Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El Mahraz
Fès

Cours Big Data

Master Recherche en Informatique Décisionnelle et Vision Intelligente (MIDVI)

Préparé par :
Pr. Noura AHERRAHROU

Spark : le moteur in-memory distribué d'Hadoop

- Malgré sa simplicité, le MapReduce n'est pas adapté à toutes les problématiques, plus précisément celles qui impliquent un traitement **interactif** ou **itératif**.
- Pour contourner ce problème, plusieurs acteurs sur le marché ont proposé des modèles de calcul alternatifs au MapReduce. **Apache Spark** est une alternative à Hadoop-MapReduce pour le calcul distribué qui vise à résoudre ces deux problèmes.

Spark VS Hadoop MapReduce

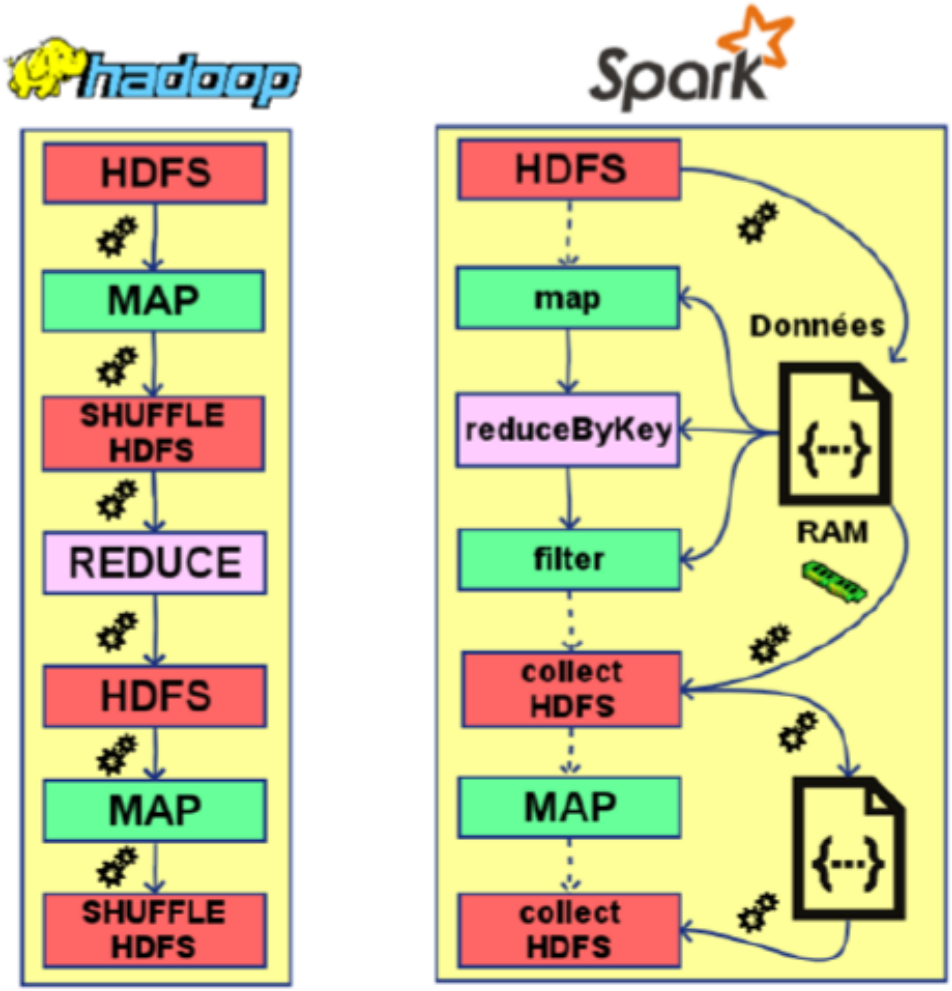
- Apache Spark a été créé au départ pour pallier les limitations de Hadoop Map/Reduce. En effet, ce dernier, qui s'avère être idéal pour implémenter des applications à base de traitements par lot (batch processing), montre certaines limites quand il s'agit d'applications à faible latence et à traitements itératifs, comme par exemple pour le machine learning ou pour les algorithmes à base de graphes.
- Spark permet de généraliser le modèle de traitement de Map-Reduce, tout en améliorant de manière conséquente les performances et l'utilisabilité.

Spark VS Hadoop MapReduce

- De plus, Spark garde la trace des données que chacun de ces opérateurs produit, et permet aux applications de stocker ces données en mémoire, améliorant ainsi considérablement la performance en permettant d'éviter les accès coûteux au disque.
- Hadoop Map Reduce est efficace pour les traitements à passe unique (un seul passage sur les données), mais pas pour des traitements à plusieurs passes. Pour ces traitements plus complexes, il faut enchaîner une série de jobs Map-Reduce et les exécuter séquentiellement, chacun de ces jobs présentant une latence élevée et aucun ne pouvant commencer avant que le précédent ne soit totalement terminé. Avec Spark, il est possible de développer des pipelines de traitement de données plus complexes.

Spark VS Hadoop MapReduce

- Spark fournit également des traitements Map-Reduce, mais en exploitant efficacement la mémoire, tout en utilisant le linkage en cas d'échec si nécessaire.
- Spark 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 également explicitement lui demander de conserver des données en RAM, parce qu'on sait qu'elles seront nécessaires entre plusieurs instances d'écriture disque.



Spark vs Hadoop MapReduce

La différence fondamentale entre Hadoop MapReduce et Spark est que Spark écrit les données en RAM, et non sur disque. Ceci a plusieurs conséquences importantes sur la rapidité de traitement des calculs ainsi que sur l'architecture globale de Spark.
Pour mémoire, voici quelques ordres de grandeurs approximatifs relatifs au transfert de données en RAM et sur disque :

Technologie	Latence (s)	Taux de transfert (Go/s)
Disque dur	10^{-2}	0.15
DDR3 SDRAM	10^{-8}	15

Spark vs Hadoop MapReduce

on peut observer en passant de Hadoop à Spark une accélération des temps de traitement d'un facteur 10 à 100 ! Dit comme ça, ça peut sembler magique, mais le choix de stocker les données intermédiaires en RAM a des conséquences sur l'architecture même de Spark. En particulier, comment avec des données en RAM, garantir une tolérance aux pannes ? Dès qu'une machine devient indisponible, les données qu'elle stockait en RAM deviennent également indisponibles.

Fonctionnement général du Spark

- Spark ne fonctionne pas selon l'architecture shared-memory, mais sur un cluster configuré en shared-nothing.
- La clé de la stratégie Spark pour faire du traitement massivement parallèle en mémoire est le RDD – *Resilient Distributed Dataset* (table partagée et résiliente).
- Il s'agit d'une abstraction de données distribuées qui permet aux développeurs d'effectuer des calculs parallèles en mémoire sur un cluster de façon complètement tolérante aux pannes. Le RDD a été créé pour résoudre le problème que posent les algorithmes itératifs et les calculs interactifs au MapReduce.

Fonctionnement général du Spark

- Les RDD sont dits « **distribués** » parce qu'ils sont partitionnés et partagés entre les nœuds du cluster. Ils sont également **résilients** parce qu'ils conservent les données en mémoire. Par conséquent, en cas de panne du noeud, ils sont automatiquement reconstructibles. Ces deux caractéristiques sont la clé de l'efficacité de Spark.
- Formellement, le RDD est une collection partitionnée d'objets accessibles en lecture seule.

Fonctionnement général du Spark

- Étant donné qu'un RDD est une abstraction, il n'a pas d'existence réelle ; dès lors, il doit être explicitement créé ou instancié à travers des opérations déterministes sur des fichiers de données existants ou sur d'autres instances de RDD. Ces opérations d'instanciation sont appelées **transformations**.
- En fait, **Spark est une implémentation du RDD**, tout comme Hadoop est une implémentation du MapReduce.
- L'écriture d'un programme en Spark commence par la définition d'un ou de plusieurs RDD à travers des **transformations**, soit sur des fichiers de données localisés ou non sur le HDFS, soit sur des instances de RDD existantes (conservées en mémoire ou sérialisées sur le disque dur). Elle finit par l'utilisation des **actions.**

Fonctionnement général du Spark

- Étant donné qu'un RDD est une abstraction, il n'a pas d'existence réelle ; dès lors, il doit être explicitement créé ou instancié à travers des opérations déterministes sur des fichiers de données existants ou sur d'autres instances de RDD. Ces opérations d'instanciation sont appelées **transformations**.
- En fait, **Spark est une implémentation du RDD**, tout comme Hadoop est une implémentation du MapReduce.
- L'écriture d'un programme en Spark commence par la définition d'un ou de plusieurs RDD à travers des **transformations**, soit sur des fichiers de données localisés ou non sur le HDFS, soit sur des instances de RDD existantes (conservées en mémoire ou sérialisées sur le disque dur). Elle finit par l'utilisation des **actions.**

Quelques transformations et actions disponibles un RDD

Transformations	Actions
Map()	Count()
Filter()	Collect()
Flatmap()	Reduce()
Sample()	Lookup()
Groupbykey()	Save()
Reducebykey()	Persist()
Union()	Sum()
Join()	Variance()
Cogroup()	Mean()
Crossproduct()	Min()
Mapvalues()	Foreach()
Sort()	
Partitionby()	

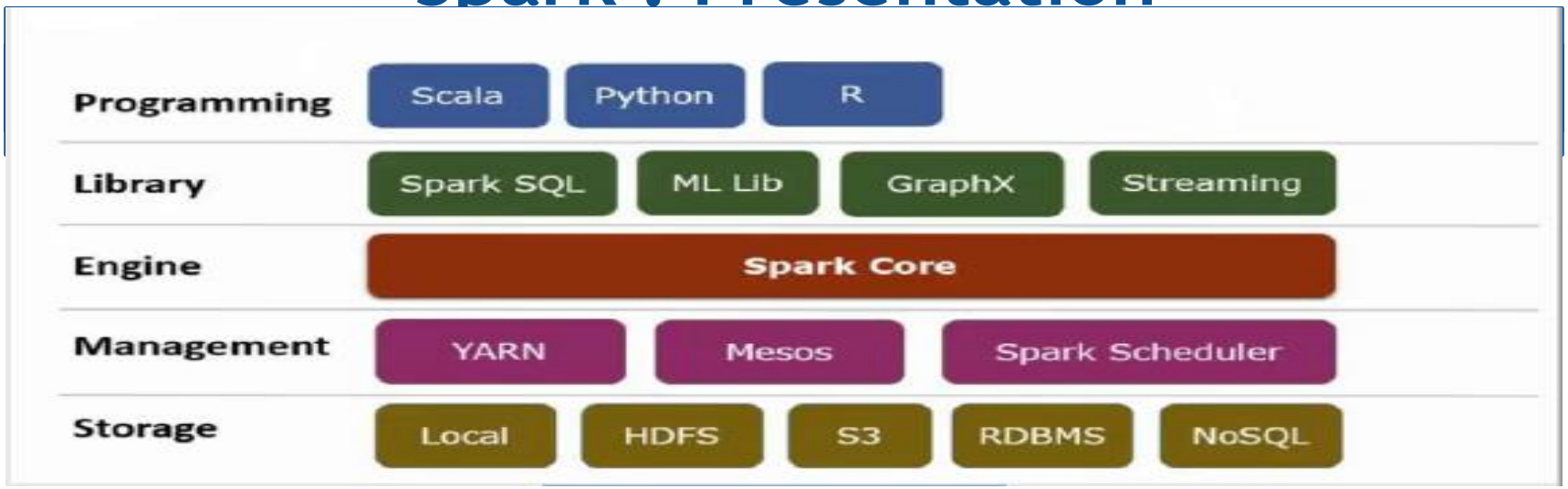
RDD : Resilient Distributed Datasets

- Dans la pratique, les développeurs écrivent ce que l'on appelle un *Spark driver* (ou pilote Spark), qui est un peu l'équivalent d'un job MapReduce en Hadoop. Ce pilote se connecte aux processus Workers qui tournent au niveau des noeuds de données du cluster et définit ensuite un RDD sur lequel il déclenche une ou plusieurs action(s).
- À la différence d'un cluster Hadoop, les processus Workers de Spark ne s'achèvent pas à la fin de l'exécution du pilote ; ils sont continuellement actifs, ce qui est nécessaire pour faire persister les partitions de RDD en mémoire cache et éviter ainsi les réplications sur disque. **La persistance en mémoire cache des partitions du RDD est la clé de la performance et de l'efficacité de Spark**.

RDD : Resilient Distributed Datasets

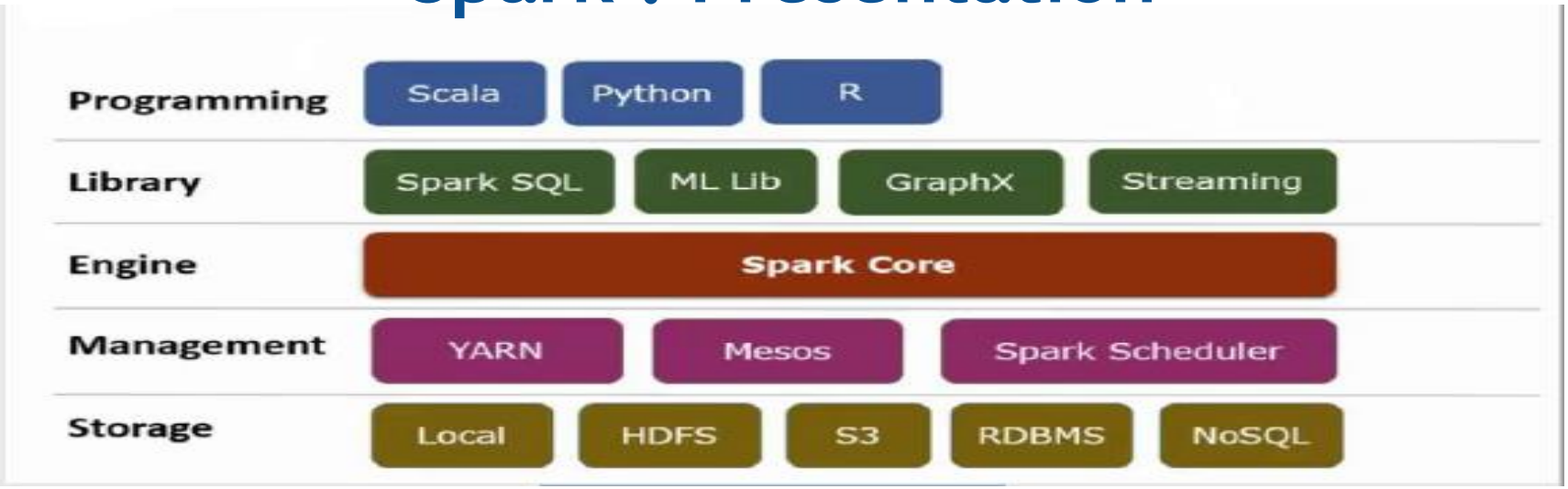
- En effet, les Workers utilisent la mémoire cache pour conserver les données dans les partitions de RDD en vue de la réutilisation ultérieure – évitant ainsi la réplication des données sur disque –, nécessaire dans Hadoop pour garantir la tolérance aux pannes du cluster.
- C'est grâce à cette utilisation optimale de la mémoire et au partage résilient du RDD que Spark affiche des résultats dix fois supérieurs à ceux d'Hadoop sur des travaux interactifs, tout en conservant les attributs de scalabilité et haute disponibilité.

Spark : Présentation



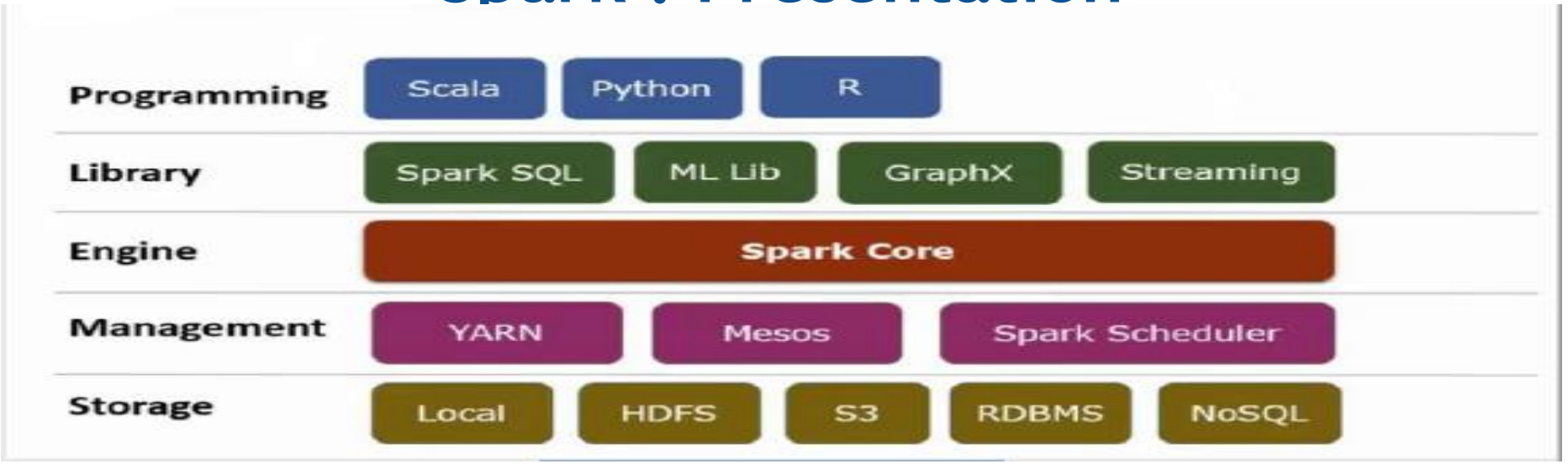
- Développé en Scala (langage orienté objet dérivé de Java et incluant de nombreux aspects des langages fonctionnels). Quatre langages supportés:
  - Scala
  - Java
  - Python (PySpark)
  - R (SparkR)
- Performances et fonctionnalités équivalentes pour les 4.

Spark : Présentation



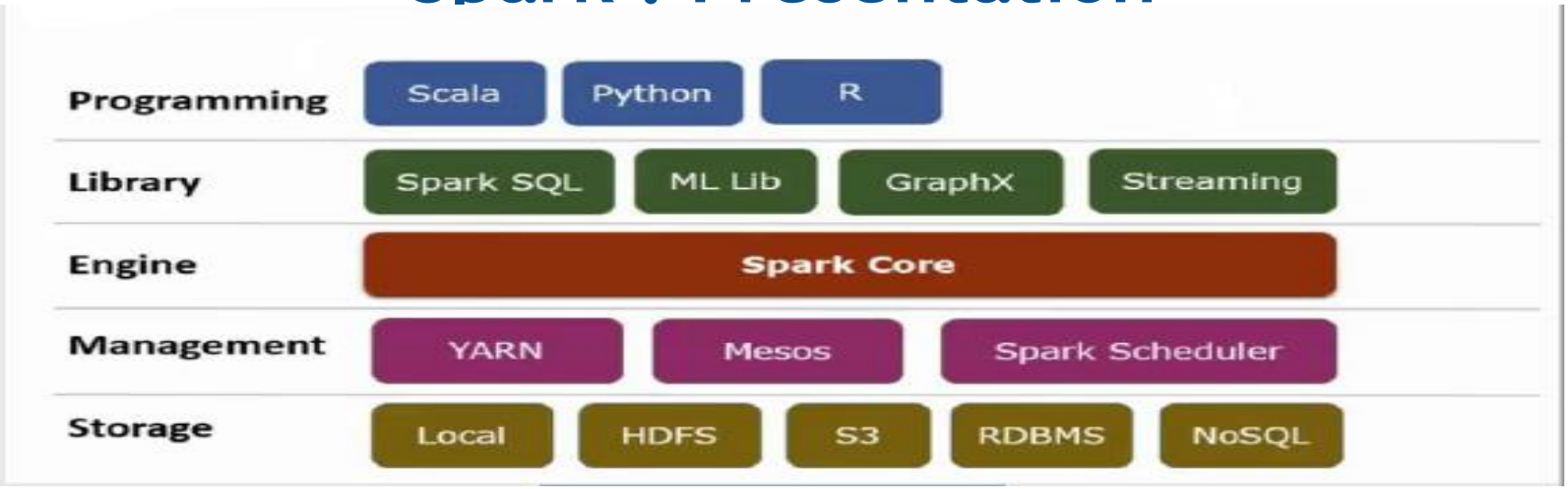
- **Spark SQL** est un module qui permet d'écrire des requêtes SQL ou HiveQL et de les exécuter sur Spark.
- **GraphX** sert à analyser et exécuter des requêtes sur un graphe (le graphe d'un réseau social, par exemple).

Spark : Présentation



- **Spark Streaming** est un module qui permet de développer des applications de données produites en temps réel ou en streaming.
- **MLib** est l'équivalent de Mahout MLib en Spark. C'est une bibliothèque d'algorithmes parallélisés adaptés à Spark, qui permet de traiter la majorité des problèmes d'apprentissage automatique de façon interactive. Attention : certains algorithmes d'apprentissage automatique ou de machine learning ne se prêtent pas au calcul parallèle ; ils ne sont pas dans MLib.

Spark : Présentation

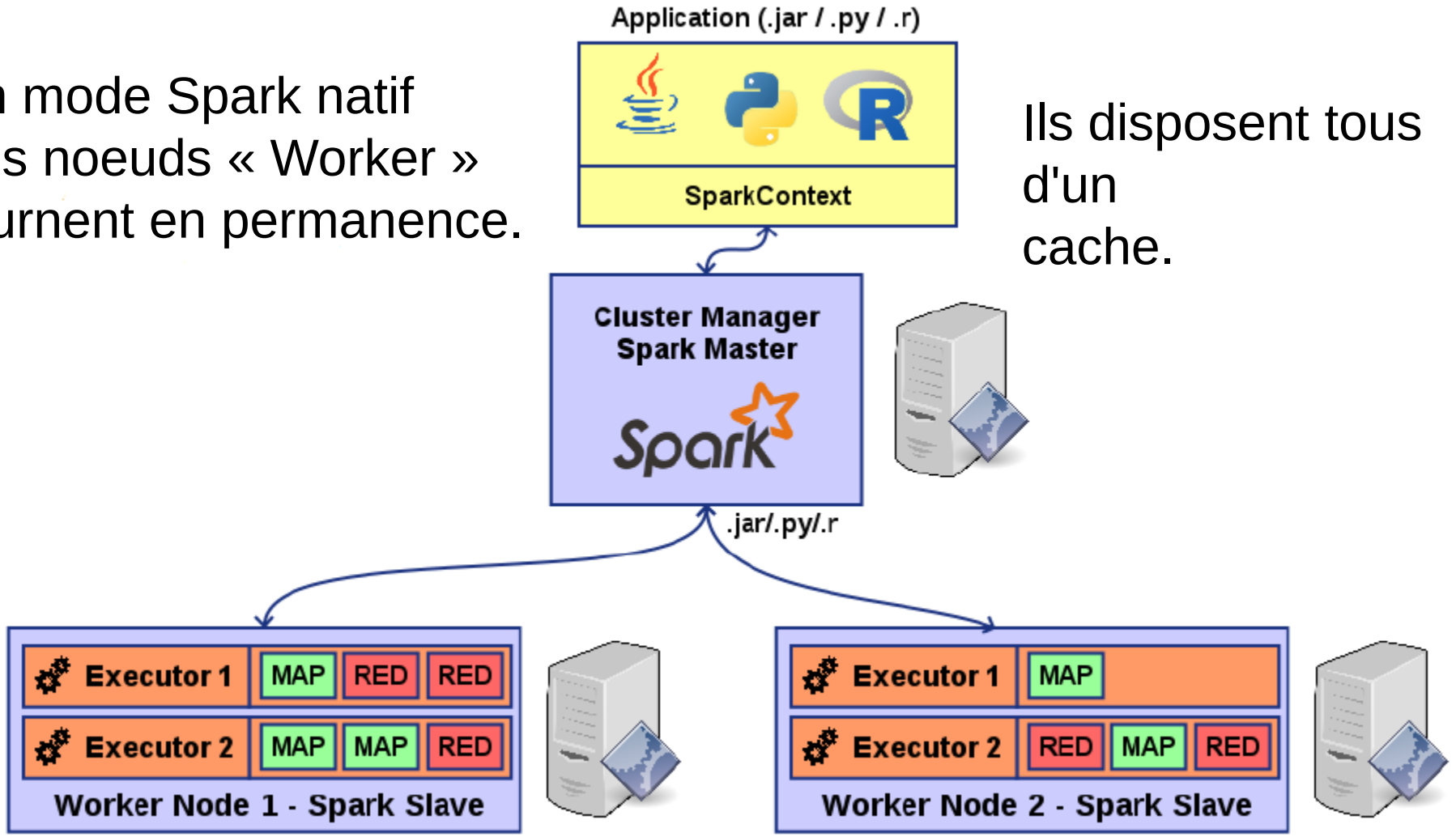


Trois modes d'exécution:

- ❑ Cluster Spark natif.
- ❑ Hadoop (YARN).
- ❑ Mesos (Spark natif + scheduler Mesos).

Spark : Architecture en mode Spark natif

En mode Spark natif
Les noeuds « Worker »
tournent en permanence.

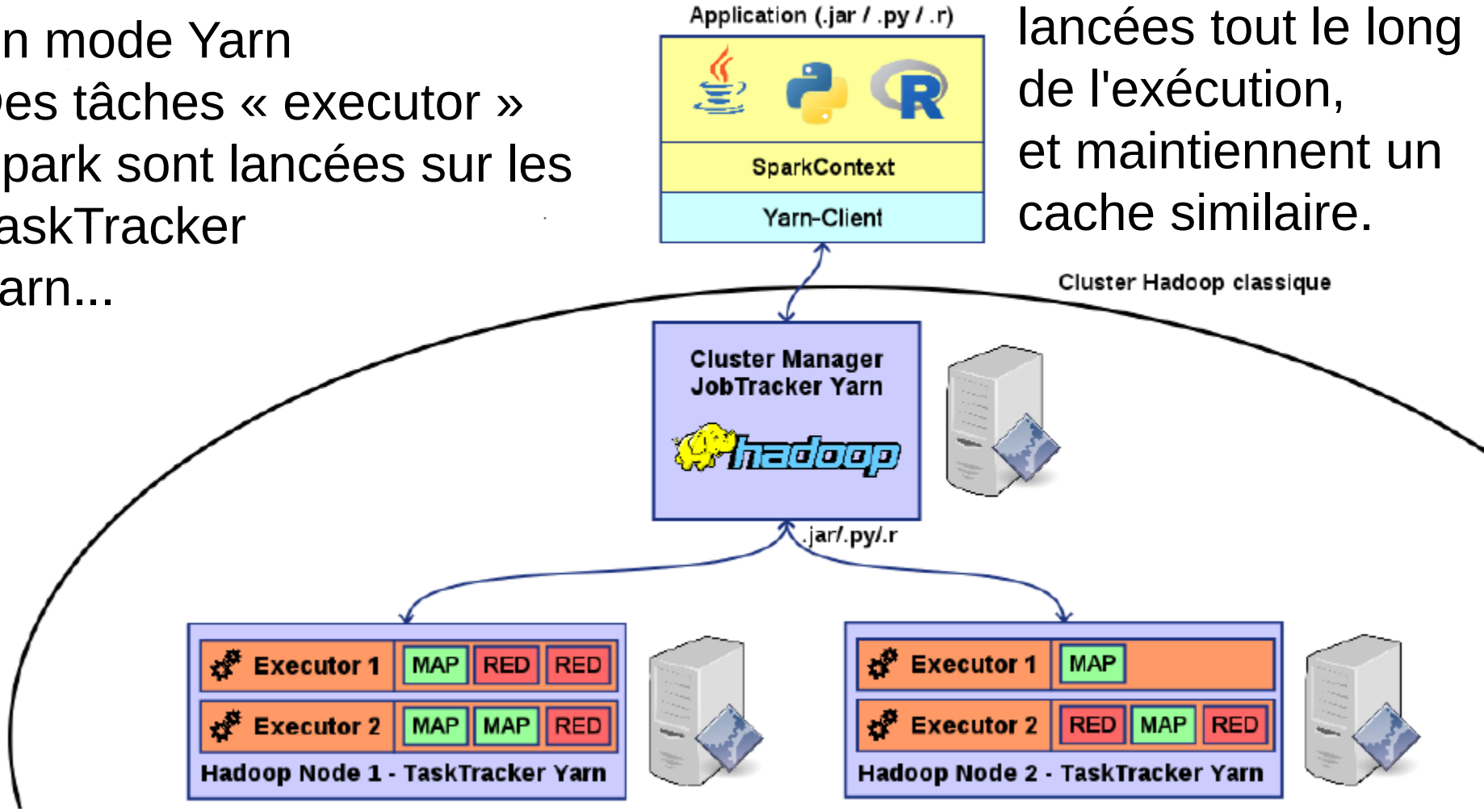


Ils disposent tous d'un
cache.

Spark : Architecture en mode Hadoop (YARN)

En mode Yarn
Des tâches « executor » Spark sont lancées sur les TaskTracker Yarn...

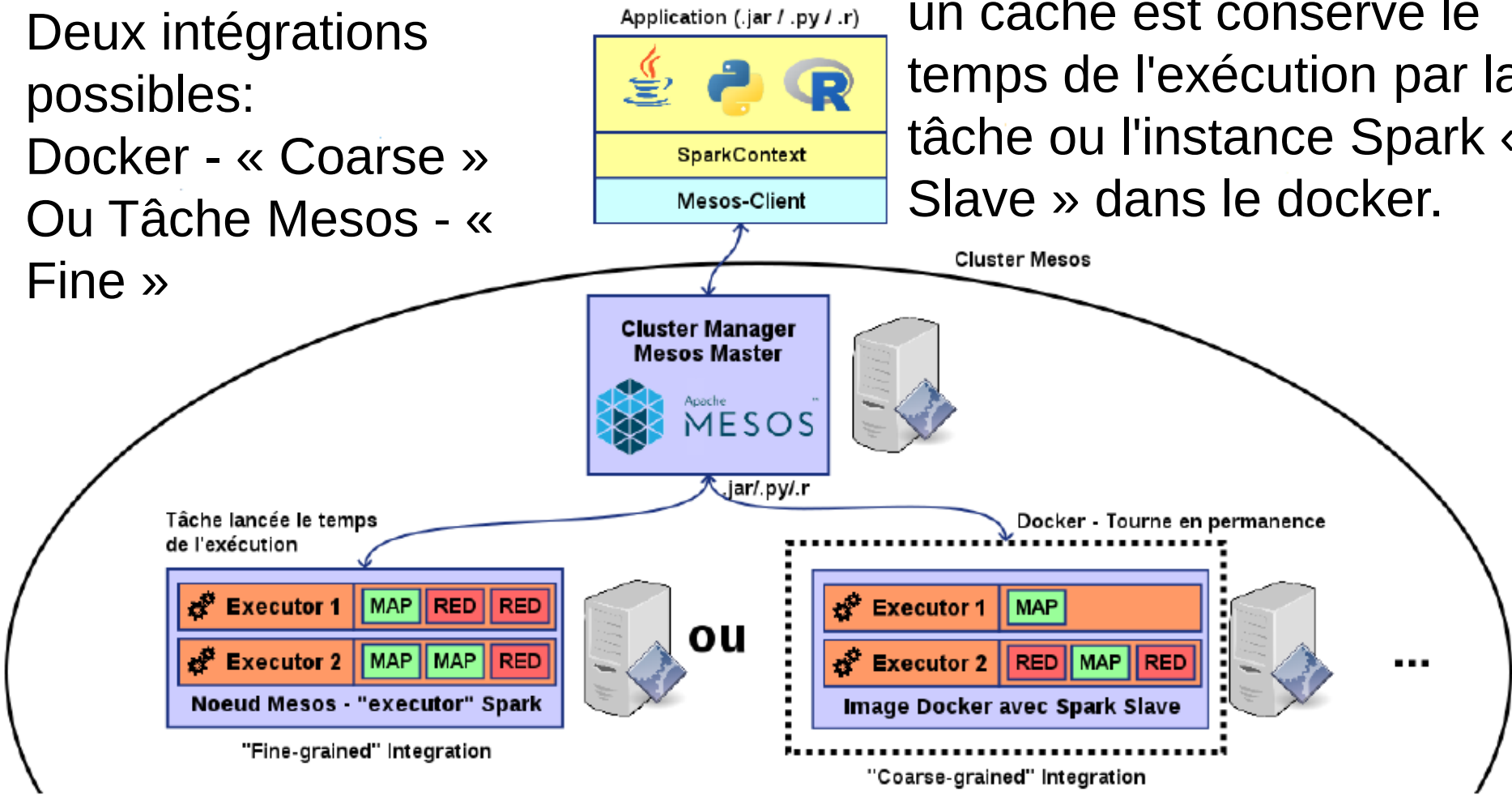
… elles resteront lancées tout le long de l'exécution, et maintiennent un cache similaire.



Spark : Architecture en mode Mesos

En mode Mesos
Deux intégrations possibles:
Docker - « Coarse »
Ou Tâche Mesos - « Fine »

Dans tous les cas, là aussi, un cache est conservé le temps de l'exécution par la tâche ou l'instance Spark « Slave » dans le docker.



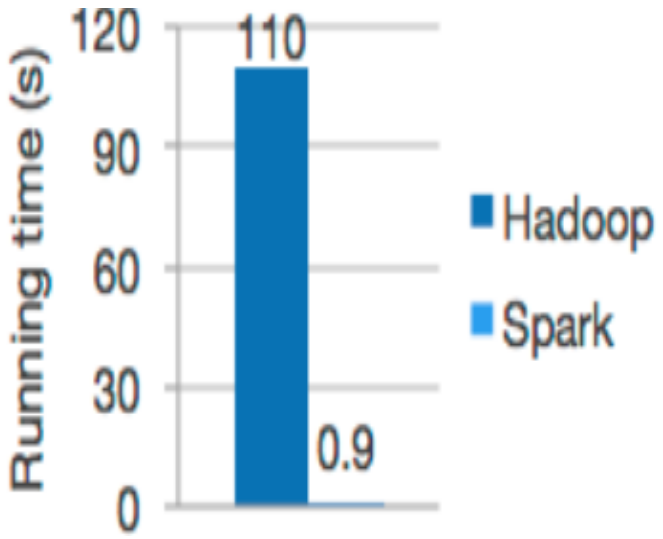
Spark : Avantages

Performances supérieures à celles de Hadoop pour une large quantité de problèmes; et presque universellement au moins équivalentes pour le reste.

- API simple et bien documentée; très simple à utiliser. Paradigme plus souple qui permet un développement conceptuellement plus simple.
- Très intégrable avec d'autres solutions; peut très facilement lire des données depuis de nombreuses sources, et propose des couches d'interconnexion très faciles à utiliser pour le reste (API dédiée Spark Streaming, Spark SQL).
- APIs dédiées pour le traitement de problèmes en machine learning (Spark MLlib) et graphes (Spark GraphX).

Spark : Performances

- Fortement dépendantes du problème mais d'une manière générale supérieures à Hadoop. Dans le cas de problèmes complexes, effectuant de nombreuses opérations sur les données et notamment sur les mêmes données antérieures, fortement supérieures (jusqu'à 100x plus rapide).



- Dans les faits, un programme Spark sera souvent susceptible d'appeler plusieurs opérations « map », « reduce », ou « shuffle », l'une à la suite de l'autre ou en parallèle; et c'est là que Spark montre vraiment ses avantages en terme de performances.

Spark : Inconvénients

- Spark consomme beaucoup plus de mémoire vive que Hadoop, puisqu'il est susceptible de garder une multitude de RDDs en mémoire. Les serveurs nécessitent ainsi plus de RAM.
- Son cluster manager (« Spark Master ») est encore assez immature et laisse à désirer en terme de déploiement / haute disponibilité / fonctionnalités additionnelles du même type; dans les faits, il est souvent déployé via Yarn, et souvent sur un cluster Hadoop existant.

Spark : WordCount Example en Java

```
public class WordCount {
public static void main(String[] args)
{
SparkConf conf = new SparkConf().setAppName("Wordcount");
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines=sc.textFile("file:///test.txt");
JavaRDD<String> words=lines.flatMap(new
FlatMapFunction<String, String>() {
public Iterator<String> call(String s)
{ return(Arrays.asList(s.split(" ")).listIterator()); }
});
```

Spark : WordCount Example en Java

```
JavaPairRDD<String, Integer> tuples=words.mapToPair(new
PairFunction<String, String, Integer>() {
public Tuple2<String, Integer> call(String s)
{ return(new Tuple2(s, 1)); }
});
JavaPairRDD<String, Integer> res=tuples.reduceByKey(new
Function2<Integer, Integer, Integer>() {
public Integer call(Integer a, Integer b)
{ return(a+b); }
});
res.saveAsTextFile("hdfs:///res-java");
}
}
```

Spark : WordCount Example en Scala

Ouvrez un Shell Scala, puis exécuter les commandes Scala suivante : **bin\spark-shell**

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
val txtFile = "README.md"
val txtData = sc.textFile(txtFile)
txtData.cache()
val wcData = txtData.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
wcData.collect().foreach(println)
```

Spark : WordCount Example en 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')
```