Spark

Laurent Lapasset

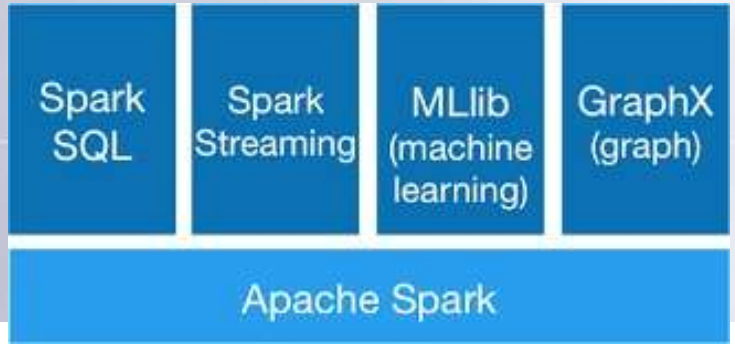
laurent.lapasset@recherche.enac.fr

2024

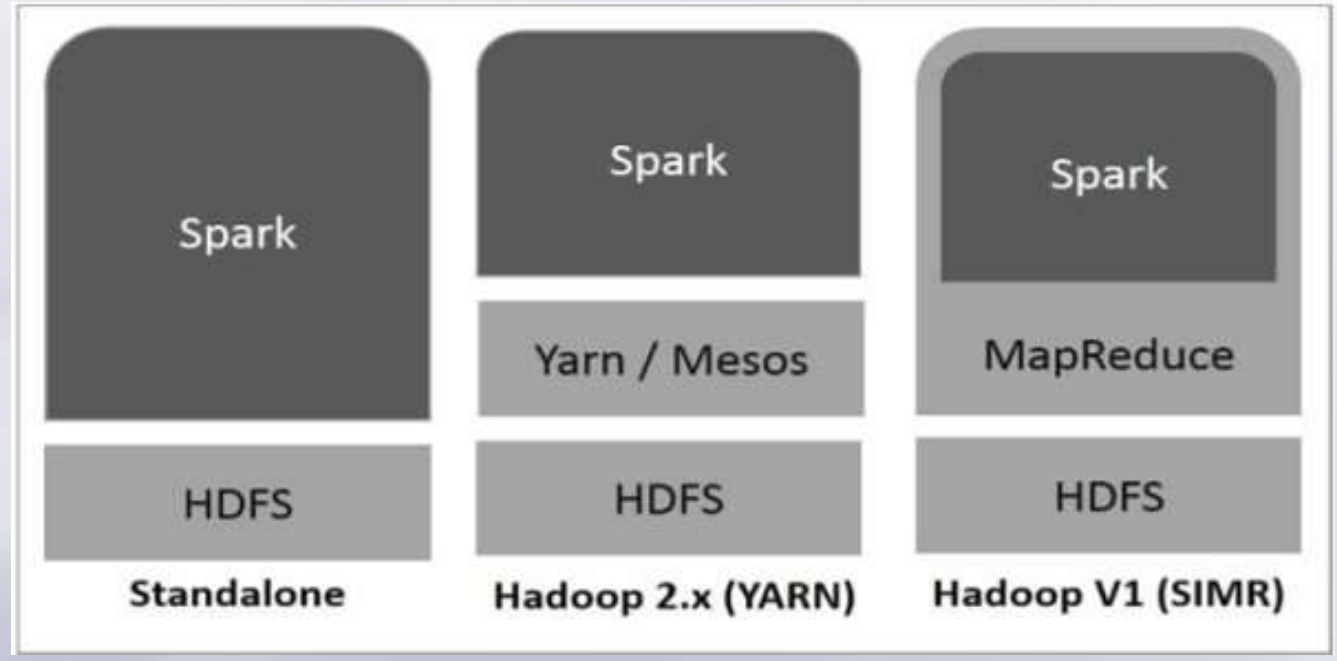
Spark en quelques mots

Evolutions depuis Hadoop

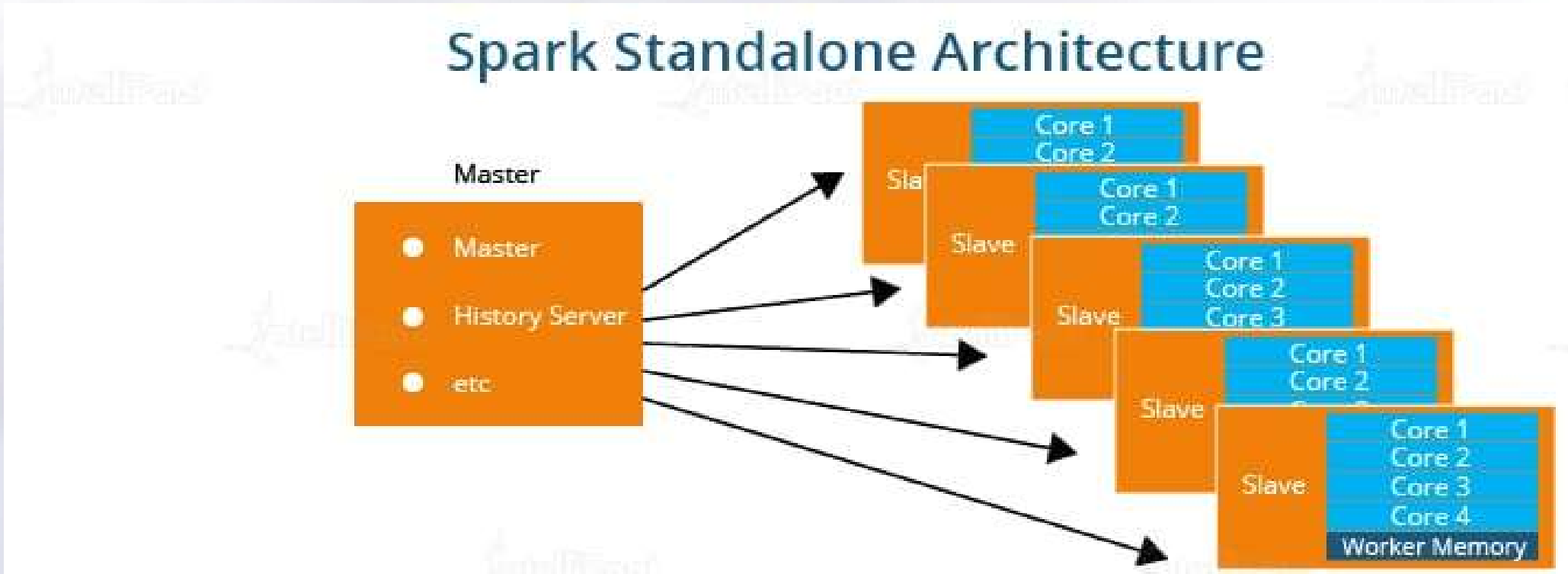
- Vitesse : réduire les opérations de lecture/écriture
  - Jusqu'à **10 fois plus rapide** lorsqu'il fonctionne sur disque
  - Jusqu'à **100 fois plus rapide** en mémoire
- Nouveau modèle de programmation
  - Multi-langues : Java, Scala ou Python
- Analyse avancée : pas seulement Map-Reduce
  - SQL
  - Données continues
  - Machine Learning
  - Algorithmes des graphes



Déploiements de Spark



Spark Standalone



Motivations

- **Le partage (réutilisation) des données est lent dans Hadoop**
- **Entre plusieurs calculs**
  - Ecrire et lire dans le HDFS
  - Impliquent de nombreuses sérialisations et IO
- **Généralement 2 schémas**
  - Itératif
  - Parallèle

Schéma itératif

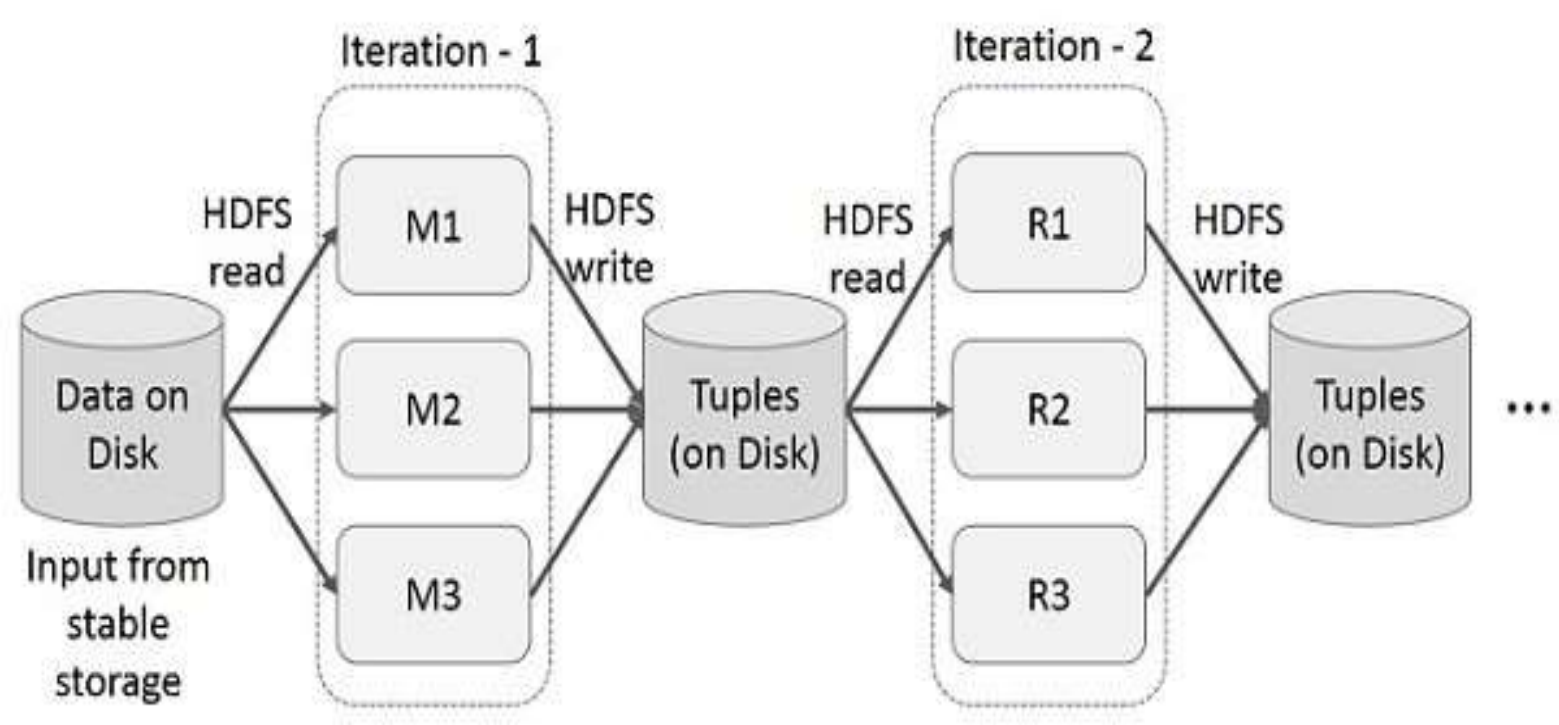


Schéma parallèle

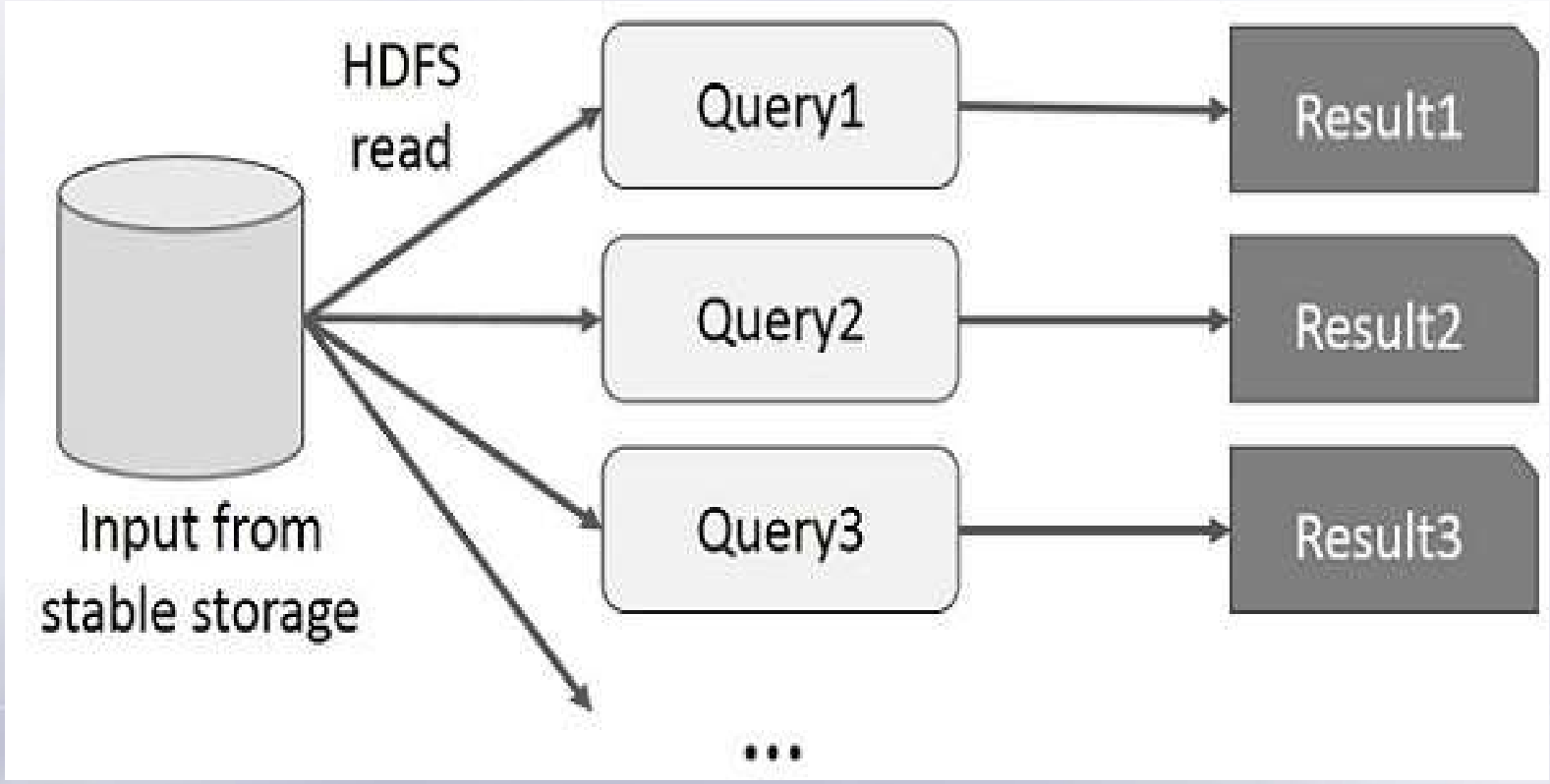
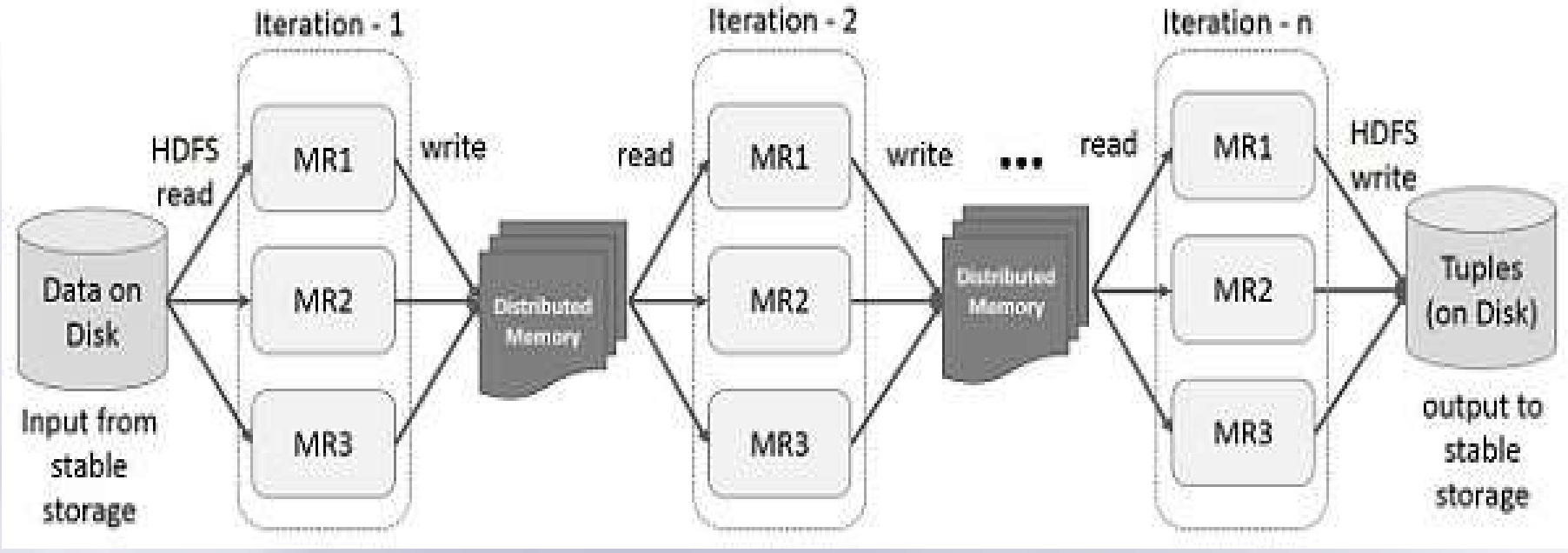
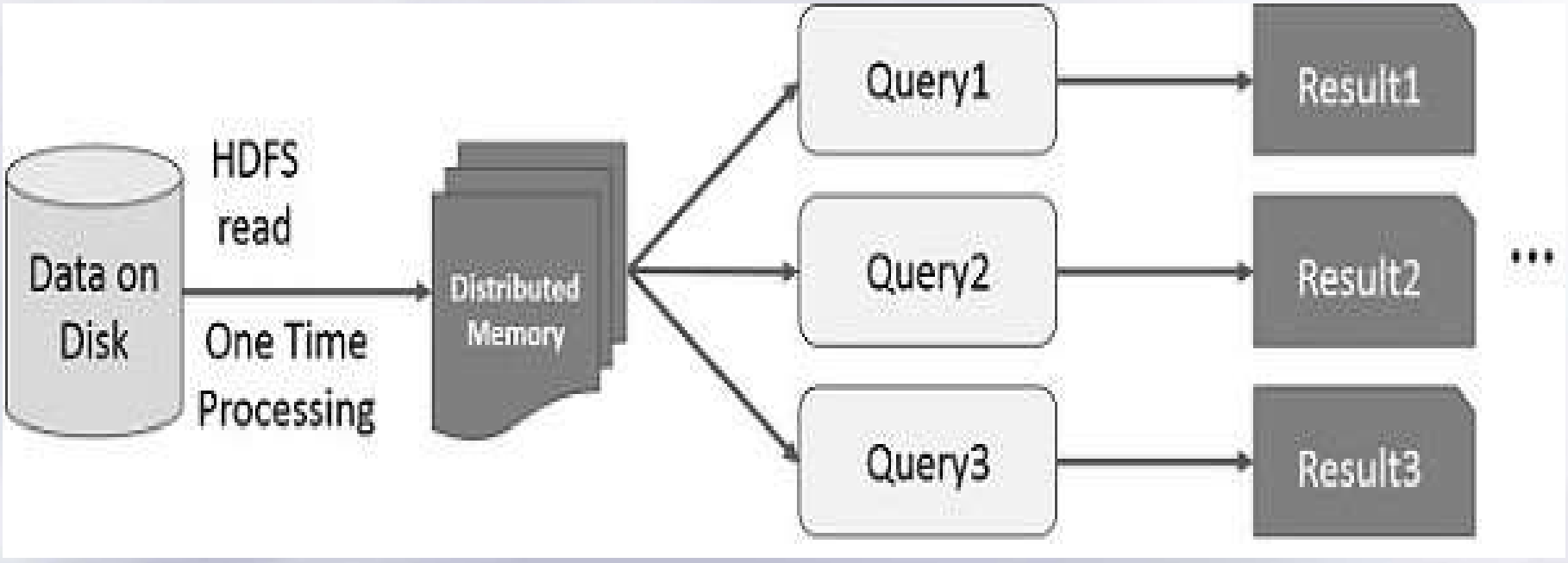


Schéma itératif avec Spark



- Conserver les données en mémoire aussi longtemps que possible
- Ne stocker sur disque que si la mémoire n'est pas suffisante
- Il n'est pas non plus nécessaire de redémarrer les JVM

Programme parallèle avec Spark



Programmer avec Spark

➢ Initialisation

```
SparkConf conf = new SparkConf().setAppName("WordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
```

➢ Spark s'appuie sur des ensembles de données distribuées résilientes appelées RDD

- Datasets sont partitionnés sur des nœuds
- Peut être exploité en parallèle

Programmer avec Spark

➢ RDD créé à partir d'un objet Java

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> rdd = sc.parallelize(data);
```

➢ RDD créé à partir d'un stockage externe (fichier)

```
JavaRDD<String> rdd = sc.textFile("data.txt");
```

Programmer avec Spark

- **Driver program : le programme principal**
- **Deux types d'opérations sur les RDD**
  - Transformations : créer un nouveau RDD à partir d'un RDD existant
    - par exemple map() passe chaque élément RDD dans une fonction donnée
  - Actions : calculer une valeur à partir d'un RDD existant
    - Par exemple, reduce() regroupe tous les éléments RDD en utilisant une fonction donnée et calcule une valeur unique
- **Les transformations sont calculées paresseusement lorsque cela est nécessaire pour effectuer une action (optimisation)**
- **Par défaut, les transformations sont mises en cache en mémoire, mais elles peuvent être recalculées si elles ne tiennent pas dans la mémoire**

Programmer avec Spark

➢ Exemple avec les expressions lambda

- map() : applique une fonction à chaque élément d'un RDD
- reduce() : applique une fonction pour agréger toutes les valeurs d'un RDD
  - **La fonction doit être associative et commutative pour le parallélisme**

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

➢ Ou avec des fonctions Java

```
class lenFunc implements Function<String, Integer> {
        public Integer call(String s) { return s.length(); }
}
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new lenFunc());
…
```

Programmer avec Spark

- **L'exécution des opérations (transformations/actions) est répartie**
  - Les variables du programme driver sont sérialisées et copiées sur les hôtes distants (il ne s'agit pas de variables globales)

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);
println("Counter value: " + counter);
```

- **Doit utiliser des variables spéciales d'accumulation et de broadcast**

Programmer avec Spark

- **De nombreuses opérations reposent sur des paires de valeurs clés**
  - Exemple (compter les lignes)
    - mapToPairs() : chaque élément de la RDD produit une paire
    - reduceByKey() : applique une fonction aux valeurs agrégées pour chaque clé

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair(s -> new Tuple2<String, Integer>(s, 1));
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b);
```

Exemple : WordCount

```
JavaRDD<String> lines = sc.textFile(inputFile);

JavaRDD <String> words = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator());

JavaPairRDD<String, Integer> pairs =
        words.mapToPair(w -> new Tuple2<String, Integer>(w,1));

JavaPairRDD <String, Integer> counts = pairs.reduceByKey((c1,c2) -> c1 + c2);
```

API Spark : Transformations

map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(func)	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	Sample a fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.

Exemple : mapPartitions

```
class Parse implements Function2<Integer, Iterator<String>, Iterator<String>> {
        public Iterator<String> call(Integer id, Iterator<String> it) {
            List<String> list = new ArrayList<String>();
            String s = id+" : ";
            while (it.hasNext()) s+= "["+it.next()+"]";  l
            ist.add(s);
            return list.iterator();
        }
}


SparkConf conf = new SparkConf().setAppName("Mappartition");
JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> data = sc.textFile(inputFile).flatMap(s -> Arrays.asList(s.split(" ")).iterator());

JavaRDD<String> partitions = data.mapPartitionsWithIndex(new Parse(), true);

partitions.saveAsTextFile(outputFile);
```

API Spark : Transformations

groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(zeroValue) (seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

API Spark : Transformations

join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(partitioner)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

API Spark : Actions

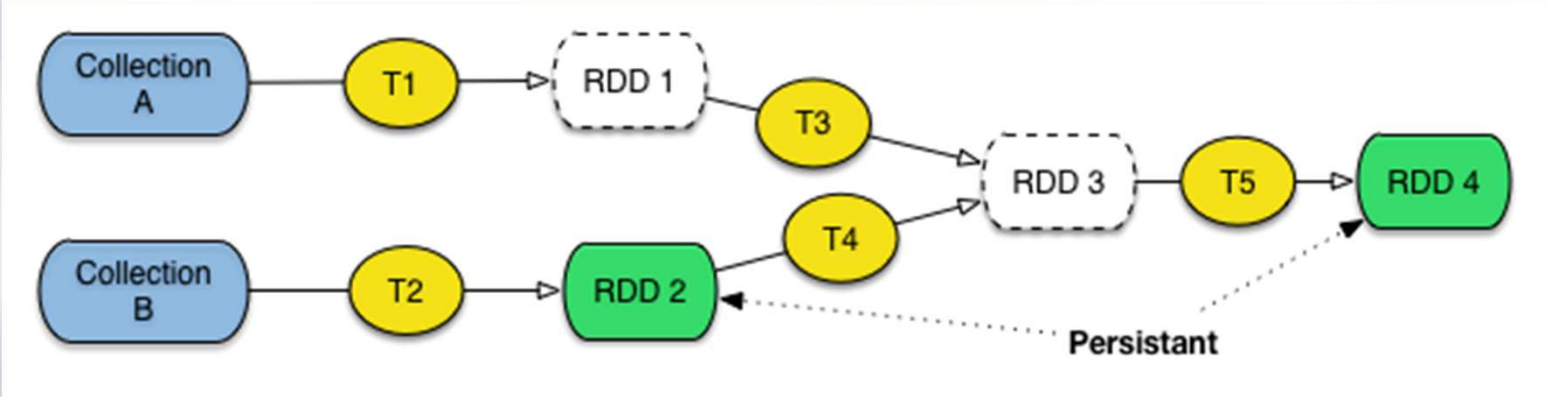
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

API Spark : Actions

saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

La chaîne de traitement Spark

- Immutabilité des collections
- RDD persistants et transitoires dans Spark



Persistance RDD

- **Les RDD sont conservés en mémoire mais peuvent être recalculés**
- **Spark gère la mise en cache en mémoire (LRU)**
- **Spark permet de contrôler la gestion de la mémoire**
  - par exemple, persist(StorageLevel.DISK_ONLY())

MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Variables Broadcast

Broadcasts

- Envoyé aux nœuds une fois
- Évitez les copies multiples si plusieurs actions sont envoyées au même nœud
- Ne doit pas être modifié

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});

broadcastVar.value();
// returns [1, 2, 3]
```

Variables Accumulator

Accumulators

- Variables mutables
- Doit être utilisé avec des opérations parallèles
- Types numériques (d'autres peuvent être mis en œuvre)

```
LongAccumulator accum = sc.sc().longAccumulator("counter");

sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x));
// ...
// 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

accum.value();
// returns 10
```

Installer Spark

Installation de Spark

- tar spark-x.y.z-bin-hadoopX.Y.tgz
- Definition des variables d'environment
  - export SPARK_HOME=<path>/spark-x.y.z-bin-hadoopX.Y
  - export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin

Développement

- **Avec eclipse**
  - Créer un Java Project
  - Ajouter les jars dans le build path
    - $SPARK_HOME/jars/spark-core_2.12-3.0.0-preview2.jar
    - $SPARK_HOME/jars/scala-library-2.12.10.jar
    - $SPARK_HOME/jars/hadoop-common-2.7.4.jar
    - Pourrait inclure tous les jars, mais pas très propre
  - Votre application est packagée dans un jar

Run

Lancer l'application

- spark-submit --class <classname> --master <url-master> target/<jarfile>
  - Centralisé : <url-master> = local or local[n]
  - Cluster : <url-master> = url to access the cluster's master

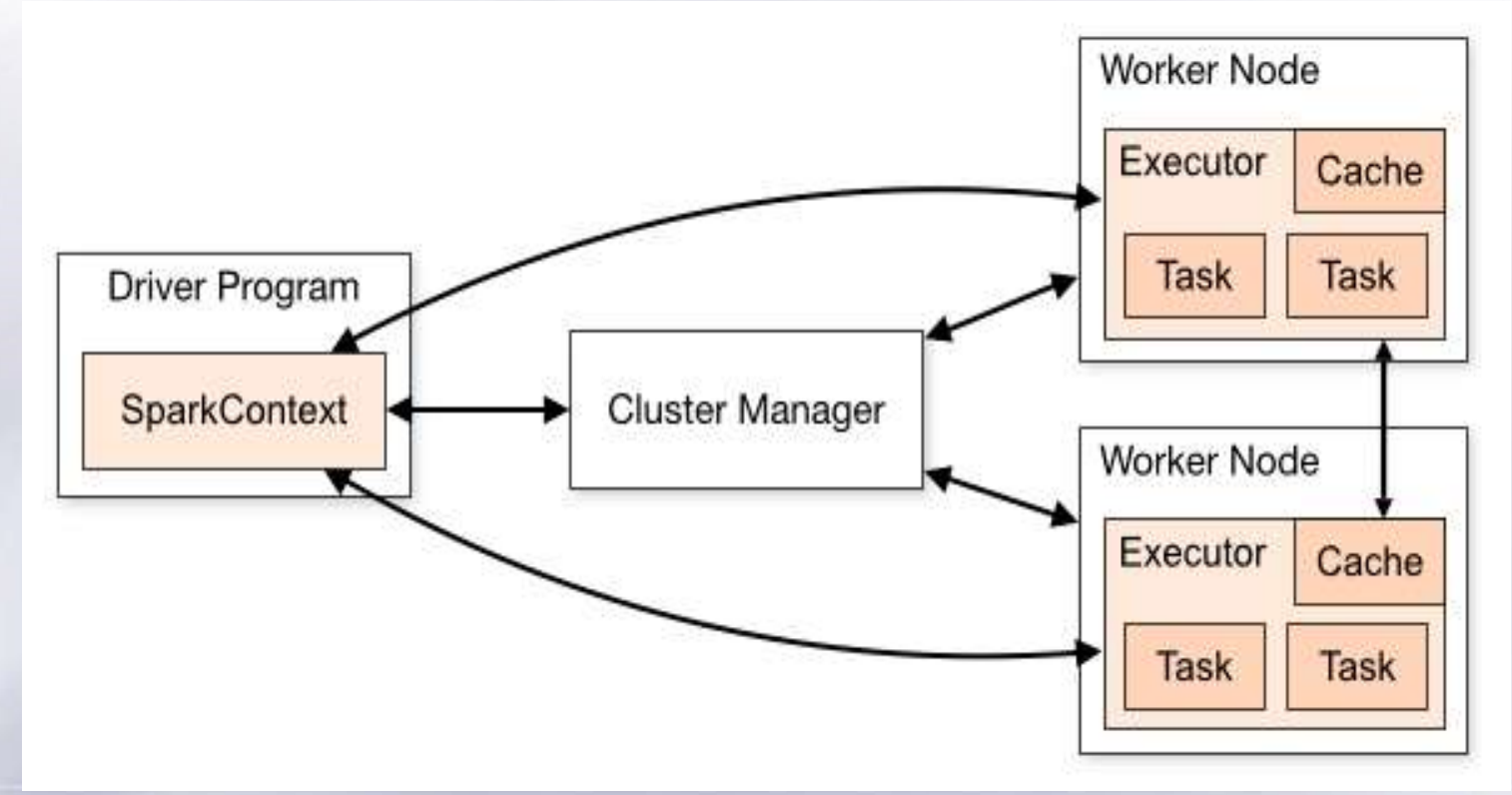
Sans HDFS

- Utilise les noms des fichiers
- Si distribué, il faut répliquer nos fichiers sur tous les nœuds

Avec HDFS

- Utiliser l'Url HDFS : hdfs://namenode:54310/input...

Mode Cluster



Mode Cluster

Démarrage du master

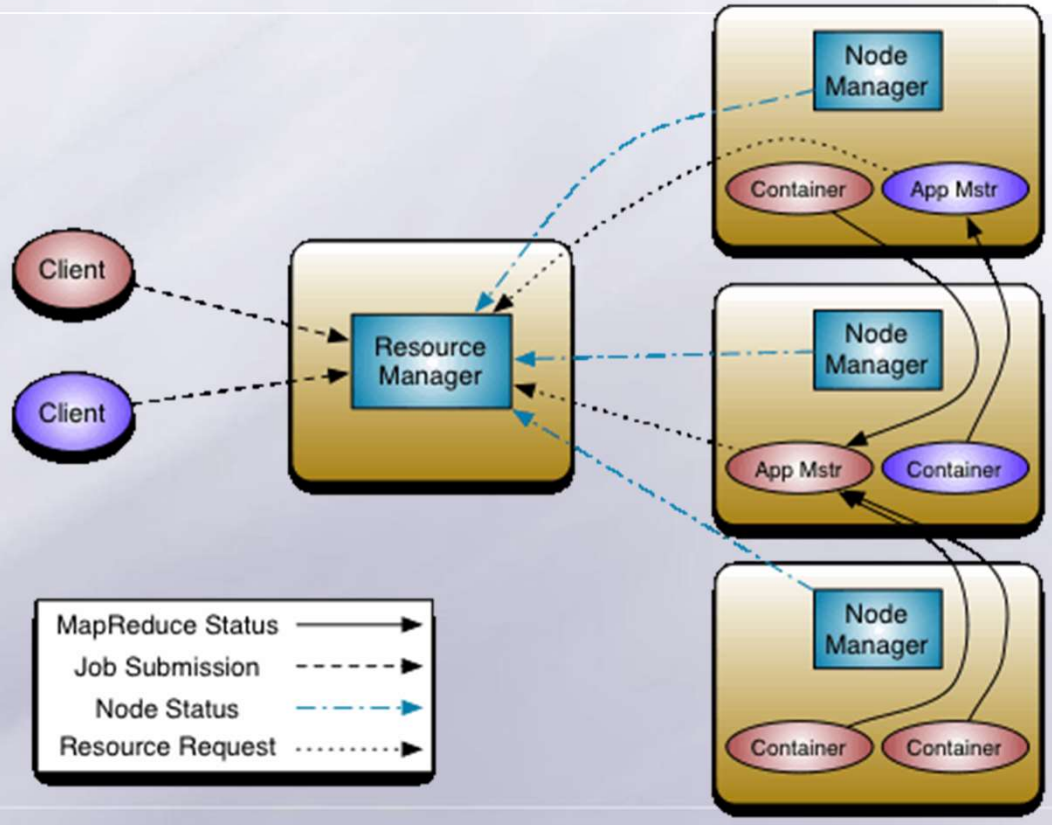
- start-master.sh
- Vous pouvez vérifier son état et voir son URL à l'adresse http://master:8080

Démarrage des slaves

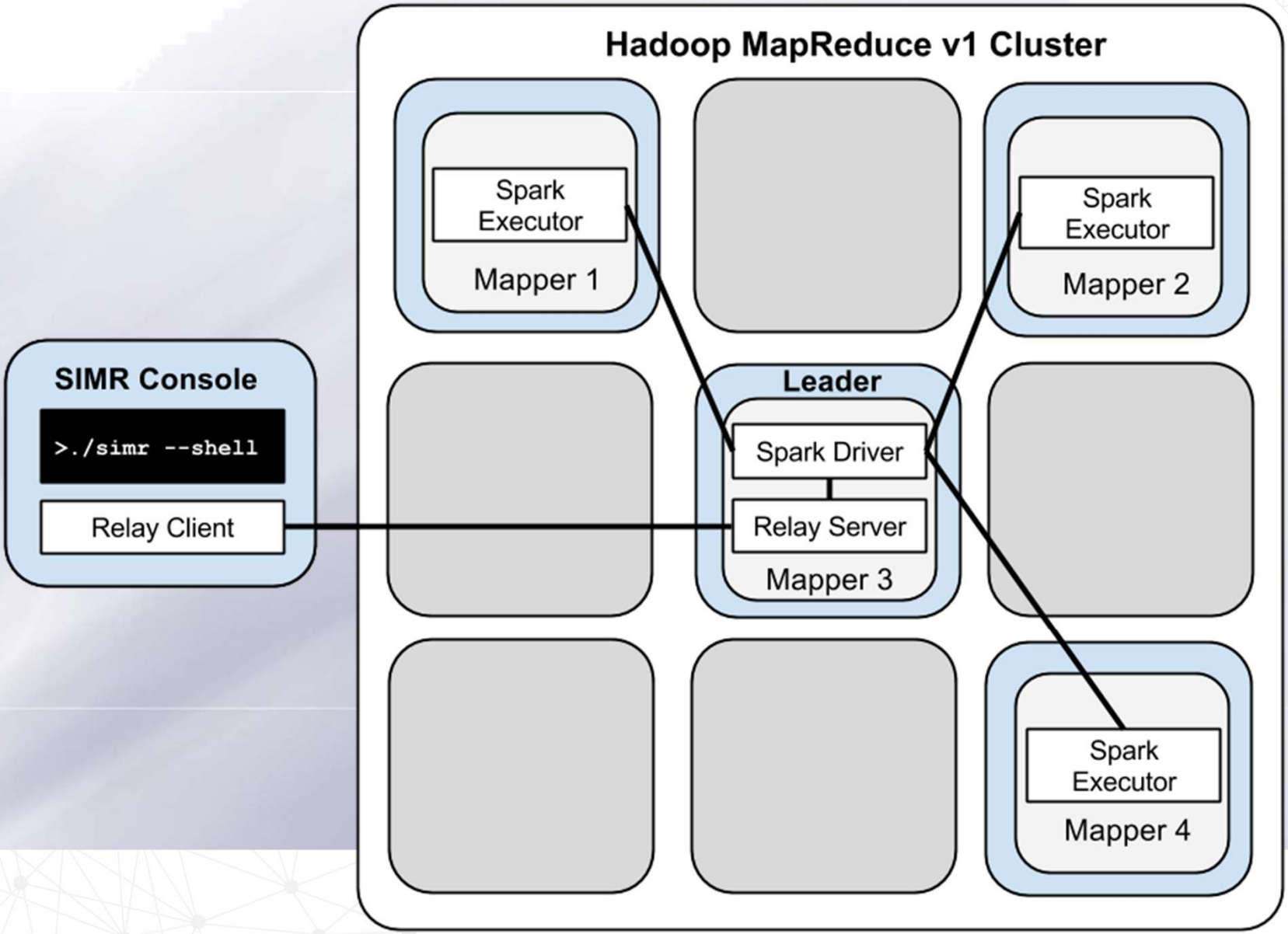
- Sur le master :
  - start-slave.sh
- Sur un esclave :
  - start-slave.sh -c 1 <url master>
  - // -c 1 d'utiliser un seul coeur

Merci de votre attention !

Spark YARN


Client
Client
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

Spark SIMR


Hadoop MapReduce v1 Cluster
Spark Executor
Mapper 1
Spark Executor
Mapper 2
SIMR Console
>./simr --shell
Relay Client
Leader
Spark Driver
Relay Server
Mapper 3
Spark Executor
Mapper 4

Exercice

- **Un ensemble de magasins**
- **Un journal (fichier) des achats (transactions)**
  - storeid,productid,number,totalprice
    - storeid : l'identifiant du magasin
    - productid : l'identifiant du produit
    - number : le nombre de produits vendus dans la transaction en cours
    - price : le prix total de la transaction (un produit peut avoir des prix différents da différents magasins)
- **Le gestionnaire veut connaître le prix moyen de chaque produit (globalement, c'est-à-dire indépendamment des magasins et des transactions)**

Exercise

- Soient deux fonctions

```
class GetProductPrice implements PairFunction<String, String, Integer> {
        public Tuple2<String, Integer> call(String s) {
                ...
                return new Tuple2<String, Integer>(productid,price);
        }
}
class GetProductNumber implements PairFunction<String, String, Integer> {  public
        Tuple2<String, Integer> call(String s) {
                …
                return new Tuple2<String, Integer>(productid,number);
}
```

- On calcule les totaux séparément

```
JavaRDD<String> input = sc.textFile(inputFile);
JavaPairRDD<String, Integer> prices = input.mapToPair(new GetProductPrice());
JavaPairRDD<String, Integer> totalprices = prices.reduceByKey((t1,t2) -> t1+t2);
JavaPairRDD<String, Integer> numbers = input.mapToPair(new GetProductNumber());
JavaPairRDD<String, Integer> totalnumbers = numbers.reduceByKey((t1,t2) -> t1+t2);

JavaPairRDD<String, Tuple2<Integer,Integer>> all = totalprices.join(totalnumbers);

JavaPairRDD<String, Integer> result = all.mapValues(t -> t._1/t._2);
```

Exercice (autre solution)

- **Soit la fonction**

```
class GetProductPriceNumber implements PairFunction<String, String, Tuple2<Integer,Integer>> {
        public Tuple2<String, Tuple2<Integer,Integer>> call(String s) {
        ...
        return new Tuple2<String, Tuple2<Integer,Integer>>(productid,
                                new Tuple2<Integer,Integer>(number,price));
```

- **Avec un seul appel à reduceByKey**

```
JavaPairRDD<String, Tuple2<Integer,Integer>> data =
        sc.textFile(inputFile).mapToPair(new GetProductPriceNumber());

JavaPairRDD<String, Tuple2<Integer,Integer>> all =
        data.reduceByKey((t1,t2) -> new Tuple2<Integer,Integer>(t1._1+t2._1,t1._2+t2._2));

JavaPairRDD<String, Integer> result = all.mapValues(t -> t._1/t._2);
```

Exercice

- Le gérant veut connaître pour chaque magasin le nombre de transactions où le prix du produit était inférieur au prix moyen

- Soit la fonction

```
class GetProductStorePrice implements PairFunction<String, String, Tuple2<String, Integer>> {
        public Tuple2<String, Tuple2<String, Integer>> call(String s) {
                …
        return new Tuple2<String, Tuple2<String, Integer>>(productid,
                                new Tuple2<String, Integer>(storeid,priceperitem));
```

```
// from previous question
JavaPairRDD<String, Integer> avg = all.mapValues(t -> t._1/t._2);

JavaPairRDD<String, Tuple2<String, Integer>> products =
sc.textFile(inputFile).mapToPair(new GetProductStorePrice());  JavaPairRDD<String,
Tuple2<Tuple2<String, Integer>, Integer>> productswithavg =
                                        products.join(avg);
JavaPairRDD<String, Tuple2<Tuple2<String, Integer>, Integer>> selectedproducts =
                        productswithavg.filter(t -> t._2._1._2 < t._2._2);
JavaPairRDD<String, Integer> storesells =
                                        selectedproducts.values().mapToPair(t -
                > t._1);  Map<String, Long> result = storesells.countByKey();
```