appeler sur des RDDs) :

- **saveAsTextFile(URL)**
  Sauvegarde les résultats dans des fichiers texte.
- **saveAsPickleFile(URL)**
  Sauvegarde les résultats dans des fichiers Python 'pickle'.
- **saveAsSequenceFile(URL)**
  Sauvegarde les résultats dans des fichiers de séquence.
- … il y en a d'autres.

API Python – Transformations

Une liste non exhaustive de transformations applicables aux RDD :

- **coalesce(numPartitions)**
  Renvoie un RDD résultant du re-partitionnement du RDD avec le nombre de partitions indiquées, qui doit être inférieur au nombre de partitions actuel.
- **distinct(numPartitions=None)**
  Renvoie un RDD où tous les éléments en double du RDD d'origine sont supprimés.
- **filter(FONCTION(1 arg))**
  Applique la fonction spécifiée sur chacun des éléments et si elle renvoie true, garde l'élément dans le RDD de retour.

API Python – Transformations

→ Exemple d'utilisation de **filter** :

```
rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
rdd2 = rdd.filter(lambda x: x > 3)
rdd2.collect()
[4, 5, 6]
```

→ Ou encore (sans fonction Lambda) :

```
def filtre(x):
  return(x % 2 == 0)
rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
rdd2 = rdd.filter(filtre)
rdd2.collect()
[2, 4, 6]
```

API Python – Transformations

➔ **map(FONCTION(1 arg))**
Renvoie un RDD résultant de l'exécution de la fonction spécifiée sur chaque élément du RDD source.
Exemples :

```
rdd = sc.parallelize(["celui", "ciel", "celui", "qui"])
rdd2 = rdd.map(lambda x: (x, 1))
rdd2.collect()
[("celui", 1), ("ciel", 1), ("celui", 1), ("qui", 1)]
```

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd2 = rdd.map(lambda x: x*2)
rdd2.collect()
[2, 4, 6, 8]
```

API Python – Transformations

➔ **flatMap(FONCTION(1 arg))**
Fonctionne comme la transformation **map** mais si la fonction qui est passée en argument retourne une **liste**, chacune de ses valeurs devient un élément distinct du RDD résultant.
Cela permet à la fonction **flatMap** de renvoyer moins ou plusieurs valeurs.
Exemple :

```
rdd = sc.parallelize(["un mot", "deux"])
rdd2 = rdd.map(lambda x: x.split())
rdd3 = rdd.flatMap(lambda x: x.split())
rdd2.collect()
[["un", "mot"], ["deux"]]  # map
rdd3.collect()
["un", "mot", "deux"]  # flatMap
```

API Python – Transformations

➔ **mapValues(FONCTION(1 arg))**
Applique la fonction passé en argument sur toutes les valeurs des couples clés/valeurs du RDD.
Exige que le RDD source contient des couples clef/valeur.
Exemple :

```
rdd = sc.parallelize([("qui", 20), ("ciel", 12)])
rdd.mapValues(lambda x: x*2).collect()
[('qui', 40), ('ciel', 24)]
# Ce qu'est d'ailleurs équivalent à :
rdd.map(lambda x: (x[0], x[1]*2)).collect()
[('qui', 40), ('ciel', 24)]
```

API Python – Transformations

➔ **flatMapValues(FONCTION(1 arg))**
C'est l'équivalent de **flatMap** mais avec le comportement de **mapValues**.
Exige que le RDD source contient des couples clef/valeur.
Exemple :

```
rdd = sc.parallelize([("qui", 20), ("ciel", 12)])
rdd.flatMapValues(lambda x: (x - 5, 5)).collect()
[('qui', 15), ('qui', 5), ('ciel', 7), ('ciel', 5)]
#  Ce qu'est d'ailleurs équivalent à :
rdd.flatMap(lambda x: [(x[0], x[1] - 5), (x[0], 5)]).collect()
[('qui', 15), ('qui', 5), ('ciel', 7), ('ciel', 5)]
```

API Python – Transformations

➔ **reduceByKey(FONCTION(2 args))**
D'abord regroupe les couples clef/valeur du RDD par clef. Ensuite applique la fonction passé en argument pour réduire chaque groupe et renvoie (un pointeur vers) le RDD résultant.
Exige que le RDD source contient des couples clef/valeur.
C'est l'équivalent d'un **Shuffle + Reduce** Hadoop.
Exemple :

```
rdd = sc.parallelize([("qui", 1), ("ciel", 1), ("qui", 1), ("ciel", 1), ("qui", 1)])
rdd.reduceByKey(lambda a, b: a + b).collect()
[("qui", 3), ("ciel", 2)]
```

API Python – Transformations

- **groupBy(FONCTION(1 arg))**
  Applique la fonction fourni en argument sur chaque élément du RDD source pour générer une clef. Puis regroupe les éléments du RDD par ses clefs génères.
  Renvoie (un pointeur vers) le RDD résultant contenant des couples clé/valeur où la clef est celle produite par la fonction fourni et la valeur est un iterable sur les valeurs du groupe.
  Exemple :

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
res = rdd.groupBy(lambda x: x % 2).collect()
[(0, <pyspark.resultiterable...>),
 (1, <pyspark.resultiterable...>)]
print([(x, sorted(y)) for (x, y) in res])
[(0, [2, 4]), (1, [1, 3, 5])]
```

API Python – Transformations

➔ **groupByKey()**
Regroupe les éléments du RDD par clef distincte. Renvoie (un pointeur vers) le RDD de retour au format similaire à celui de groupBy().
Exige que le RDD source contient des couples clef/valeur.
Exemple:

```
rdd = sc.parallelize(
   [("qui", 1), ("ciel", 1), ("qui", 1), ("ciel", 1), ("qui", 1), ("qui", 1)]
)
res = rdd.groupByKey().collect()
[("qui", <pyspark.resultiterable...>),
 ("ciel", <pyspark.resultiterable...>)]
print([(x, sorted(y)) for (x, y) in res])
[('qui', [1, 1, 1, 1]), ('ciel', [1, 1])]
```

➔ ... c'est l'équivalent de l'étape **Shuffle** de Hadoop (avant Reduce).

API Python – Transformations

➔ **intersection(RDD)**
Renvoie un RDD résultant de l'intersection avec un autre RDD. Exemple :

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([2, 4, 8])
rdd.intersection(rdd2).collect()
[2, 4]
```

➔ **union(RDD)**
Renvoie un RDD résultant de l'union avec un autre RDD. Exemple :

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([2, 4, 8])
rdd.union(rdd2).collect()
[1, 2, 3, 4, 5, 2, 4, 8]
```

API Python – Transformations

➔ **subtract(RDD)**
Renvoie un RDD résultant de la soustraction avec un autre RDD. Exemple :

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([2, 4, 8])
rdd.subtract(rdd2).collect()
[1, 3, 5]
```

➔ **sortBy(FONCTION(1 arg))**
Renvoie un RDD après tri du RDD source; la fonction renvoie la clef à utiliser pour le tri. Exemple :

```
rdd = sc.parallelize([('z', 1), ('y', 2), ('m', 3), ('e', 4)])
rdd.sortBy(lambda x: x[0]).collect()
[('e', 4), ('m', 3), ('y', 2), ('z', 1)]
rdd.sortBy(lambda x: x[1]).collect()
[('z', 1), ('y', 2), ('m', 3), ('e', 4)]
```

API Python – Transformations

- **join(RDD)**
- **fullOuterJoin(RDD)**
- **leftOuterJoin(RDD)**
- **rightOuterJoin(RDD)**

Effectue des jointures entre le RDD courant et le RDD spécifié.

Exige que le RDD source contient des couples clef/valeur.

Les jointures sont effectuées sur les clefs. Ils renvoient des couples clef/valeur dont la clef et celle utilise pour la jointure et la valeur est un tuple composé des valeurs trouvée dans le premier et second RDD.

Pour **fullOuterJoin**, **leftOuterJoin** et **rightOuterJoin**, si une valeur n'a pas été trouvée dans l'autre RDD, **'None'** est utilisé.

API Python – Transformations

→ Exemples :

```
rdd = sc.parallelize([('a', '1'), ('b', 4), ('c', 9)])
rdd2 = sc.parallelize([('a', '0'), ('c', 2), ('d', 7)])

rdd.join(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2))]

rdd.fullOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('b', (4, None)), ('d', (None, 7))]

rdd.leftOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('b', (4, None))]

rdd.rightOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('d', (None, 7))]
```

API Python – Transformations

- **setName(STR)**
  Permet de donner un nom au RDD; Utile pour retrouver les traces associes dans les logs de Spark.
- **cache()**
  Permet de demander à ce que le RDD soit persisté en memoire. Cet appel ne déclenche pas immédiatement l'évaluation;
- **unpersist()**
  Demande au RDD de ne plus être persisté en mémoire.

API Python – Actions

- Les actions déclenchent toutes une évaluation du RDD.

Une liste non exhaustive des actions applicables aux RDD :

- **collect()**
  Récupère le contenu d'un RDD sous la forme d'une liste.
  Non recommandé pour des RDD volumineux. Utile pour le développement
- **count()**
  Renvoie le nombre d'éléments dans le RDD.
- **collectAsMap()**
  Comme collect(), mais retourne une hashmap qui associe clefs et valeurs.
  Exige que le RDD source contient des couples clef/valeur.

API Python – Actions

➔ **countByKey()**
Renvoie le nombre d'éléments dans le RDD pour chaque clef distincte sous la forme d'une hashmap.
Exige que le RDD source contient des couples clef/valeur.

```
rdd=sc.parallelize([('qui', 1), ('qui', 3), ('ciel', 2), ('qui', 2)])
rdd.countByKey()
defaultdict(<type 'int'>, {'qui': 3, 'ciel': 1})
```

➔ **countByValue()**
Renvoie le nombre d'éléments dans le RDD pour chaque valeur distincte sous la forme d'une hashmap.

```
rdd=sc.parallelize(["qui", "qui", "ciel", "qui", "croyait"])
rdd.countByValue()
defaultdict(<type 'int'>, {'qui': 3, 'croyait': 1, 'ciel': 1})
```

API Python – Actions

- **first()**
  Renvoie le premier élément du RDD.
- **getNumPartitions()**
  Renvoie le nombre de partitions du RDD.
- **glom()**
  Renvoie un RDD où chaque élément est une liste contenant l'ensemble des valeurs d'une partition.
  Exemple:

```
rdd = sc.parallelize([1, 2, 3, 4, 5], 2)
rdd2 = sc.parallelize([1, 2, 3, 4, 5], 3)
rdd.glom().collect()
[[1, 2], [3, 4, 5]]
rdd2.glom().collect()
[[1], [2, 3], [4, 5]]
```

API Python – Actions

- **max(key=FONCTION(1 arg))**
  Renvoie l'élément maximal dans le RDD; avec optionnellement une fonction à appeler renvoyer une valeur numérique pour effectuer une comparaison.
- **min(key=FUNCTION(1 arg))**
  L'inverse de max().

```
rddstr=sc.parallelize(["qui", "qui", "ciel", "qui", "croyait"])
rddint=sc.parallelize([4, 8, 15, 16, 23, 42])
rddint.max()
42
rddint.min()
4
rddstr.max(lambda x: len(x))
'croyait'
rddstr.min(lambda x: len(x))
'qui'
```

API Python – Actions

➔ **reduce(FONCTION(2 args))**
Applique la fonction fournie de manière récursive sur le RDD. Renvoie une valeur finale unique comme résultat.
Exemple :

```
rdd2 = sc.parallelize([1, 2, 3, 4, 5])
print(rdd2.reduce(lambda a, b: a + b))
15

def red(a, b):
    if a > b:
        return(a)
    return(b)
print(rdd2.reduce(red))
5
```

API Python – Actions

- **foreach(FONCTION(1 arg))**
  Applique une fonction sur chacun des éléments du RDD.
- **foreachPartition(FONCTION(1 arg))**
  Applique une fonction sur chacune des partitions du RDD.
  Exemple :

```
rdd = sc.parallelize([4, 8, 15], 2)
rdd.foreach(lambda x: print(f"Value: {x}"))
Value: 4
Value: 8
Value: 15
rdd.foreachPartition(lambda x: print(f"Partition has {len(list(x))} elements."))
Partition has 1 elements.
Partition has 2 elements.
```

API Python – Actions

- **isEmpty()**
  Renvoie true si le RDD est vide.
- **repartition(numPartitions)**
  Augmente ou diminue le nombre de partitions du RDD. Provoque un shuffle entier. Pour diminuer le nombre de partitions utilisez coalesce (moins coûteux).

- **saveAsTextFile**
- **saveAsPickleFile**
- **saveAsSequenceFile**

Déjà discutées dans la partie « Sauvegarde de données ».

API Python – Variables broadcast

- Lors de l'exécution d'un programme Spark, les étapes sont envoyées aux différents nœuds de travail.
- Toutes les données pertinentes requises sont également envoyées. Cela inclut les variables locales.
- Par exemple :

```
multiplier = 2
rdd = sc.parallelize([4, 8, 15])
rdd.map(lambda x: x * multiplier).collect()
```

- La variable 'multiplier' sera sérialisée et envoyée avec l'étape pour son exécution.

API Python – Variables broadcast

- Si une variable locale est utilisée dans plusieurs étapes, elle sera transférée plusieurs fois.
- Et si la variable est volumineuse cela cause une utilisation élevée de la bande passante.
- Pour **distribuer efficacement** une telle variable et la mettre en cache sur tous les nœuds de travail du cluster, Spark introduit le concept de **variables broadcast**.
- L'utilisation des variables broadcast évite **les transferts répétés**.
- Ses variables broadcast sont cependant **en lecture seule**.

API Python – Variables broadcast

- Pour créer une variable broadcast : (‘sc’ = l’objet SparkContext)

```
sc.broadcast(VARIABLE)
```

- La fonction renvoie la variable de broadcast; on peut accéder à sa valeur via **.value**.
- Exemple :

```
capitals = {"FR": "Paris", "DE": "Berlin", "UK": "London"}
capitals_bc = sc.broadcast(capitals)
rdd = sc.parallelize(["FR", "DE", "UK"])
rdd.map(lambda x: capitals_bc.value[x]).collect()
['Paris', 'Paris', 'Berlin’]
```

API Python – Les accumulateurs

- Les transformations et actions dans Spark sont effectuées de manière distribuée.
- La mise à jour des variables locales dans les transformations/actions ne mettra pas à jour la variable locale dans le Driver : elle mettra à jour seulement sa copie sur un nœud de travail exécutant l'opération.
- Pour communiquer des informations au Driver, Hadoop offre des compteurs.
- L'équivalent dans Spark : **les accumulateurs**.

API Python – Les accumulateurs

- Comme les Compteurs Hadoop, les accumulateurs Spark ne peuvent être qu'incrémentés.
- Le Driver peut également incrémenter les accumulateurs.
- Pour déclarer un accumulateur : **sc.accumulateur(VALEUR_INITIALE)**
- Pour augmenter sa valeur : **.add(INCREMENT)**

```
count = sc.accumulator(0)
# ...
count.add(1)
```

API Python – Les accumulateurs

→ Exemple :

```
invalide  = 0
valide = sc.accumulator(0)

def mapfunc(x):
    global invalide
    global valide
    invalide = invalide + 1
    valide.add(1)
    return(len(x))

data = sc.parallelize(["lake", "leak", "kale", "boat", "car"])
result = data.map(mapfunc)
print(f"invalide={invalide} ; valide={valide.value}")  # invalide=0 ; valide=0
# Ici, 'valide' vaut toujours 0 car pas encore d'évaluation.
result.collect()  # [4, 4, 4, 4, 3]
print(f"invalide={invalide} ; valide={valide.value}")  # invalide=0 ; valide=5
```

API Python – Exemples

Le compteur d'occurrences de mots, version Spark/Python :

```
from pyspark import SparkContext

sc = SparkContext("local[2]", "Wordcount")
lines = sc.textFile("hdfs:///poeme.txt")
words = lines.flatMap(lambda x: x.split())
tuples = words.map(lambda x: (x, 1))
res = tuples.reduceByKey(lambda a, b: a + b)
res.saveAsTextFile("hdfs:///res")
```

API Python – Exemples

Le détecteur d'anagrammes :

(notez l'usage de .cache() évitant la double évaluation du RDD 'grouped')

```
from pyspark import SparkContext

sc = SparkContext("local[2]", "Anagrammes")
words = sc.textFile("hdfs:///common_words_en_subset.txt")
tuples = words.map(lambda x: (''.join(sorted(list(x))), x))
grouped = tuples.groupByKey().mapValues(lambda x: list(x))
grouped.cache()
filtered = grouped.filter(lambda x: len(x[1]) > 1)
res = filtered.mapValues(lambda x: ", ".join(x))
res.saveAsTextFile("hdfs:///res-filtered")
res2 = grouped.mapValues(lambda x: ", ".join(x))
res2.saveAsTextFile("hdfs:///res-unfiltered")
```

API Python – Exemples

Le compteur d'occurrences de mots, version Spark/Scala :

```
object WordCount {
   def main(args: Array[String]) {
      val conf = new SparkConf().setAppName("Wordcount")
      val sc = new SparkContext(conf)
      val lines = sc.textFile("hdfs:///poeme.txt")
      val words = lines.flatMap(line => line.split())
      val tuples = words.map(word => (word, 1))
      val res = words.reduceByKey(_+_)
      res.saveAsTextFile("hdfs:///res")
   }
}
```

API Python – Conclusion

- Il y a bien sûr beaucoup d'autres fonctions, transformations et actions.
- Il y a d'autres API Spark que nous n'avons pas couvert : SQL, MLlib, etc.
- Pour plus d'informations, se référer à la documentation : https://spark.apache.org/docs/latest/api/python/

Environnement de développement

- API Spark **Java** : même type d'IDE que pour Hadoop ; avec Gradle ou Maven pour les dépendances.
- API Spark **Scala** : même chose.
- API Spark **Python** : un simple éditeur de texte peut suffire ; ou un IDE Python avancé. (PyCharm, Atom...).
- API Spark **R** : un simple éditeur de texte ou un IDE R avancé (RStudio...).

Environnement de développement

Pendant le développement :

- ➔ Utiliser de petits sous-ensembles de grands ensembles de données de production permet l'utilisation de collect(), glom() et similaires.
- ➔ Possible d'extraire un tel sous-ensemble représentatif en utilisant la commande Spark takeSample().
- ➔ Pour les tests, on peut utiliser le **Shell Spark interactif**.

Usage

- Pour soumettre un programme entier à Spark (Java/Scala/Python/R) :
  - **spark-submit**
- Pour accéder àu **Shell Spark interactif** (utile pour rapidement tester de petits programmes Spark) :
  - **spark-shell** (Scala)
  - **pyspark** (Python)
  - **sparkR** (R)
- Si vous utilisez le Shell interactif, vous disposez déjà d'une instance SparkContext avec la variable '**sc**'.
- Par défaut, les deux commandes fonctionnent en mode local : ils vont exécuter le programme sur un cluster Spark simulé.

Usage

➔ Les deux commandes Spark ont deux arguments importants :

--master [URL] --deploy-mode [client|cluster]

➔ **--master** permet de spécifier l'URL du cluster manager.
- Accepte la même syntaxe utilisée lors de l'instanciation de l'objet **SparkContext**.
- Exemples : 'spark://...' , 'yarn', 'mesos://...', 'local[N]'

➔ **--deploy-mode** permet de spécifier le mode d'exécution.
- Mode **Client** : le Driver s'exécute à l'endroit où le programme est initialement lancé.
- Mode **Cluster** : le Driver s'exécute dans un nœud sur le cluster.

Usage

→ Pour soumettre un programme Spark **Java** ou **Scala** :

```
spark-submit [OPTIONS] --class DRIVER_CLASSPATH JARFILE
```

→ Par exemple :

```
spark-submit --master yarn --deploy-mode cluster --class org.embds.wcount WordCount.jar
```

Usage

- Pour soumettre un programme **Python**, nous pouvons également inclure des bibliothèques supplémentaires en utilisant le paramètre :

```
--py-files
ZIPFILE
```

- Par exemple :

```
spark-submit --master yarn --deploy-mode cluster --py-files opencv.zip word_count.py
```

Usage

- Il y a évidemment de nombreux autres arguments pour ces deux commandes.
- Pour plus d'informations sur leur usage, se référer à la documentation: https://spark.apache.org/docs/latest/

DATUM ACADEMY

Écosystème autour d'Hadoop

ZooKeeper, Pig, Hive, Hbase, MongoDB, Oozie, Mahout

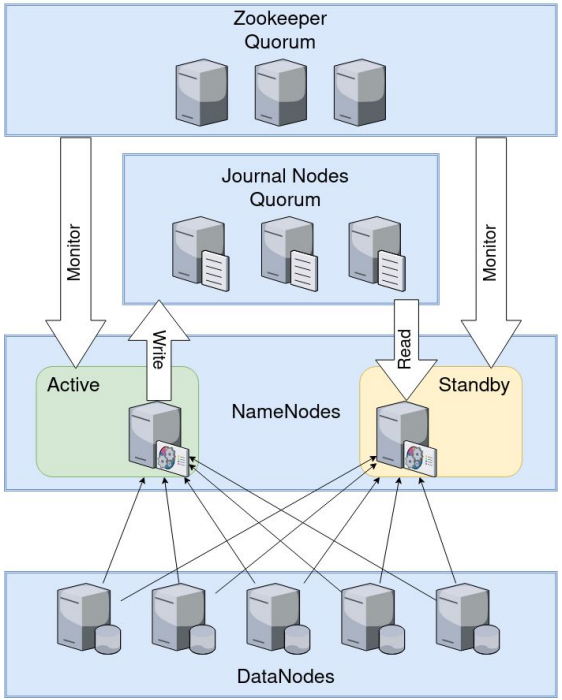
Zookeeper

- Un service hautement disponible.
- Maintient des données de coordination. (Comme un très fiable petit système de fichiers distribué)
- Notifie aux clients les changements dans ces données.
- Surveille les défaillances des clients.
- Assure la haute disponibilité des services HDFS et YARN
- Est capable de faire beaucoup plus: mettre en œuvre des protocoles de consensus, de gestion de groupe, d'élection du leader etc.
- Pour plus d'information: https://zookeeper.apache.org/

Zookeeper exemple Haute disponibilité HDFS

- Il y a plusieurs Namenodes déployés (mais qu'un est active et les autres sont en standby)
- Le Namenode active persiste tous les changements fait sur HDFS sur un répertoire NFS partagé ou via la composante Quorum Journal Manager.
- Dans cet exemple Zookeeper est en charge de:
  - Détection de panne (Surveillance du bon fonctionnement des Namenodes).
  - Élection d'un nouveau Namenode en tant qu'active en cas de panne.

Zookeeper exemple Haute disponibilité HDFS



Apache PIG

- Créé à l'origine chez Yahoo.
- Est une plateforme d'analyse et traitement des données massives.
- Fonctionne avec un langage de haut niveau (**Pig Latin**) qui se compile en tâches Map-Reduce et les exécute (en parallèle) sur un cluster (Hadoop ou Spark).
- Ainsi il abstrait la programmation du paradigme Java MapReduce
- Pour plus d'informations - voir la documentation officielle: http://pig.apache.org/docs/r0.17.0/
- Note: Apache PIG est de plus en plus remplacé par Apache Hive et Apache Spark (la dernière release de PIG date de 2017)

Apache PIG - Exemple comptage des mots (PigLatin)

➔ Exemple :

```
-- Charger le fichier ‘poeme.txt’ depuis HDFS par ligne.
data = LOAD 'poeme.txt’ AS line;
-- Découper chaque ligne par mot.
words = FOREACH data GENERATE FLATTEN(TOKENIZE(line, ' ‘)) AS word;
-- Regrouper la sortie par mot.
grouped = GROUP words BY word;
-- Pour chaque groupe compte le nombre d'occurrences.
wordcount = FOREACH grouped GENERATE group, COUNT(words);
-- Sauvegarder le résultat dans le dossier ‘wordcount-output’ sur HDFS.
STORE wordcount INTO 'wordcount-output';
```

Apache Hive

- Initialement développé par Facebook.
- Est un logiciel de gestion de données facilitant la lecture et l'écriture de grands ensembles de données résidant dans un stockage distribué à l'aide de SQL.
- Fonctionne avec Hive Query Language (HiveQL) - très similaire à SQL.
- Est souvent utilisé pour des tâches telles que l'extraction/transformation/chargement (ETL), la création de rapports et l'analyse des données.

Apache Hive

- Interagit avec les fichiers stockés sur HDFS avec SQL. (les utilisateurs définissent le schéma des données pour les fichiers)
- Supporte beaucoup des formats de fichiers - TXT, CSV, TSV, ORC, Parquet, AVRO, etc. mais aussi des formats sur mesure.
- Hive propose 2 different types de tables:
  - **Externe** (external): Les données à la source n'appartiennent pas à Hive et si on supprime les tables dans Hive - ses données seront conservées.
  - **Interne** (internal): Les données à la source appartiennent à Hive, et si on supprime les tables dans Hive - ses données seront également supprimées.
- Possible d'interconnecter avec d'autres sources des données (ex. Mysql, MongoDB, etc.) avec des connecteurs ODBC (Open Database Connectivity) ou JBDC (Java Database Connectivity)
- Pour plus d'information: https://hive.apache.org/

Apache Hive - Exemple HiveQL

```
$ # beeline est un CLI pour interagir avec Hive.
$ beeline
beeline> -- Connexion a Hive.
beeline> !connect jdbc:hive2://{HIVESERVER}:{PORT} {USERNAME}
beeline> -- Création d'une table Hive externe avec 2 colonnes pour des fichiers TSV (séparés par tab).
beeline> -- Chaque ligne dans ses fichiers dans le dossier "/dictionary" est sous forme "MOT
DESCRIPTION".
beeline>CREATE EXTERNAL TABLE dictionary (word STRING, description STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/dictionary';
beeline> -- Par exemple: Cherchons la description du mot "SWEET" dans notre table "dictionary".
beeline>SELECT description FROM dictionary WHERE word = "SWEET";
```

Apache HBase

- Est une base de données Open Source, NoSQL, distribuée et tolérante aux pannes, modelée à partir de BigTable de Google.
- Fournit des capacités de type BigTable au-dessus de HDFS*.
- *(peut aussi fonctionner avec le système de fichiers local, S3 et d'autres systèmes de fichiers MAIS suppose implicitement que les données sont stockées de manière fiable).
- Bien adapté au traitement des données en temps réel ou à l'accès aléatoire et cohérent en lecture/écriture à de grands volumes de données.
- Fonctionne bien avec Hive.
- Peut servir à la fois d'entrée et de sortie pour les tâches map/reduce.
- Pour plus d'informations: https://hbase.apache.org/

MongoDB

- MongoDB est un SGBD (système de gestion de bases de données) NoSQL orientée document.
- Il s'agit d'un logiciel libre; initialement propriétaire (2007), il a été plus tard diffusé en licence AGPLv3 (2009).
- Stocke les données sous la forme de documents rédigés dans un langage proche de JSON: BSON (Binary JSON).
- Est extrêmement scalable, capable de stocker de très larges quantités de données.
- Il est développé et maintenu essentiellement par l'entreprise MongoDB Inc., mais également par de nombreux bénévoles et entités tierces qui l'utilisent.
- Un connecteur Hadoop est par ailleurs disponible; permettant de lire les données d'entrée d'un programme Hadoop directement depuis MongoDB, et d'écrire les résultats d'un tel programme directement dans MongoDB également.
- Site officiel: https://www.mongodb.com/

Apache Oozie

- Est un système de planification des tâches - gère et exécute les tâches sur Hadoop.
- Peut exécuter plusieurs tâches Hadoop dans un ordre séquentiel ou/et en parallèle.
- 3 types de tâches (niveau d'abstraction):
  - *Oozie Workflow jobs* - représentent des séquences d'actions à exécuter.
  - *Oozie Coordinator Jobs* - déclenchent les "Oozie Workflow jobs" en fonction du temps, des données disponibles ou d'événements externes.
  - *Oozie Bundles* - représentent un ensemble des "Oozie Coordinator Jobs".
- Pour plus d'informations: https://oozie.apache.org/

Apache Airflow

- Airflow a débuté en tant que projet open source chez Airbnb, puis a rejoint la fondation officielle Apache en 2016.
- Est une plateforme permettant de créer, de planifier et de contrôler les workflows de manière programmatique (scripts python).
- Fournit une interface web puissante pour surveiller, planifier et gérer les workflows.
- Très extensibile: des intégrations existent pour de nombreux outils d'ingénierie des données couramment utilisés.
- Pour plus d'informations: https://airflow.apache.org/

Apache Mahout

- Apache Mahout est une bibliothèque open-source dédiée à l'algèbre linéaire et au Machine Learning à l'échelle distribuée.
- Offre diverses algorithmes de filtrage collaboratif, clustering, classification, etc.
- Est extensible, permettant de développer rapidement des algorithmes distribués en utilisant un DSL (langage spécifique au domaine) mathématiquement expressif, similaire au langage R.
- Initialement, il fonctionnait au-dessus de Hadoop. Utilise maintenant Apache Spark comme noyau de calcul distribué et peut être étendu à d'autres noyaux de calcul distribués.
- Pour plus d'informations: https://mahout.apache.org/