Institut Supérieur d'Informatique et Mathématiques
Monastir

Big Data et NoSQL

Chapitre 3: Traitement des données massives avec Spark

--Partie 1--

Mastère de Recherche en Génie Logiciel – Niveau 2

2021-2022

Asma KERKENI asma.kerkeni@gmail.com

Objectifs

- Au terme de ce chapitre, vous serez capable de:
  - Distinguer les modes de traitement de données.
  - Citer les limites de Hadoop Map/Reduce.
  - Présenter Spark et l'intégrer dans l'écosystème du BigData.
  - Expliquer la différence clé entre Hadoop et Spark.
  - Maitriser les différents modes de traitement de données.
  - Enumérer les caractéristiques des RDDs.

Plan

- Contexte
- Spark : C'est quoi?
- Spark : Un bref Historique
- Spark: quel langages?
- Spark : cas d'utilisation
- Spark Vs Hadoop
- RDD: Définition, caractéristiques, opérations
- Exemple
- Reprise sur panne

Références du cours

- **Livres:**
  - Juvénal Chokogoue, Maitriser l'utilisation des technologies Hadoop: Initiation à l'écosystème Hadoop. Paris, 2018.
  - H.Karau, A. Konwinski, P. Wendell, M. Zaharia, "Learning Spark", O'Reilly Media, 2015.
- **Cours:**
  - Cours Big Data de Mme Lilia Sfaxi: http://liliasfaxi.wixsite.com/liliasfaxi/big-data
  - Cours de Conception et Développement d'applications d'Entreprise à Large échelle de Jonathan Lejeune : https://pages.lip6.fr/Jonathan.Lejeune/CODEL.php
  - https://openclassrooms.com/fr/courses/4297166-realisez-des-calculs-distribues-sur-des-donnees-massives/4308671-domptez-les-resilient-distributed-datasets
- **Article:**
  - Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012, April). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (pp. 2-2). USENIX Association.
- **Vidéos:**
  - Atelier Spark sur la chaine youtube TechWall

Contexte:

Modes de traitement de données

5

- Deux mode de traitement de données:
  - traitement par lot (Batch)
  - traitement de flux (Streaming)

Contexte:

Modes de traitement de données

6

- Traitement par lot : *Batch Processing*
  - Moyen efficace de traiter de grands volumes de données.
  - Les données sont collectées, stockées et traitées, puis les résultats sont fournis.
  - Le traitement est réalisé sur l'ensemble des données qui doivent être prêtes avant le début du job.
  - Plus concerné par **le débit** (nombre d'actions réalisées en une unité de temps) que par **la latence** (temps requis pour réaliser l'action) des différents composants du traitement.

Contexte:

Modes de traitement de données

7

- Traitement par lot: *Batch Processing (suite)*
  - Inconvénients:
    - N'est pas approprié pour des traitements en ligne ou temps réel.
    - Il n'est pas possible d'exécuter des travaux récursifs ou itératifs de manière inhérente: Certains modèles de calcul batch (comme Hama et Mahout) offrent l'itérativité mais ils ne sont pas adaptés au traitement interactif.
    - Latence d'exécution élevée: Produit des résultats sur des données relativement anciennes.
  - Cas d'utilisation:
    - Les chèques de dépôt dans une banque sont accumulés et traités chaque jour.
    - Les statistiques par mois/jour/année.
    - Factures générées pour les cartes de crédit (en général mensuelles).

Contexte:

Modes de traitement de données

8

- Traitement de flux (à la volée) : *Stream Processing*
  - Les traitements se font sur un élément ou un petit nombre de données récentes.
  - Le traitement est relativement simple.
  - Doit compléter chaque traitement en un temps proche du temps-réel.
  - Les traitements sont généralement indépendants.
  - **Asynchrone:** les sources de données n'interagissent pas directement avec l'unité de traitement en streaming, en attendant une réponse.
  - La latence de traitement est estimée en secondes.

Contexte:

Modes de traitement de données

9

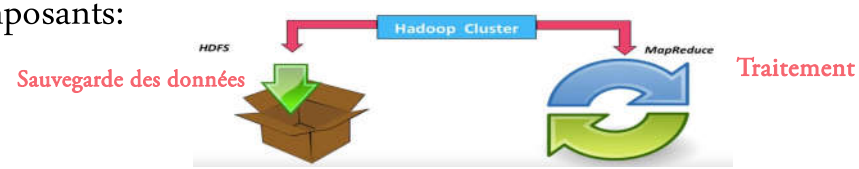
- Traitement de flux : *Stream Processing*
  - Inconvénients:
    - Pas de visibilité sur l'ensemble des données.
    - Complexité opérationnelle élevée
    - Plus complexes à maintenir que les traitements batch.
    - Le système doit être toujours connecté, toujours prêt, avoir des temps de réponses courts, et gérer les données à l'arrivée.
    - Risque de perte de données.
  - Cas d'utilisation:
    - Recommandation en temps réel: Prise en compte de navigation récente, géolocalisation, publicité en ligne, re-marketing.
    - Surveillance de larges infrastructures.
    - Agrégation de données financières à l'échelle d'une banque.

Contexte :

Hadoop Map/Reduce: Quel type de traitement?

10

- Hadoop (principale plateforme Big Data), est composé deux composants:



- Map-Reduce simplifie énormement l'analyse des données massives sur des grands clusteurs peu fiables.
- Hadoop Map-Reduce est un exemple de système adapté **uniquement pour traitement par lot.**

→ Une première limite de Hadoop. Existe-il d'autres limites?

Contexte :

Limites de Hadoop M/R

11

- Nécessité d'écriture sur disque après une opération map ou reduce pour permettre aux mappers et aux reducers de communiquer entre eux.
  - Ceci permet une certaine tolérance aux pannes.
  - **Cependant,** ces écritures et lectures sont coûteuses en temps, surtout pour les algorithmes itératifs.
- Limite du jeu d'expressions composé exclusivement d'opérations map et reduce.
  - Difficulté d'exprimer des opérations complexes en n'utilisant que cet ensemble de deux opérations.

Contexte:

Besoins des applications Big Data

12

- Des besoins plus complexes dans le domaine Big Data ne cessent d'émerger:
  - **Algorithmes itératifs plus complexes** (apprentissage automatique, traitement de graphe, etc).
  - **Plus de requêtes adhoc interactives** pour le data mining.
  - Besoin de **plus de traitement en temps réel** (comme par exemple la classification des spams, la détection de fraude, les tweets…):

→ Vers des approches In-memory.

Contexte:

Traitement In-Memory

13

- *In-Memory Proceesing*: Traitement des données en mémoire:
  - Charger tout le fichier de données en mémoire.
  - C'est le mode utilisé par défaut dans les applications de statistiques, data mining et reporting.
  - Par conséquent: faible latence

Technology	Latency (s)	Data transfer rate(Go/s)
Disque dur	10^{-2}	0.15
SSD	10^{-4}	0.5
DDR3 SDRAM	10^{-8}	15

  - Est-il possible de traiter des données massives en mémoire?
    - Oui: deux modes: Clusters Shared Memory et Clusters Shared-nothings.

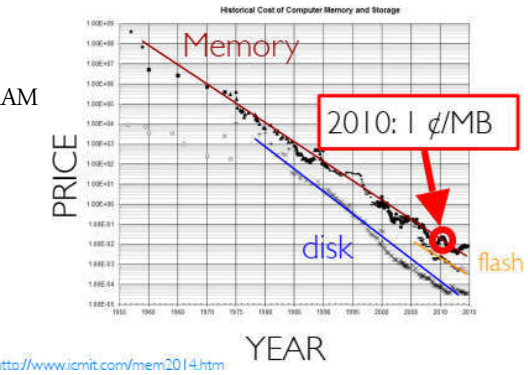
Contexte:

Traitement In-Memory

14

- Baisse des prix des RAM:
  - En 2000, prix d'1 Mo de RAM: **1,12$.**
  - En 2005, il passe à **0,185$.**

    Hadoop est apparu et une machine de 4Go RAM est considérée puissante.
  - En 2010, il tombe à 0,00122$
  - Aujourd'hui, le prix de 1 Go de RAM <10 $
  - Il est normal de trouver des serveurs avec 256 Go de RAM.



- Conséquences:
  - Upsizing des machines: ajout des barrettes mémoire.
  - Offre des solutions in-memory comme Oracle Exadata (Traiter des données >20To)

Contexte:

Besoins des applications Big Data

16

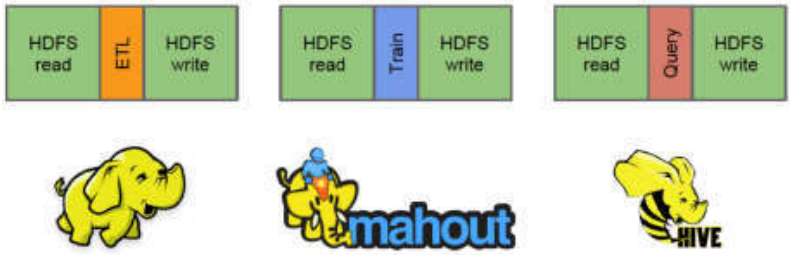
- Des frameworks spécifiques pour répondre à ces besoins.
  - Diversité des APIs,
  - Vision peu unifiée,
  - interactions coûteuses entre logiciels.

General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

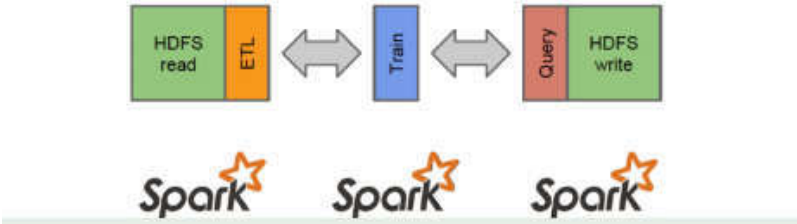
Solution: Vers un Framework unifié

17

Plusieurs frameworks spécialisés



Framework unifié



Spark : C'est quoi?

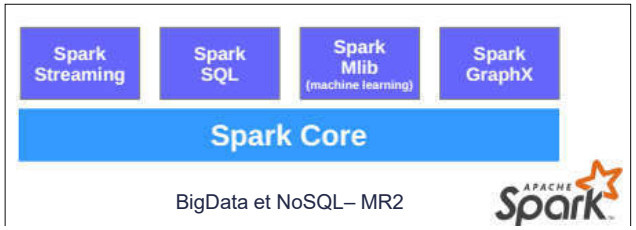
18

- Apache Spark est un **framework de calcul distribué à grande échelle** s'inscrivant dans la mouvance BigData.
- Apparu en 2010, Spark se présente comme une **extension du pattern d'architecture** ***Map/Reduce***. Il offre des fonctionnalités plus puissantes que le ***Map/Reduce***.
- C'est une approche ***in-Memory*** (*shared-nothing*) et tolérente aux fautes.
- C'est un framework supportant les traitements temps réel.

Spark: c'est quoi?

19

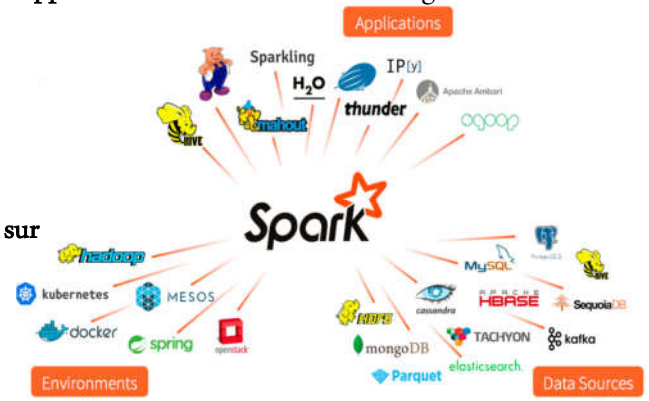
- Une plateforme unifiant plusieurs librairies:
  - **Spark Core** : librairie basique
  - **Spark Streaming** : Librairie pour flux de données temps réel
  - **Spark SQL** : Librairie pour manipuler des données structurées
  - **Spark MLib** : Librairie pour analyse et fouille de données (machine learning)
  - **Spark GraphX** : Librairie pour calcul de graphes



Spark : C'est quoi?

20

Il **est utilisé par des applications** dans le domaine du big data et du machine learning.



Spark **peut s'exécuter sur plusieurs plateformes**: Hadoop, Mesos, en standalone ou sur le cloud.

Il peut **accéder diverses sources de données,** comme HDFS, Cassandra, HBase et S3.

Spark : Un bref Historique

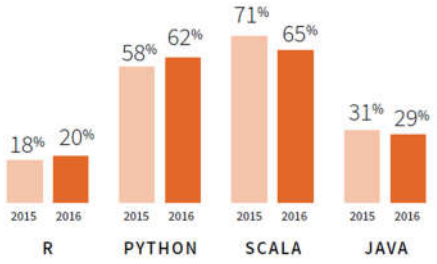
21

- Spark est développé à UC Berkeley AMPLab en 2009 dans le cadre de la these de **Matei Zaharia**.
- Il est passé en open source sous forme de projet Apache en 2010 (licence BSD).
- Il devient le projet le plus important de Apache en Février 2014.
- En 2014, **Matei Zaharia** a fondé Databricks (fournit le support commercial).
- C'est maintenant le projet open source le plus actif en BigData.
- Version actuelle: 2.3.2 (Depuis Septembre 2018).

amplab UC BERKELEY — The Apache Software Foundation Community-led development since 1999. — databricks making big data simple

Spark et langages

- Spark est polyglotte:
  - Il est écrit en Scala
  - Mais il supporte 4 langages: Scala, Java, Python (PySpark) et R (SparkR).



Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Standalone Programs
•Python, Scala, & Java

Interactive Shells
•Python & Scala

Performance
•Java & Scala are faster due to static typing
•...but Python is often fine

Source: https://adtmag.com/articles/2016/10/03/databricks-spark-survey.aspx

Spark : cas d'utilisation

- Recommandations (article, produit,....).
- Traitement de fichiers texte
- Détection de fraude
- Analyse de logs
- Analytics

Spark Vs Hadoop

- Spark offre un moteur de traitement de données qui remplace Hadoop Map/Reduce.
- Spark ne possède pas un système de stockage propre, il s'appuie sur HDFS (ou d'autres sources de données).
- En 2014, Spark détrône Hadoop Map-Reduce en battant le record du tri le plus rapide de 100 To.
  - Hadoop Map-Reduce : 72 minutes avec 2100 machines.
  - Spark : 23 minutes avec 206 machines.

	Data Size	Time	Nodes	Cores
Hadoop MR (2013)	102.5 TB	72 min	2,100	50,400 physical
Apache Spark (2014)	100 TB	23 min	206	6,592 virtualized

Spark Vs Hadoop

Java Hadoop

```
public class WordCount {
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();
   public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
     String line = value.toString();
     StringTokenizer tokenizer = new StringTokenizer(line);
     while (tokenizer.hasMoreTokens()) {
       word.set(tokenizer.nextToken());
       context.write(word, one);
 } } }
 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {
     int sum = 0;
     for (IntWritable val : values) {
       sum += val.get();
     }
     context.write(key, new IntWritable(sum));
 } }
 public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   job.setMapperClass(Map.class);
   job.setReducerClass(Reduce.class);
   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);
   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));
   job.waitForCompletion(true);
 }
```

Scala Spark

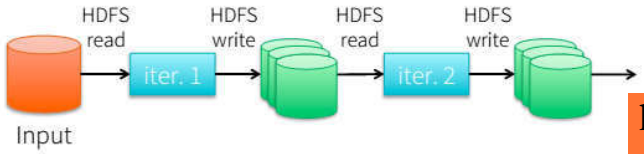
```
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

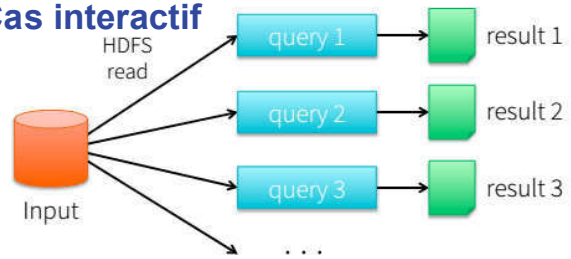
Spark Vs Hadoop : Différence clé

Modèle de Partage de données avec Map/Reduce

Cas séquentiel



Cas interactif

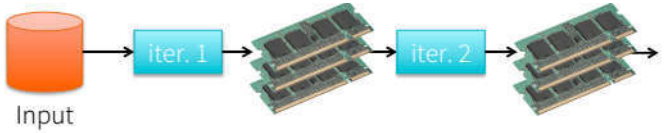


lent en raison de la réplication, de la sérialisation et des E/S, mais nécessaire pour la tolérance aux pannes.

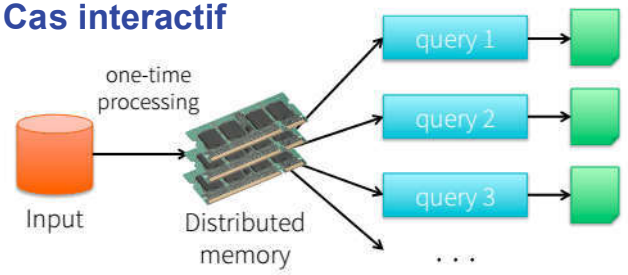
Spark Vs Hadoop : Différence clé

Modèle de partage de Spark

Cas séquentiel



Cas interactif



10-100 fois plus rapide que le réseau et le disque grâce aux mécanisme de mise en mémoire avec les RDDs.

RDD: *Resilient Distributed Datasets*

NSDI '12 Home
Registration Information
Discounts
Organizers
At a Glance
Technical Sessions
Poster and Demo Session
Birds-of-a-Feather Sessions
Workshops

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

CONNECT WITH US

Authors:
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, *University of California, Berkeley*

Awarded Best Paper!

Awarded Community Award Honorable Mention!

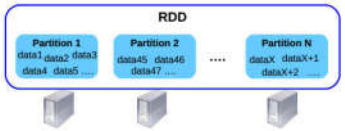
RDD: *Resilient Distributed Datasets*

- C'est le concept central de Spark.
- Il s'agit de larges ***hashmaps*** stockées en mémoire et sur lesquelles on peut appliquer des traitements.
- Un RDD résulte de la **transformation** d'un autre RDD ou d'une **création** à partir de données existantes.
- Un RDD représente un **Graph Acyclique Direct** des différentes opérations à appliquer aux données chargées.

RDD: Caractéristiques

30

- **Résilient:** tolérant aux fautes, si les données en mémoire (ou sur un nœud) sont perdus, il est possible de le récupérer.
- **Distribué:** réparti sur plusieurs machines afin de paralléliser les traitements.
- **Immutable :** en lecture seul (pas d'opération de mise à jour). Un traitement appliqué à un RDD donne lieu à la création d'un nouveau RDD.
- **Ordonnée:** chaque élément a un index.
- **Redondé :** limite le risque de perte de données.
- **En mémoire:** Un RDD est stocké en mémoire.
- **Partitionnée:** sur un ensemble de machines.
- **Typé:** un RDD possède un type.



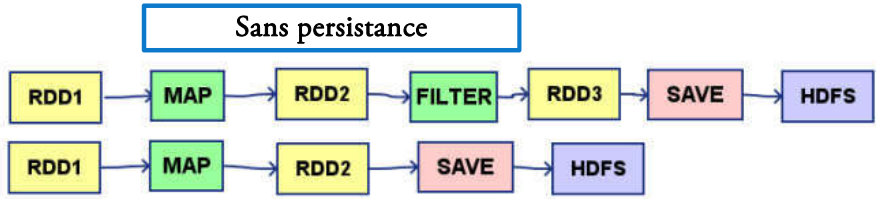
RDD: Caractéristiques

31

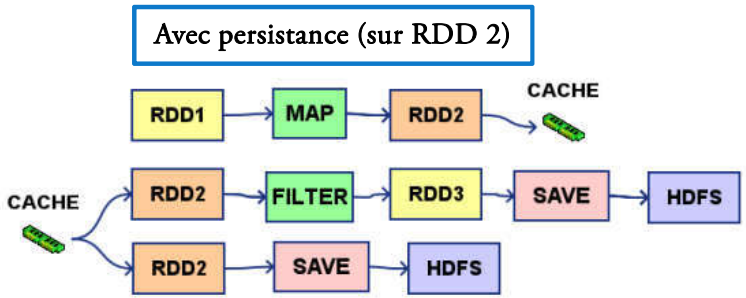
- **Persistant:** Un RDD peut être marqué comme **persistant** pour une réutilisation future, ses partitions sont sauvegardés sur les nœuds qui l'héberge. Spark fournit 3 options de stockage pour les RDDs persistants :
  - stockage en mémoire comme objet java désérialisé,
  - stockage en mémoire comme objet java sérialisé,
  - stockage sur disque.
- ***Lazy evaluated:***
  - Un RDD ne contient pas vraiment de données, mais seulement un traitement. Le traitement n'est effectué que lorsque cela apparaît nécessaire. On appelle cela l'évaluation paresseuse *(Lazy evaluation).*
  - L'évaluation paresseuse évite le calcul inutile. Ceci favorise l'optimisation du traitement.

RDD et persistance

32



- Sans persistence, le map entre RDD1 et RDD2 est exécuté deux fois.



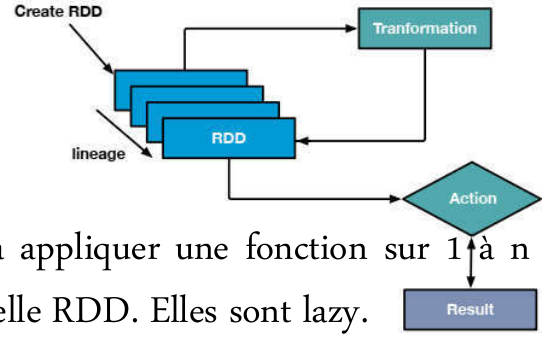
RDD et persistance

33

- Les RDDs persistés sont stockés dans le cache des nœuds executor.
- **Utilité de la persistance** pour les opérations interactives effectuant plusieurs requêtes sur un même dataset intermédiaire:
  - Travailler sur un RDD virtuel (non calculé) permet de définir des résultats intermédiaires sans les calculer immédiatement, et donc de passer à l'étape suivante sans attendre un long traitement,
  - La sauvegarde explicite d'un résultat intermédiaire permet de gagner du temps lorsqu'on sait qu'il servira de point de départ de plusieurs requêtes. ET tandis qu'il s'exécute en arrière-plan, sa vue « virtuelle » peut être utilisés dans d'autres requêtes.

RDD: opérations

- RDD supporte deux type d'opérations:
  - Transformation
  - Action



- **Une transformation** consiste à appliquer une fonction sur 1 à n RDD et à retourner une nouvelle RDD. Elles sont lazy.
- **Une action** consiste à appliquer une fonction et à retourner une valeur au programme driver pour par exemple les afficher sur l'écran ou les enregistrer dans un fichier.

RDD: Opérations

- **Exemples de transformations:**
  - `map()` : une valeur → une valeur
  - `mapToPair()` : une valeur → un tuple
  - `filter()` : filtre les valeurs/tuples
  - `groupByKey()` : regroupe les valeurs par clés
  - `reduceByKey()` : aggrège les valeurs par clés
  - `join()`, `cogroup()`... : jointure entre deux RDD
- **Exemples d'actions:**
  - `count()` : compte les valeurs/tuples
  - `saveAsHadoopFile()` : sauve les résultats au format Hadoop
  - `foreach()` : exécute une fonction sur chaque valeur/tuple
  - `collect()` : récupère les valeurs dans une liste (`List<T>`)

RDD et les transformations *"lazy"*

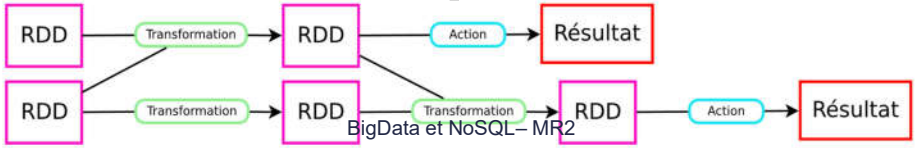
- RDD[T]: Une *sequence* d'objets de type T.
- Les transformations sont "lazy":

map(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]

https://github.com/mesos/spark/blob/master/core/src/main/scala/spark/PairRDDFunctions.scala

Calculs distribués sous forme de graphe DAG

- Dans un **graphe acyclique orienté (DAG),** les nœuds sont les RDD et les résultats.
- Les connexions entre les nœuds sont soit des transformations, soit des actions. Ces connexions sont orientées (un seul sens de passage).
- Le graphe est dit *acyclique* car aucun RDD ne permet de se transformer en lui-même via une série d'actions.
- Lorsqu'un nœud devient indisponible, il peut être regénéré à partir de ses nœuds parents: **tolérance aux pannes.**

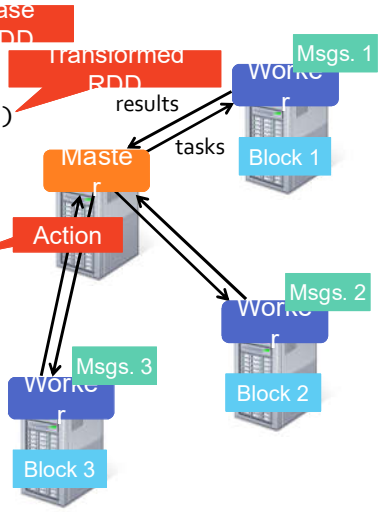


Example : Log Mining

38

- Charger des messages d'erreurs à partir de log et recherche interactive de divers patron.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
// rien ne se passe avant l'appel des actions
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

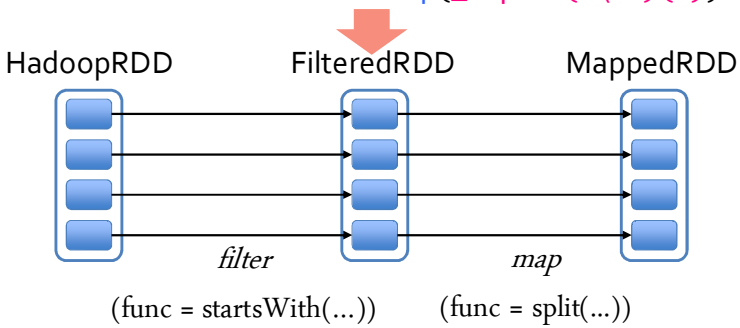


Spark: Reprise sur panne

40

- Les RDD suivent le graphique des transformations qui les ont construits : lignée (***lignage***) pour reconstruire les données perdues.
- **Exemple:**

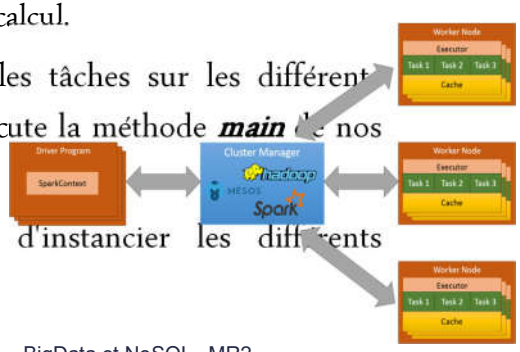
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Spark: Architecture

42

- Les applications Spark s'exécutent comme un ensemble de processus indépendants sur un cluster coordonné par un objet ***SparkContext*** du programme principal appelé **Driver program** (mode Master Slave).
- Un cluster Spark est composé de :
  - un ou plusieurs ***workers*** : chaque worker instancie un executor chargé d'exécuter les différentes tâches de calcul.
  - ***un driver*** : chargé de répartir les tâches sur les différents **executors**. C'est le driver qui exécute la méthode ***main*** de nos applications.
  - ***un cluster manager*** : chargé d'instancier les différents workers.

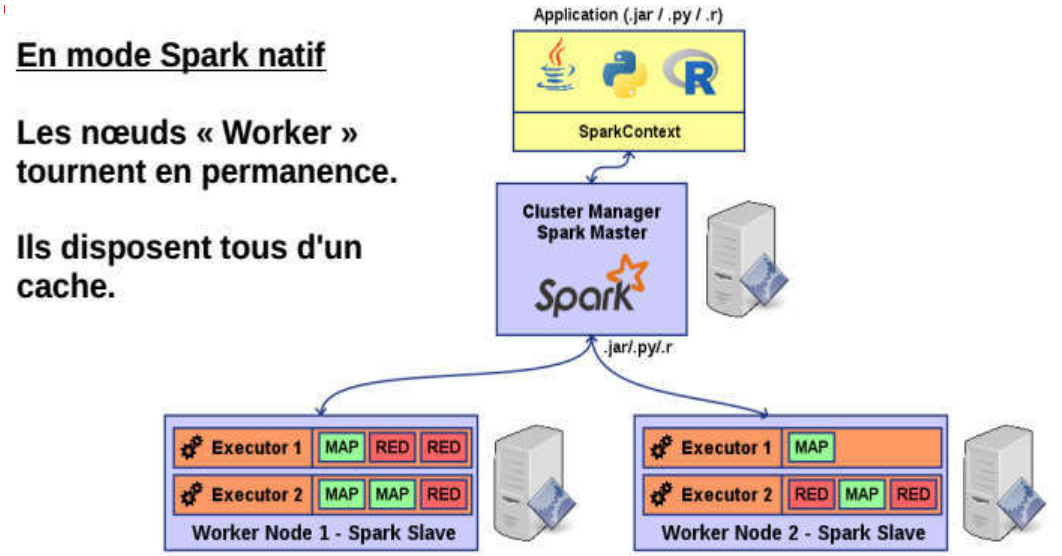


Spark: Architecture

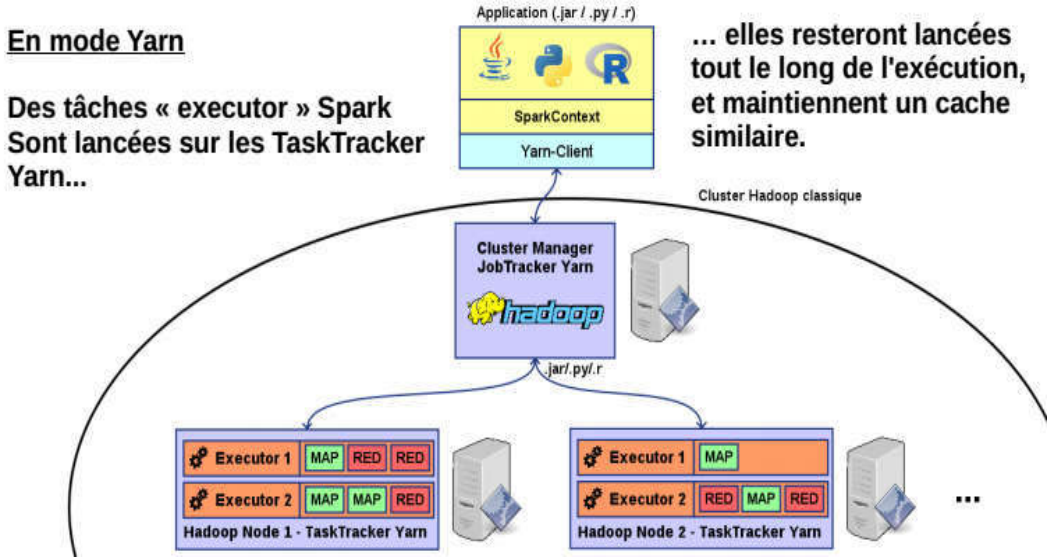
43

- Il existe trois plateformes existantes sur lequel on peut exécuter un application Spark :
  - **Spark standalone :** un gestionnaire de ressources dédié uniquement aux déploiements d'application Spark et qui fonctionne sur un schéma maître-esclaves.
  - **Apache Yarn :** un gestionnaire de déploiement de JVM.
  - **Apache Mesos :** un gestionnaire de déploiement de conteneurs type Docker.

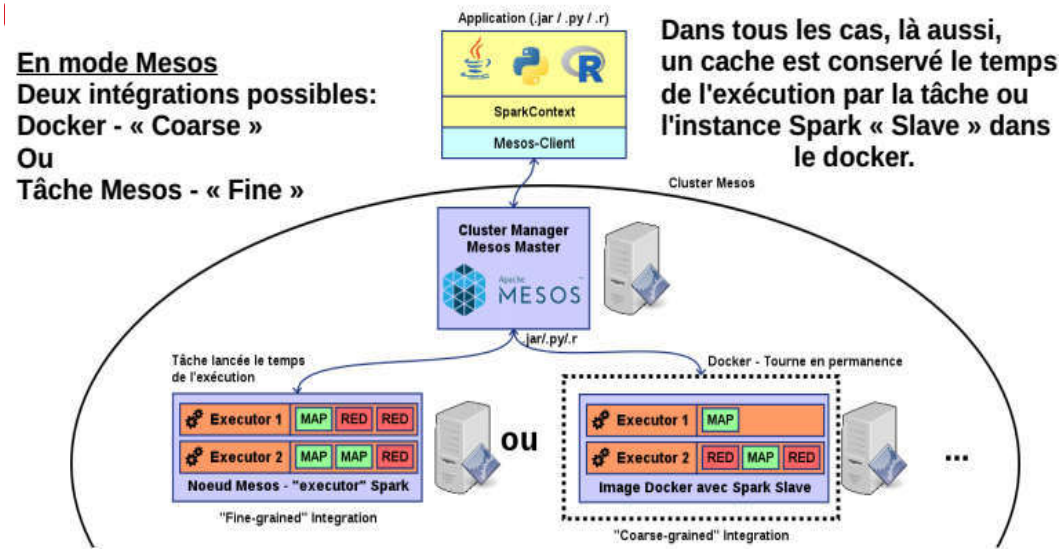
Spark: Architecture



Spark: Architecture



Spark: Architecture



Spark : Mode de fonctionnement

- A l'instar de Yarn, Spark peut fonctionner en trois modes :
  - **mode local** : le job s'exécutera sur une seule machine (sans gestionnaire de ressources distribuées) en mode multi-threadé qui permet de profiter d'un minimum de parallélisation même si ce mode reste réservé aux phases de déploiement et de débogage.
  - **mode pseudo-distribué** : le gestionnaire de ressources se résume au déploiement du master et d'un worker sur la machine locale.
  - **mode full-distribued** : le gestionnaire de ressource est entièrement déployé sur l'ensemble des machines du cluster.

FIN