



**Institut Supérieur d'Informatique et Mathématiques
Monastir**

Big Data et NoSQL

Chapitre 3: Traitement des données massives avec Spark

Mastère de Recherche en Génie Logiciel – Niveau 2

MAJ: 06/03/24

Asma KERKENI asma.kerkeni@gmail.com

Objectifs

2

- 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.
 - ▣ Maîtriser les RDDs.
 - ▣ Ecrire des programmes en utilisant l'API PySpark

Plan

3

- ❑ Modes de traitement des données
- ❑ De Hadoop Vers Spark
- ❑ RDD Spark
- ❑ Opérations Spark
- ❑ Applications
- ❑ Architecture de Spark

Références du cours

4

□ 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

5

Modes de traitement des données

Modes de traitement de données

6

□ Deux modes de traitement de données:

- ▣ traitement par lot (Batch)
- ▣ traitement de flux (Streaming)

Batch



Streaming

Modes de traitement de données

8

□ 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.

Modes de traitement de données

9

□ 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 récuratifs 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).

Modes de traitement de données

10

□ 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.

Modes de traitement de données

11

□ Traitement de flux : *Stream Processing (suite)*

■ 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.

12

De Hadoop Vers Spark

Contexte d'apparition de Spark:

Limites de Hadoop M/R

13

- ❑ **Limite 1 :** Hadoop Map-Reduce est un exemple de système adapté uniquement pour traitement par lot.
- ❑ **Limite 2 :** Nécessité d'écriture sur disque après une opération map ou reduce pour permettre aux mappers et aux reducers de communiquer.
 - 😊 Tolérance aux pannes.
 - 😞 Ecritures et lectures sont coûteuses en temps, surtout pour les algorithmes itératifs.
- ❑ **Limite 3 :** Limite du jeu d'expressions composé exclusivement d'opérations map et reduce.
 - 😞 Difficulté d'exprimer des opérations complexes en n'utilisant que cet ensemble.

Contexte d'apparition de Spark:

Besoins des applications Big Data

15

- Des besoins plus complexes (que le traitement batch) 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...):

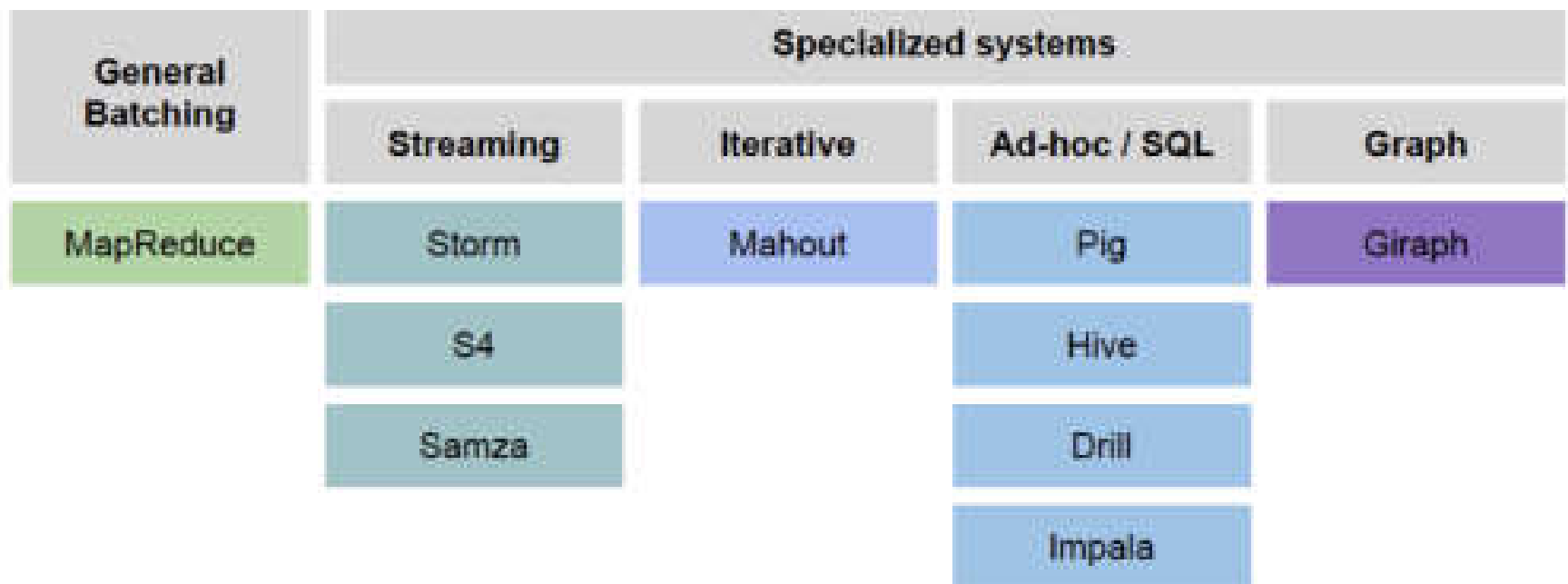
Contexte d'apparition de Spark:

Besoins des applications Big Data

16

- Apparition de plusieurs outils dans l'écosystème de Hadoop pour répondre à ces besoins.

Outils disponibles dans l'écosystème de Hadoop



□ Limites:

- Diversité des APIs,
- Vision peu unifiée,
- interactions coûteuses entre logiciels.

Contexte d'apparition de Spark:

Traitement In-Memory

17

- *In-Memory Processing*: 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.
 - ▣ Contraire du batch processing (traitement sur disque).
 - ▣ 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 d'apparition de Spark:

Traitement In-Memory

18

□ 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)



<http://www.icmit.com/mem2014.htm>

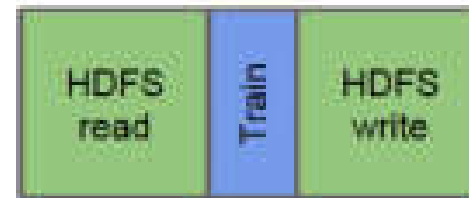
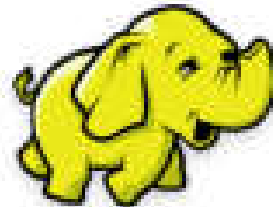
22

Spark : Introduction

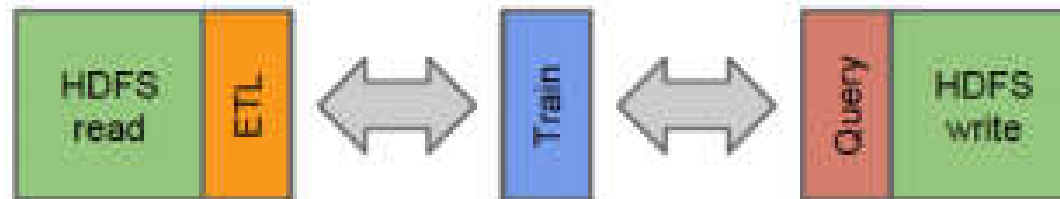
Vers un Framework unifié

23

Plusieurs
frameworks
spécialisés



Framework unifié



Spark : Un bref Historique

24

- ❑ 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: 3.3.4 (Depuis Décembre 2023).



Spark : C'est quoi?

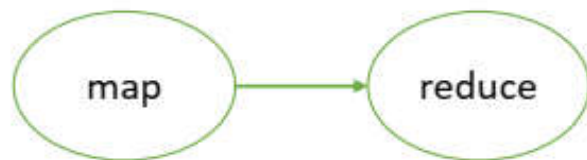
25

- 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 tolérante aux pannes.
- C'est un framework supportant **les traitements temps réel**.

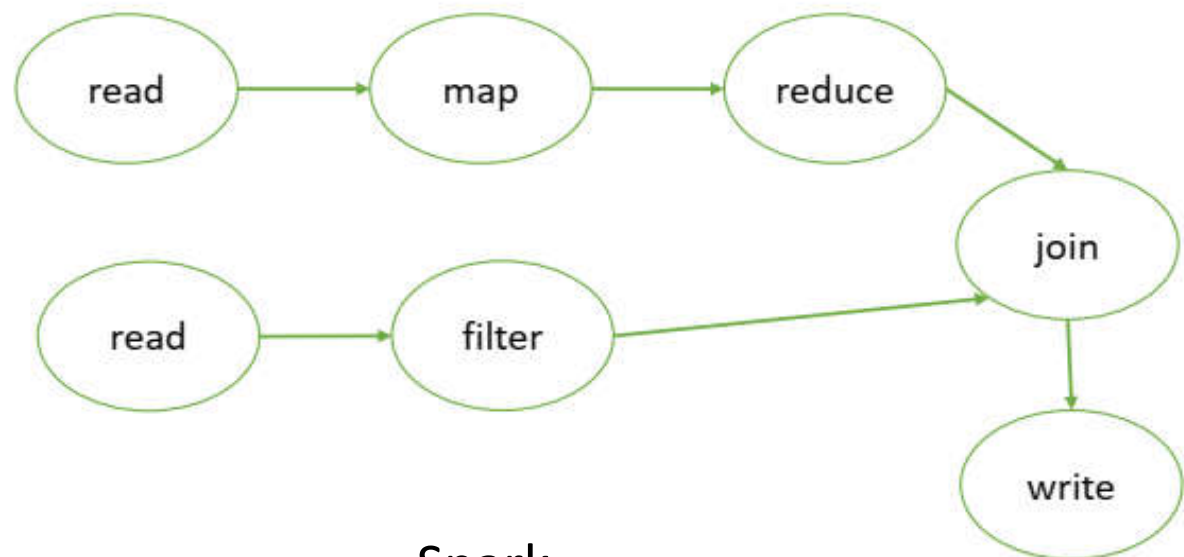
Spark : C'est quoi?

26

- Spark offre les opérations Map et Reduce mais beaucoup plus:
 - ▣ filter(), map(), flatmap(), groupByKey(), ReduceByKey(), join().....
 - ▣ collect() first(), take(), save().....



Hadoop

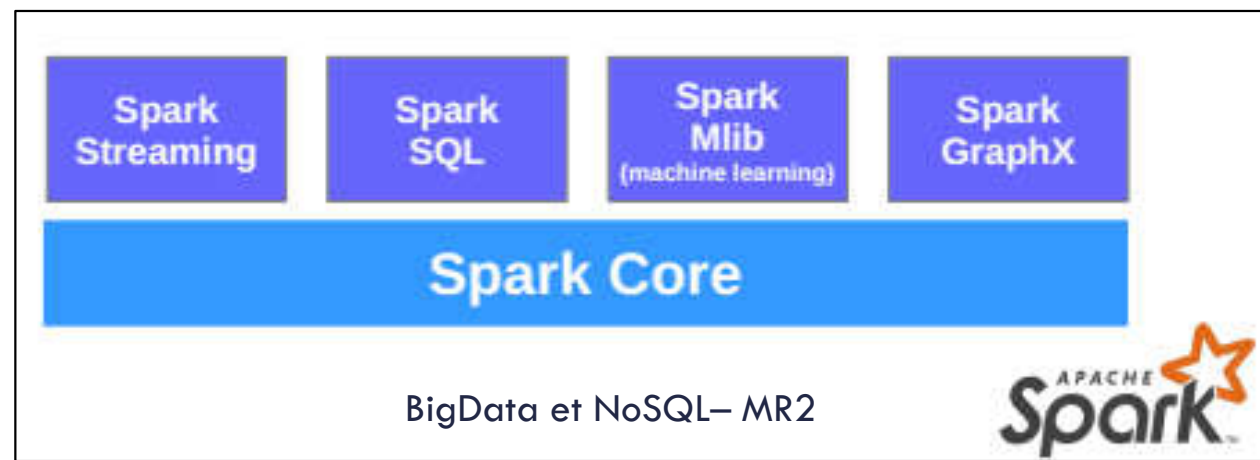


Spark

Spark: c'est quoi?

27

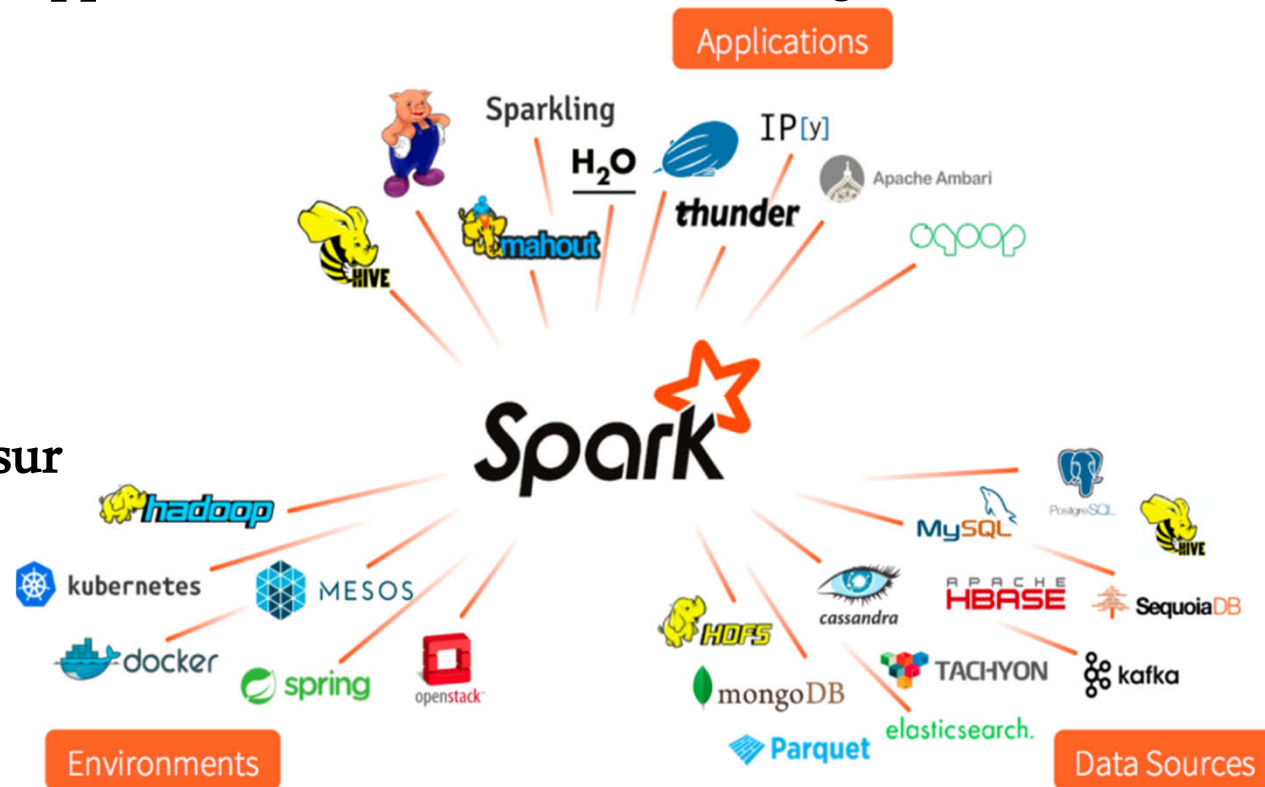
- Une plateforme unifiant plusieurs librairies:
 - ▣ **Spark Core** : librairie basique
 - ▣ **Spark Streaming** : Librairie pour flux de données temps réel
 - ▣ **Spark SQL** : Librairie pour manipuler des données structurées
 - ▣ **Spark MLib** : Librairie pour analyse et fouille de données (machine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



Spark : C'est quoi?

28

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



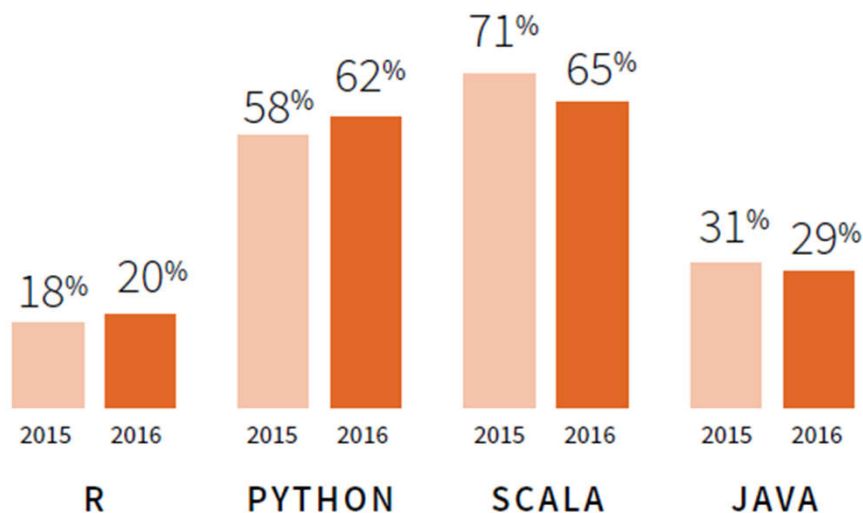
Spark peut s'exécuter sur
plusieurs plateformes:
Hadoop, Mesos, en
standalone ou sur le
cloud.

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

Spark et langages

29

- 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()
```

Standalone Programs
•Python, Scala, & Java

Scala

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

Interactive Shells
•Python & Scala

Java

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

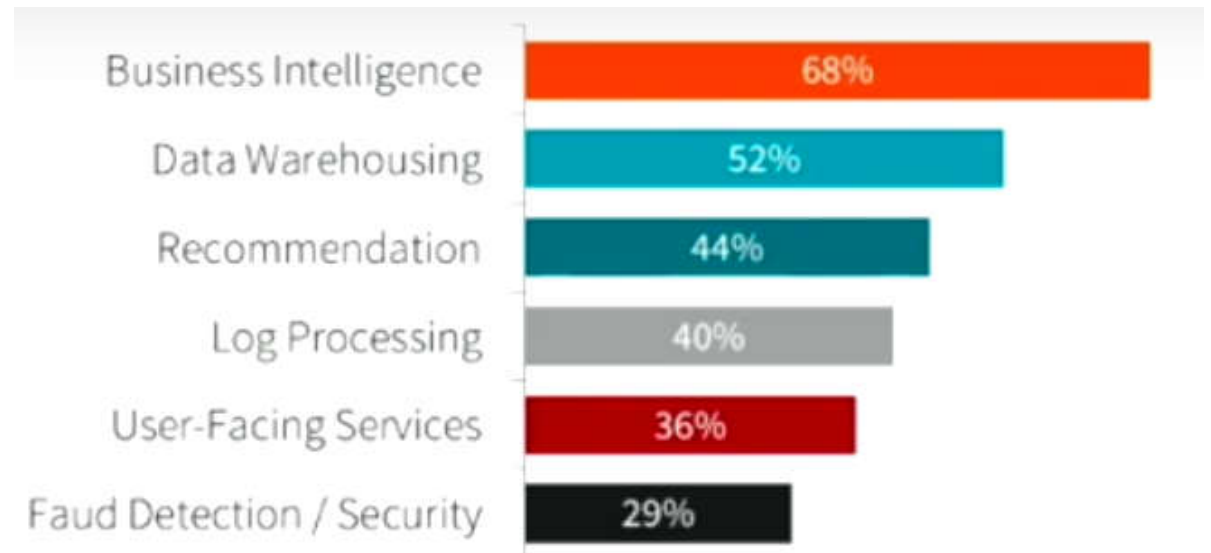
Performance
•Java & Scala are faster
due to static typing
•...but Python is often fine

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

Spark : cas d'utilisation

30

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



Spark Vs Hadoop

31

- ❑ 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

DIGITAL ET NOUVEL MRZ

Spark Vs Hadoop

32

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(_ + _)
```

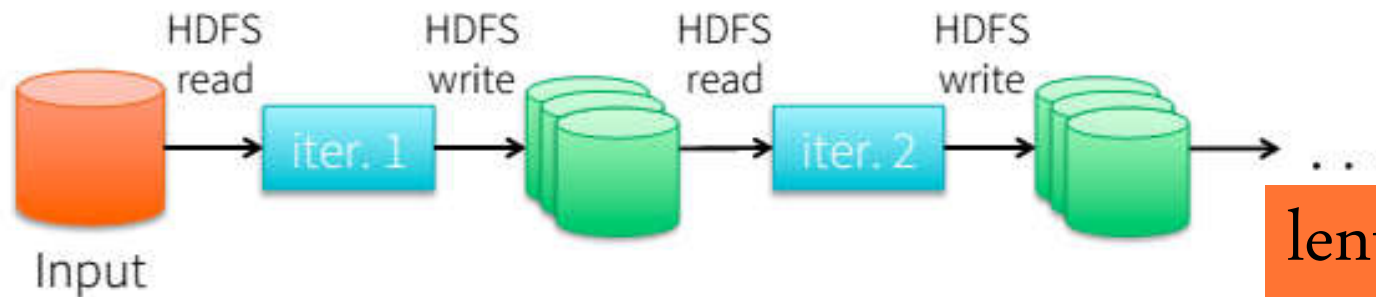
4

Spark Vs Hadoop : Différence clé

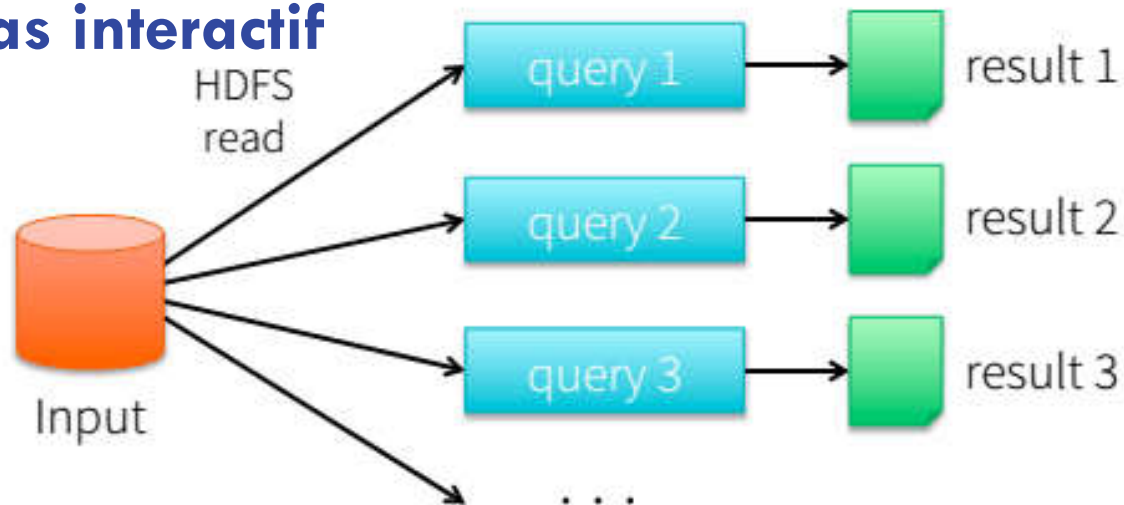
33

Modèle de Partage de données avec Map/Reduce

Cas séquentiel



Cas interactif



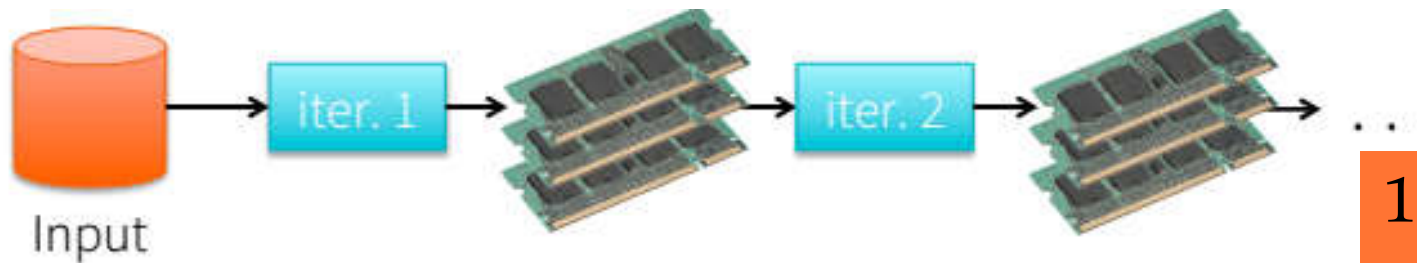
lent en raison de la
réplication, de la
sérialisation et des E/S,
mais nécessaire pour la
tolérance aux pannes.

Spark Vs Hadoop : Différence clé

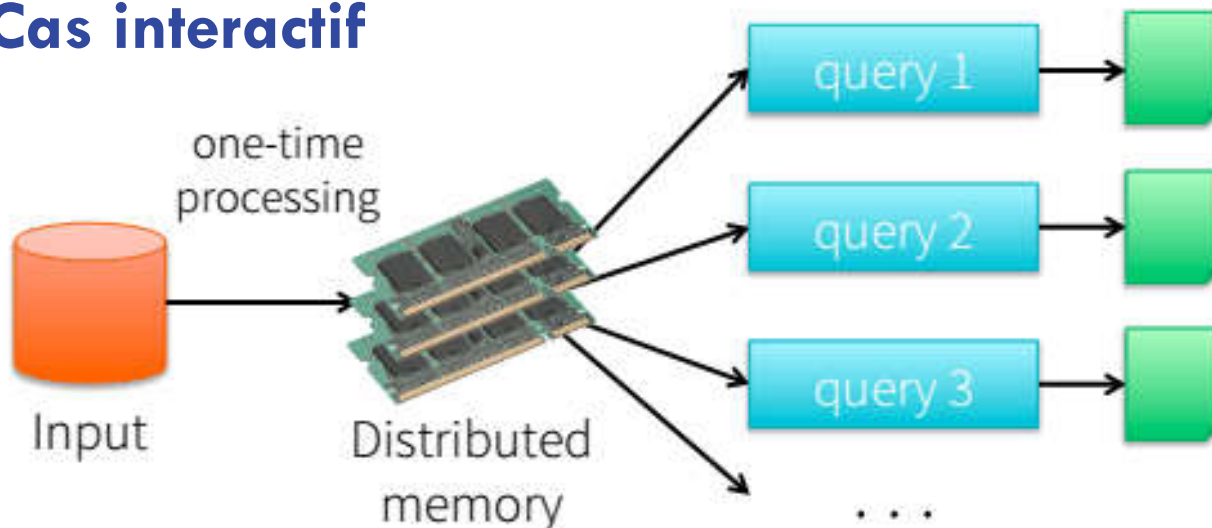
34

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**.

35

RDD Spark

RDD: *Resilient Distributed Datasets*

36



[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: Définition

37

- ❑ C'est l'abstraction basique de Spark.
- ❑ RDD: Ensemble résilient de données distribuées.
- ❑ Un RDD est une collection distribuée d'objets immuables.
- ❑ Les RDDs peuvent contenir n'importe quel type d'objets Python, Java ou Scala, y compris les classes définies par l'utilisateur.
- ❑ Un RDD est partitionné
 - Les données de chaque RDD sont divisées en plusieurs partitions.
 - Chaque partition réside sur un nœud du cluster.
 - Deux partitions peuvent résider sur le même nœud.

RDD: Exemple

38

```
rdd = sc.parallelize(range(1,25),5)
```

rdd is split into
partitions

Partitions on stored
in worker's memory

Partition 0

[1,2,3,4,5]

Partition 1

[6,7,8,9,10]

Partition 2

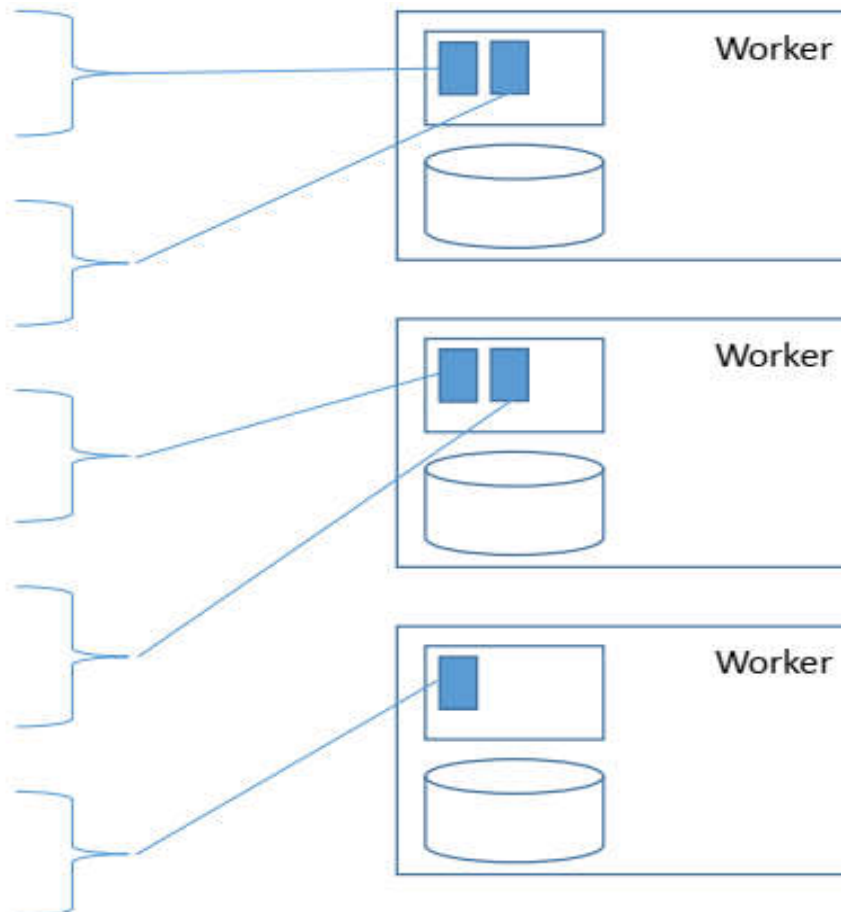
[11,12,13,14,15]

Partition 3

[16,17,18,19,20]

Partition 4

[21,22,23,24]

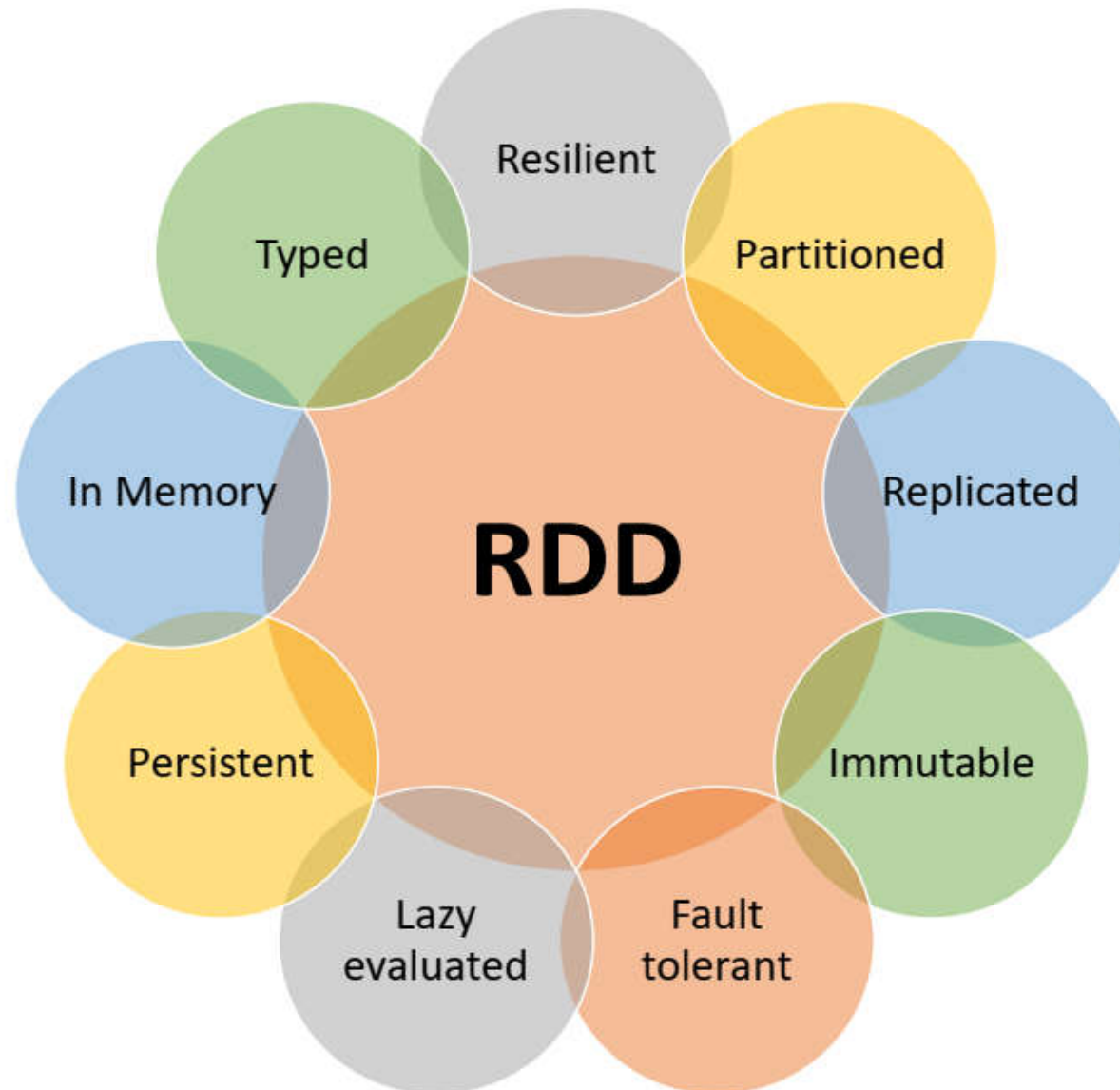


IF File in HDFS
By default
1 block = 1 partition

It is possible to
specify a different
number of partitions

RDD: Caractéristiques

39

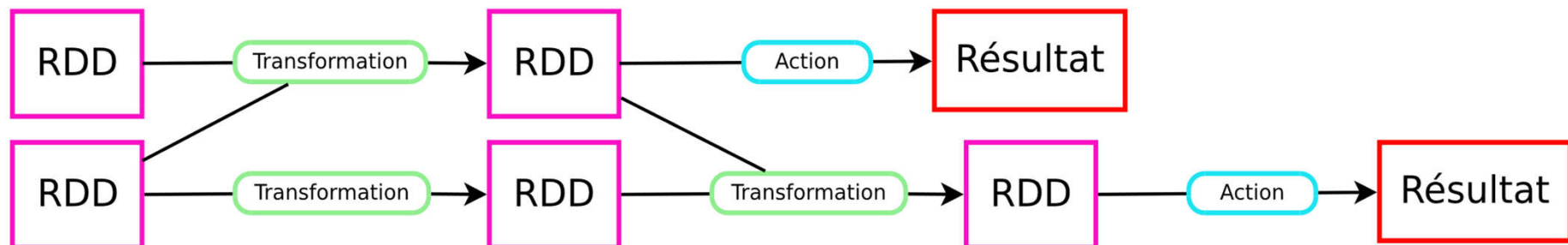


RDD: Caractéristiques

40

□ Un RDD est Résilient:

- Résilient: tolérant aux pannes, si les données en mémoire (ou sur un nœud) sont perdues, il est possible de le récupérer.
- Un RDD conserve son information de lignée, ce qui signifie qu'il peut être recréé à partir des RDD parents.
- Les RDD suivent le graphique des transformations qui les ont construits : lignée (lignage) pour reconstruire les données perdues.
- Coder dans Spark consiste à créer une lignée de RDDs sous forme de graphe acyclique orienté (DAG).

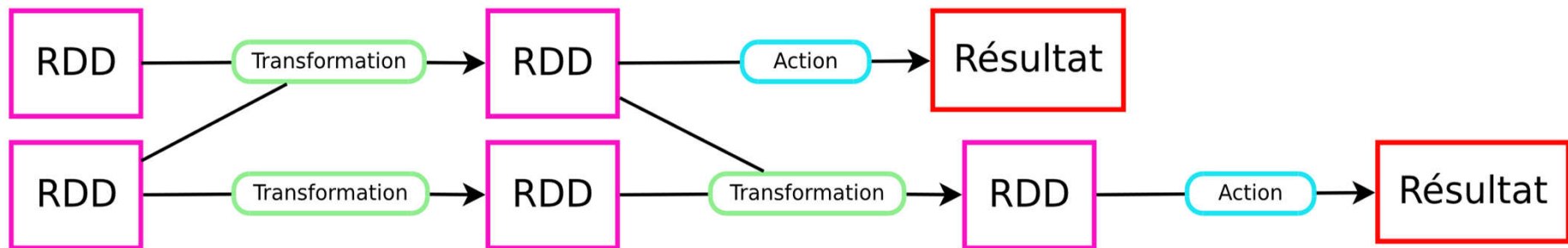


RDD: Caractéristiques

41

□ Un RDD est Résilient (suite):

- 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 régénéré à partir de ses nœuds parents: **tolérance aux pannes**.



RDD: Caractéristiques

42

□ Un RDD est:

- ▣ **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.
- ▣ **Redondant :** limite le risque de perte de données.
- ▣ **En mémoire:** Un RDD est stocké en mémoire.
- ▣ **Typé:** un RDD possède un type (Python, Java, Scala incluant RDD [int], RDD [long], RDD [string]). Il existe également des RDDs dont les éléments sont des paires (clé, valeur).

RDD: Caractéristiques

43

□ Un RDD est:

■ **Persistant:** Un RDD peut être marqué comme **persistant** pour une réutilisation future, ses partitions sont sauvegardées 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

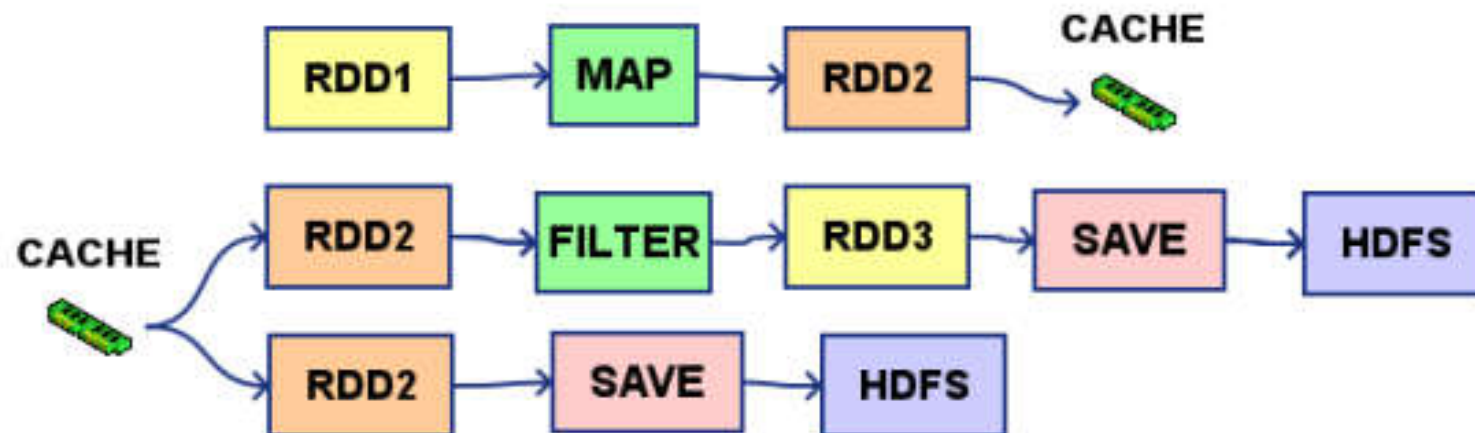
44

Sans persistance



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

Avec persistance (sur RDD 2)



RDD et persistance

45

- 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,
 - Le 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.

47

Opérations Spark

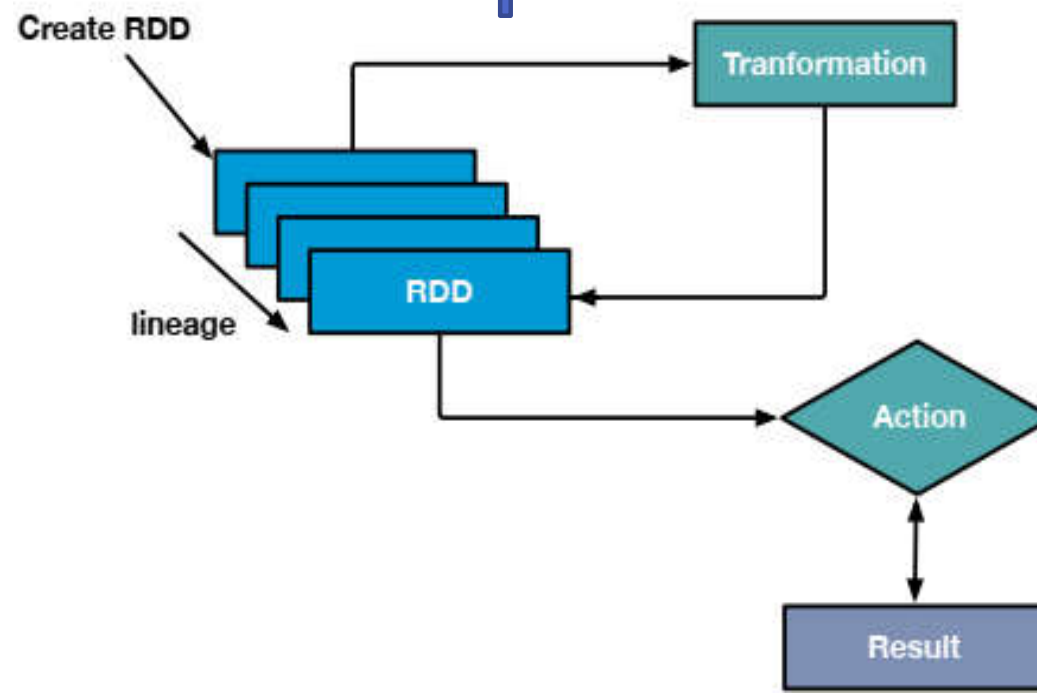
RDD: opérations

48

□ RDD supporte deux types d'opérations:

Transformation: appliquer une fonction sur 1 à n RDD et retourner une nouvelle RDD. Elles sont lazy.

Action: à appliquer une fonction et retourner une valeur au programme driver pour par exemple les afficher ou les enregistrer dans un fichier.



RDD: Opérations

49

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- groupBy
- sortBy
- flatMapValues
- groupByKey
- reduceByKey
- foldByKey
- sortByKey
- combineByKey

Math / Stats

- sample
- sampleByKey
- randomSplit

Set Theory

- union
- intersection
- subtract
- distinct
- cartesian
- zip
- join
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueId
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe
- partitionBy

ACTIONS

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

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

- takeOrdered

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

RDD: Opérations

50

□ Exemples de transformations:

- `map()` : une valeur \rightarrow une valeur
- `mapToPair()` : une valeur \rightarrow un tuple
- `filter()` : filtre les valeurs/tuples
- `groupByKey()` : regroupe les valeurs par clés
- `reduceByKey()` : agrège 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>`)

Transformations des RDDs:

Définition & Types

51

Caractéristique des transformation

- Permet de décrire une fonction de transition entre un ou plusieurs RDD parent(s) et un RDD fils.
- Étape de transition décrivant un flux de données
- Exécution paresseuse : permet des optimisations avant l'exécution



2 types de transformation

- transformations étroites
- transformations larges

Transformations des RDDs:

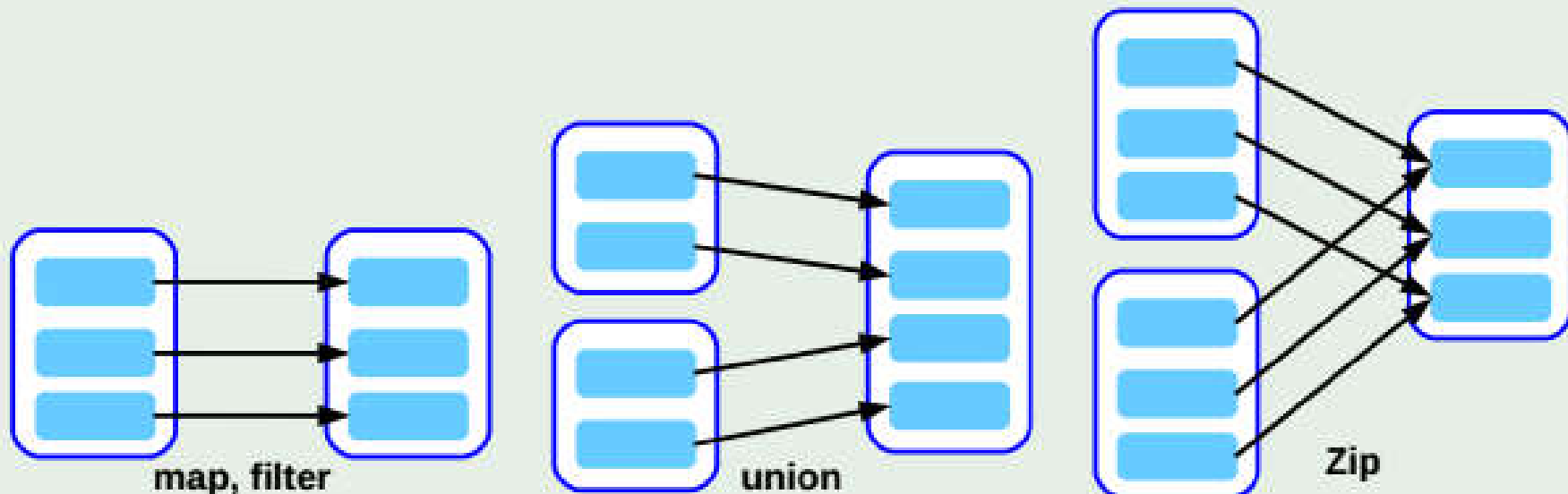
Transformations Etroites

52

Une relation 1 to 1

- Chaque partition d'un parent RDD est utilisée par au plus une partition d'un RDD fils
- pas besoin de synchronisation pour passer du RDD parent au RDD fils

Exemples :



Transformations des RDDs:

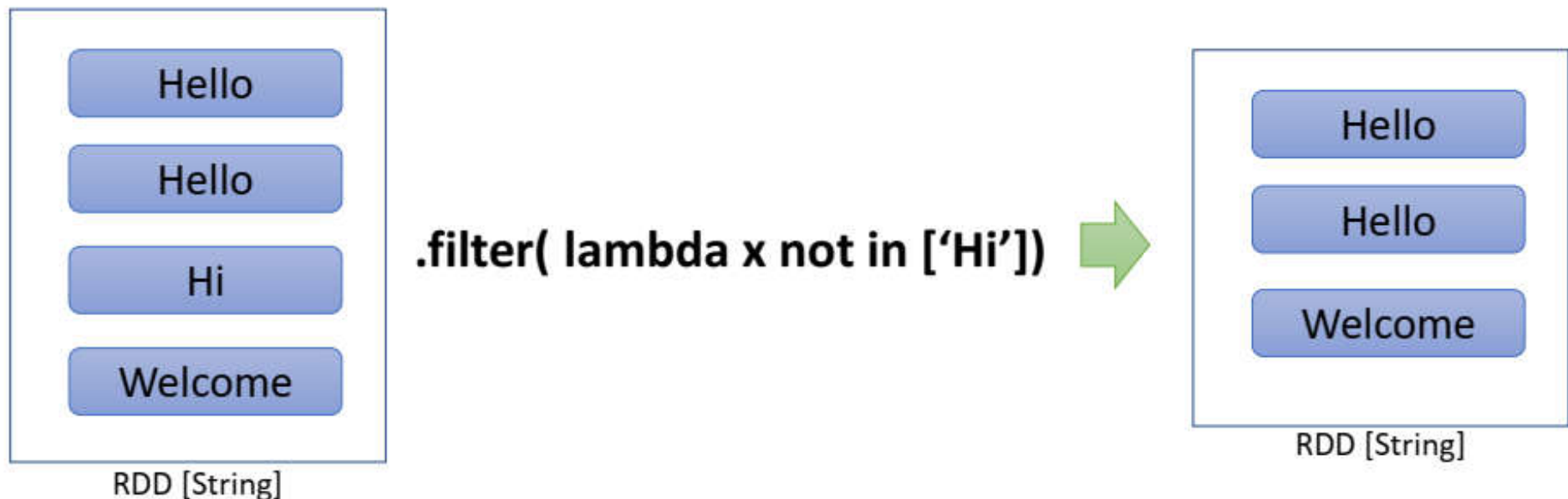
Transformations Etroites

54

□ Exemple : Transformation étroite « *filter* »

filter(*func*)

Return a new dataset formed by selecting those elements of the source on which *func* returns true.



50

Transformations des RDDs:

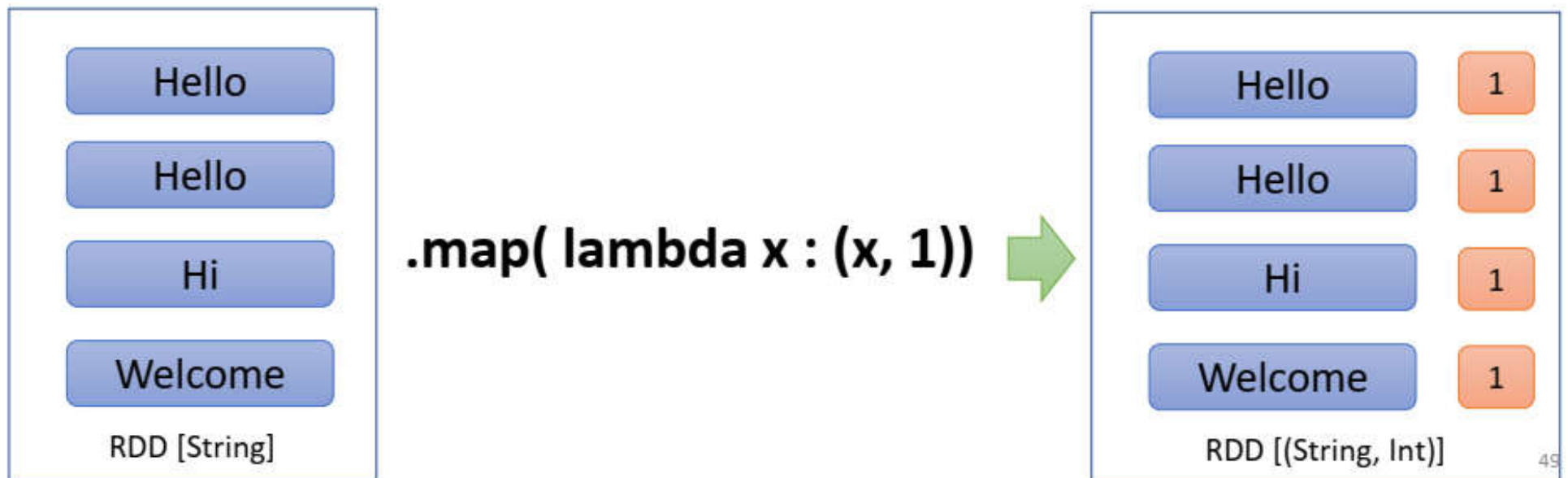
Transformations Etroites

55

□ Exemple : Transformation étroite « *map* »

map(func)

Return a new distributed dataset formed by passing each element of the source through a function *func*.



Transformations des RDDs:

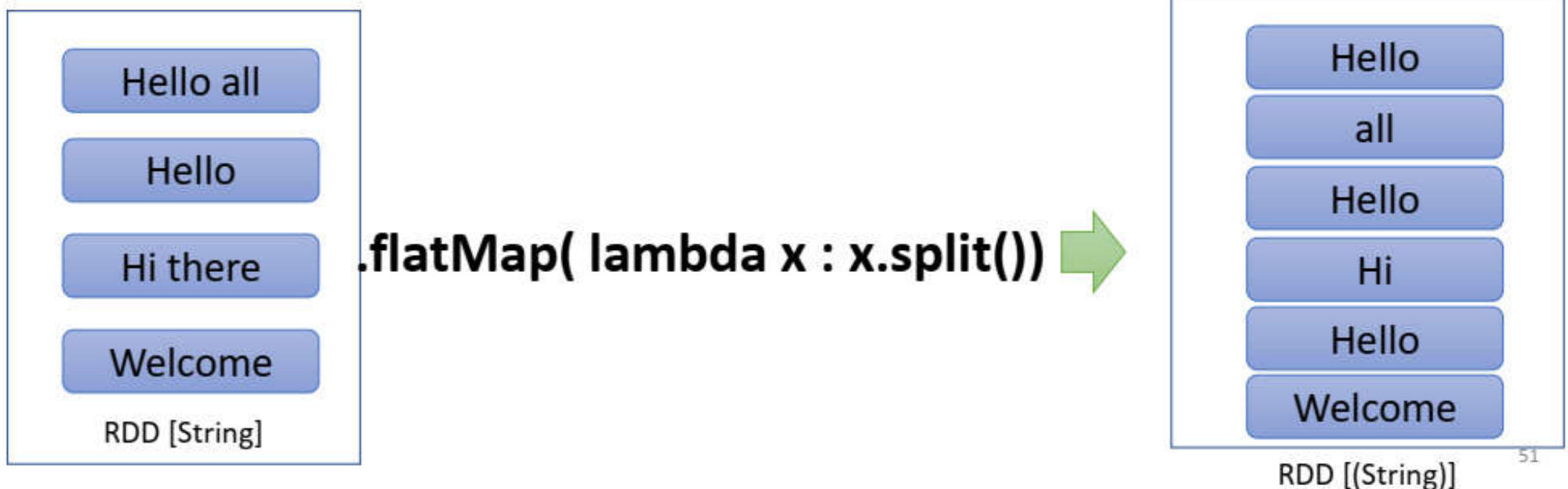
Transformations Etroites

56

□ Exemple : Transformation étroite « *flatMap* »

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).



Transformations des RDDs:

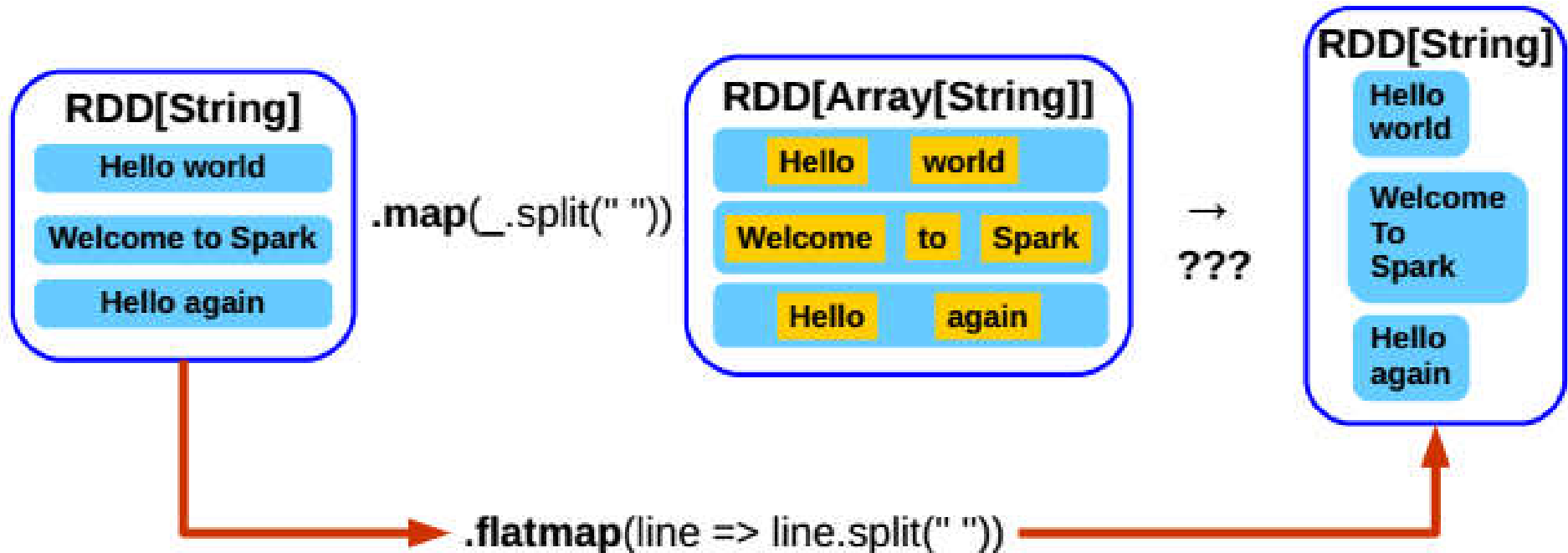
Transformations Etroites

57

□ Exemple : Transformation étroite « *flatMap* »

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).



Transformations des RDDs:

Transformations Larges

58

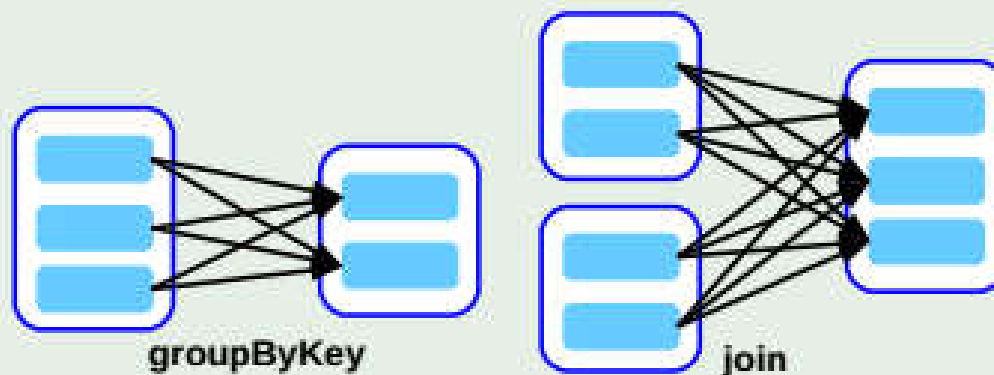
Une relation all to all (= Shuffle)

- plusieurs partitions filles peuvent dépendre d'une partition donnée
- les données de toutes les partitions parentes doivent être présentes
- implique des I/O disque et réseau , de synchronisation entre nœuds

Opérations coûteuses

mais configuration fine des paramètres permet d'améliorer les performances

Exemples :



Transformations des RDDs:

Transformations Larges

59

□ Exemple : Transformation large « *intersection* »

intersection(*otherDataset*)

Return a new RDD that contains the intersection of elements in the source dataset and the argument.



Transformations des RDDs:

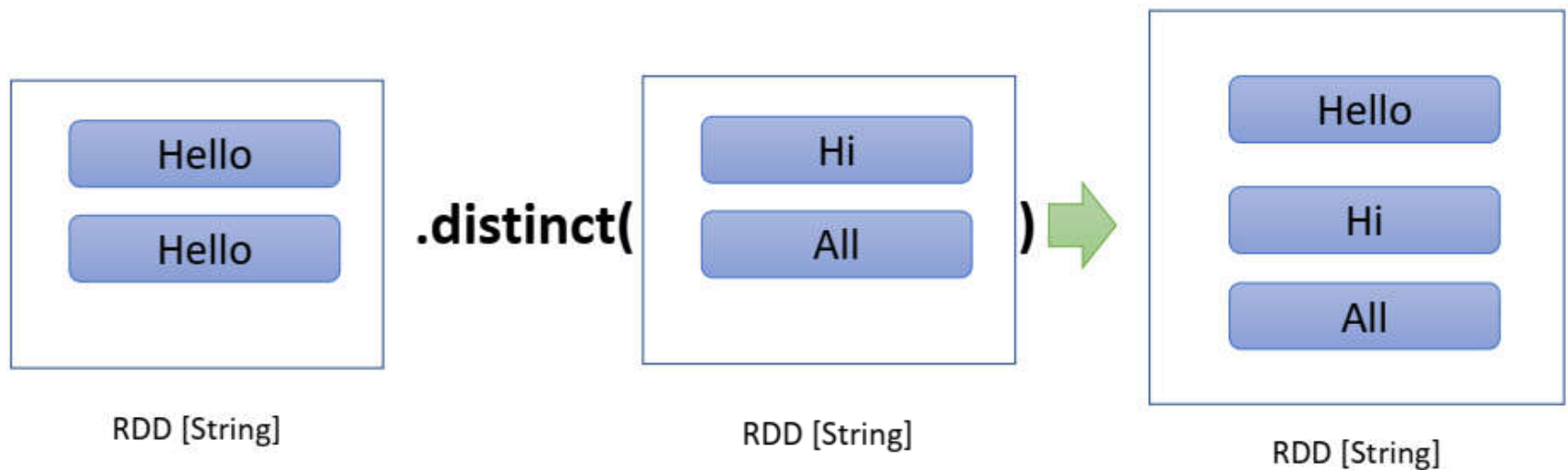
Transformations Larges

60

□ Exemple : Transformation large « *distinct* »

distinct(*numPartitions*)

Return a new dataset that contains the distinct elements of the source dataset.



55

Transformations des RDDs:

Transformations Larges

61

□ Exemple : Transformation large « *groupByKey* »

Uniquement applicable sur les RDD[(K,V)]

groupByKey([*numPartitions*])

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.



Dr. Asma Cherif Fall 2019

56

Transformations des RDDs:

Transformations Larges

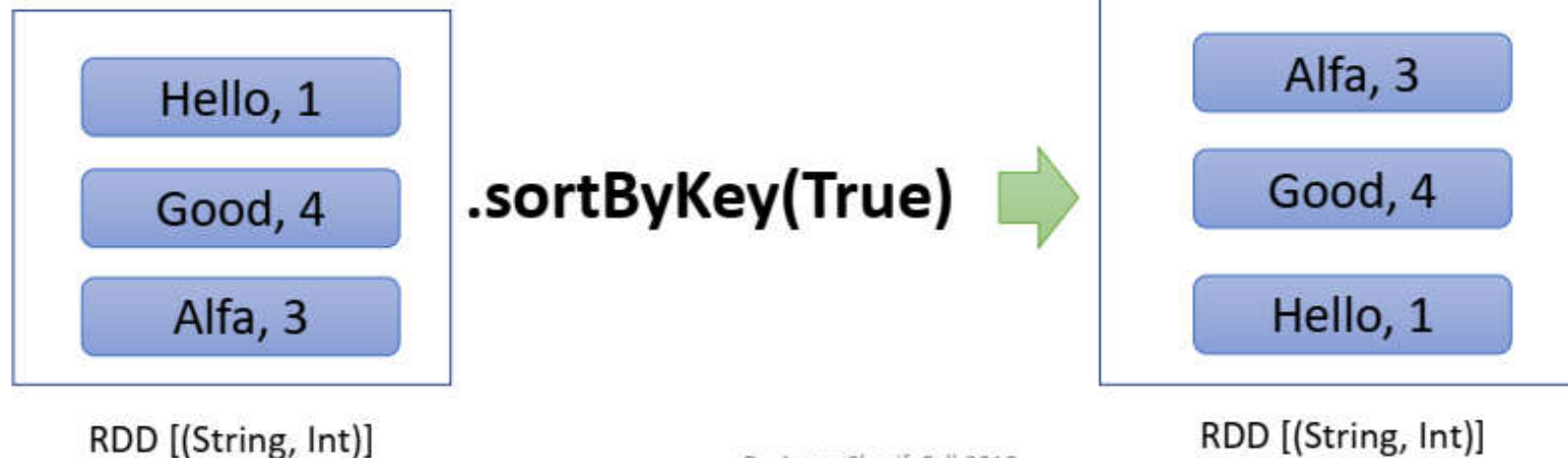
64

□ Exemple : Transformation large « *sortByKey* »

Uniquement applicable sur les RDD[(K,V)]

sortByKey([*ascending*], [*numPartitions*])

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.



Transformations des RDDs:

Transformations Larges

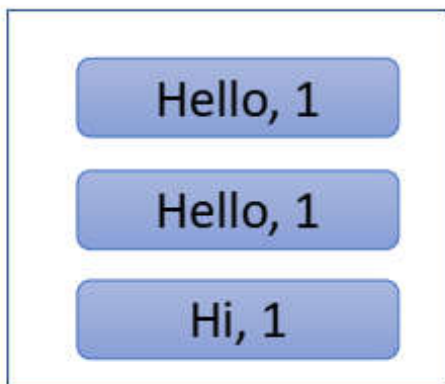
65

□ Exemple : Transformation large « *reduceByKey* »

Uniquement applicable sur les RDD[(K,V)]

reduceByKey(*func*, [numPartitions])

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.



RDD [(String, Int)]

.reduceByKey(lambda x,y: x+y)



RDD [(String, Int)]

Transformations des RDDs:

Transformations Larges

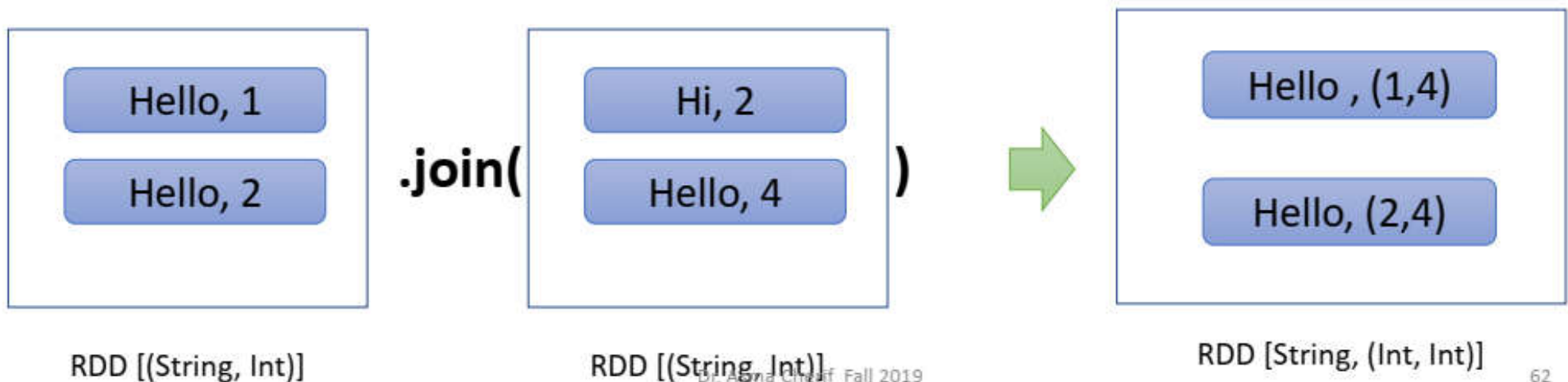
66

□ Exemple : Transformation large « *reduceByKey* »

Uniquement applicable sur les RDD[(K,V)]

join(otherDataset, [numPartitions])

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.



Transformations des RDDs:

Comparaison

72

□ Transformations étroites (narrow)

- Permet d'enchaîner l'exécution sur un nœud du cluster, sans barrière.
- Crash peu coûteux, reconstruction isolée.

□ Transformations larges

- Nécessitent les données de toutes les partitions parentes, barrière dans l'exécution
- Crash coûteux, il faut reconstruire les données de toutes les machines.

Actions sur les RDDs

73

Définition

- Marque la fin d'un flux de donnée :
 - en retournant une valeur résultat à l'application
 - et/ou en exportant les données sur un stockage stable
- Déclenche un **job Spark** :
 - ⇒ déploiement et l'exécution du flux de données sur l'infrastructure

NB : une application Spark peut impliquer plusieurs Jobs Spark

Exemples d'actions simples

- `def max()(implicit ord: Ordering[T]): T`
- `def min()(implicit ord: Ordering[T]): T`
- `def isEmpty(): Boolean` : teste si le RDD est vide
- `def first(): T` : retourne le premier élément du RDD
- `def count(): Long` : retourne la taille du RDD

Actions sur les RDDs

74

Actions pour le contenu

- `def collect(): Array[T]` : Retourne un tableau qui contient tous les éléments du RDD.
- `def take(num: Int): Array[T]` : retourne les num 1er éléments du RDD

⇒ À n'utiliser que pour les phases de debug ou bien sur des RDD relativement petits

Actions de traitement

- `def foreach(f: (T) =>Unit): Unit` : Applique un traitement à chaque élément
- `def reduce(f: (T, T) =>T): T` : Réduit les éléments du RDD en utilisant la fonction commutative et associative f

Actions sur les RDDs

75

Actions de sauvegarde

- `def saveAsObjectFile(path: String): Unit :`
Sauvegarde en tant qu'objets sérialisés dans le fichier path.
- `def saveAsTextFile(path: String): Unit :`
Sauvegarde au format texte en utilisant la représentation String des éléments

Action de sauvegarde pour les RDD[(K,V)]

`def saveAsNewAPIHadoopFile[F <: OutputFormat[K, V]](path: String): Unit :`
Sauvegarde au format Hadoop sur le chemin path.

76

Programmation Spark

Le shell interactif

77

- Le moyen le plus rapide pour apprendre Spark.
- Disponible en Python et Scala.
- Il s'exécute comme une application sur un cluster Spark existant.....
- Ou s'exécute localement.

```

root@ip-172-31-11-254:~ — ssh — 85x22
root@ip-172-31-11-254:~
[root@ip-172-31-11-254 ~]# /opt/cloudera/parcels/SPARK/pyspark
...
Welcome to

  ____  _
 / ___|| | | |
| |___| |_| |
|___ \_  _/ |
   ___|| | | |
  |___||_| |_|

version 0.8.0

Using Python version 2.6.6 (r266:84292, Sep 11 2012 08:34:23)
Spark context available as sc.
...
>>> file = sc.textFile("hdfs://ip-172-31-11-254.us-west-2.compute.internal:8020/user/
hdfs/ec2-data/pageviews/2007/2007-12/pagecounts-20071209-180000.gz")
...
>>> file.count()
...
856769
>>> file.filter(lambda line: "Holiday" in line).count()
...
101

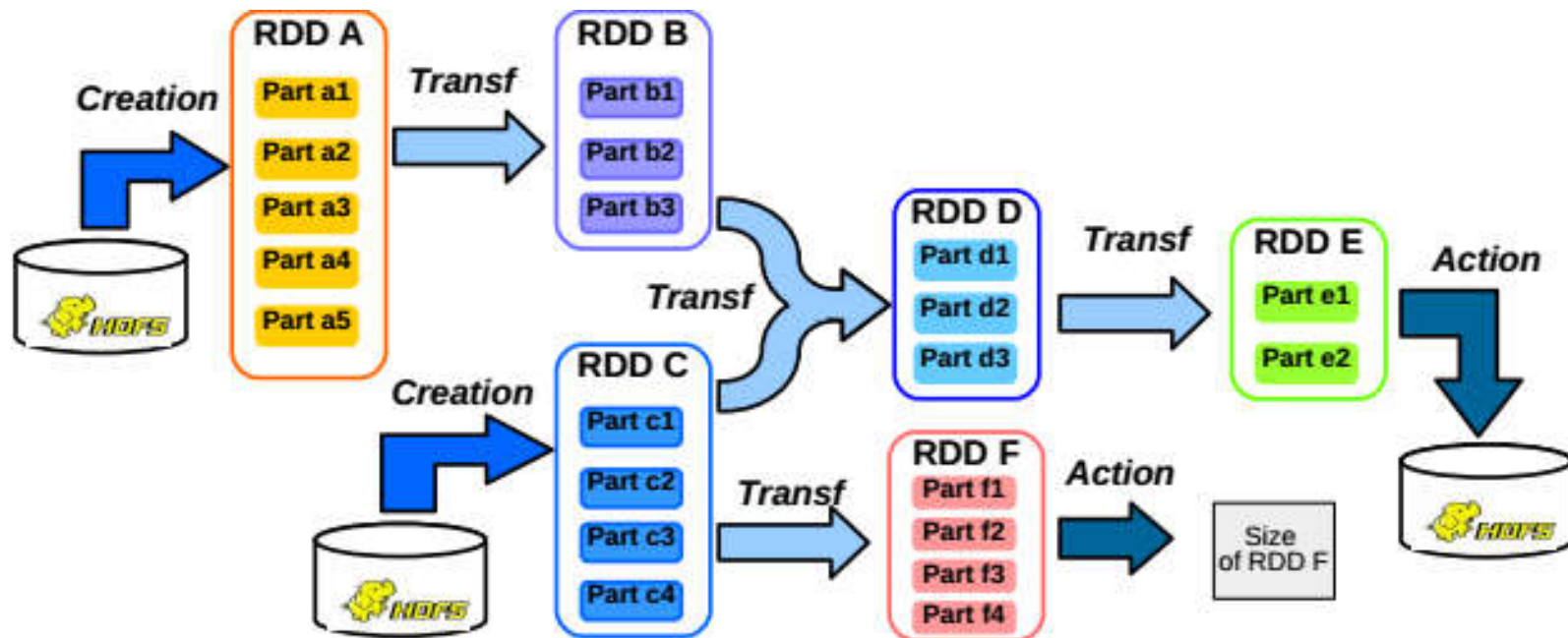
```

Etapes d'un programme Spark

78

□ Les 4 étapes du programme:

1. Initialisation d'un Spark Context
2. Expression de la création du ou des premier(s) RDD
3. Expression des transformations entre RDD
4. Application des actions sur les RDD finaux



Création d'un contexte Spark

79

- Le SparkContext est la couche d'abstraction qui permet à Spark de savoir où il va s'exécuter.
- Un SparkContext standard sans paramètres correspond à l'exécution en local sur 1 CPU du code Spark qui va l'utiliser.


Scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sc = new SparkContext("url", "name", "sparkHome", Seq("app.jar"))
```

Java

```
import org.apache.spark.SparkContext
JavaSparkContext sc = new JavaSparkContext(
    "masterUrl", "name", "sparkHome", new String[] {"app.jar"});
```



Python

```
from pyspark import SparkContext

sc = SparkContext("masterUrl", "name", "sparkHome", ["library.py"])
```

Création de RDDs

Turn a Python collection into an RDD

> `sc.parallelize([1, 2, 3])`

Load text file from local FS, HDFS, or S3

> `sc.textFile("file.txt")`

> `sc.textFile("directory/*.txt")`

> `sc.textFile("hdfs://namenode:9000/path/file")`

> Load key-value file (sequence file)

> `sc.sequenceFile("hdfs://share/data1.seq")`

>

Use existing Hadoop InputFormat (Java/Scala only)

> `sc.hadoopFile(keyClass, valClass, inputFmt, conf)`

Transformations Basiques

```
> nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
> squares = nums.map(lambda x: x*x)    // {1, 4, 9}

# Keep elements passing a predicate
> even = squares.filter(lambda x: x % 2 == 0) // {4}

# Map each element to zero or more others
> nums.flatMap(lambda x: => range(x))
  > # => {0, 0, 1, 0, 1, 2}
```

Range object (sequence
of numbers 0, 1, ..., x-1)

Actions Basiques

```
> nums = sc.parallelize([1, 2, 3])  
  
# Retrieve RDD contents as a local collection  
> nums.collect() # => [1, 2, 3]  
  
# Return first K elements  
> nums.take(2)    # => [1, 2]  
  
# Count number of elements  
> nums.count()    # => 3  
  
# Merge elements with an associative function  
> nums.reduce(lambda x, y: x + y) # => 6  
  
# Write elements to a text file  
> nums.saveAsTextFile("hdfs://file.txt")  
> Write pairs to a sequence file  
> RDD.saveAsSequenceFile("hdfs://share/data2.seq")
```

Quelques Operations Key-Value

- `pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])`
- `pets.reduceByKey(lambda x, y: x + y)`
=> {(cat, 3), (dog, 1)}
- `pets.groupByKey()` # => {(cat, [1, 2]), (dog, [1])}
- `pets.sortByKey()` # => {(cat, 1), (cat, 2), (dog, 1)}
- `reduceByKey` also automatically implements combiners on the map side

Quelques Operations Key-Value

- `visits = sc.parallelize([("index.html", "1.2.3.4"),
("about.html", "3.4.5.6"),
("index.html", "1.3.3.1")])`
- `pageNames = sc.parallelize([("index.html", "Home"),
("about.html", "About")])`
- `visits.join(pageNames)`
`("index.html", ("1.2.3.4", "Home"))`
`("index.html", ("1.3.3.1", "Home"))`
`("about.html", ("3.4.5.6", "About"))`
- `visits.cogroup(pageNames)`
`("index.html", (["1.2.3.4", "1.3.3.1"], ["Home"]))`
`("about.html", (["3.4.5.6"], ["About"]))`

Example : Log Mining

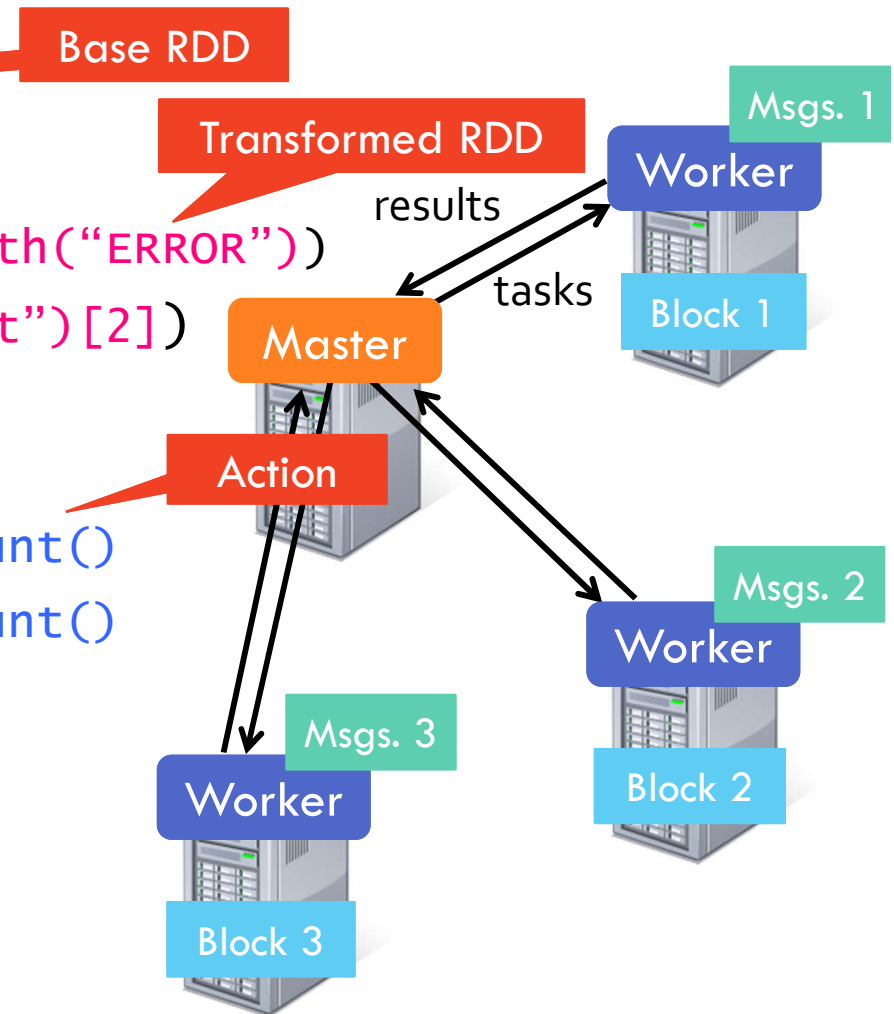
88

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERR"))
messages = errors.map(lambda s: s.split("\t")[2])
cachedMsgs = messages.cache()

// rien ne se passe avant l'appel des actions

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
```

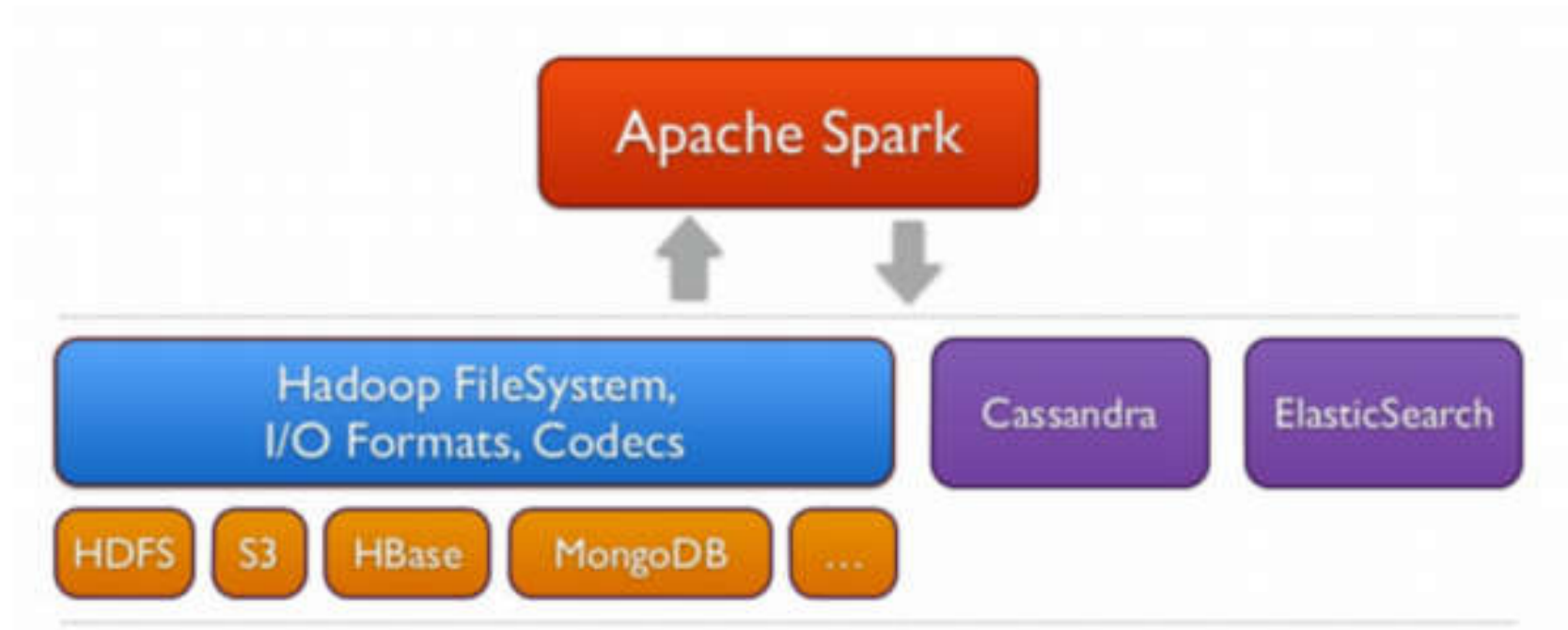



90

Application 1: WordCount

Application 1: WordCount

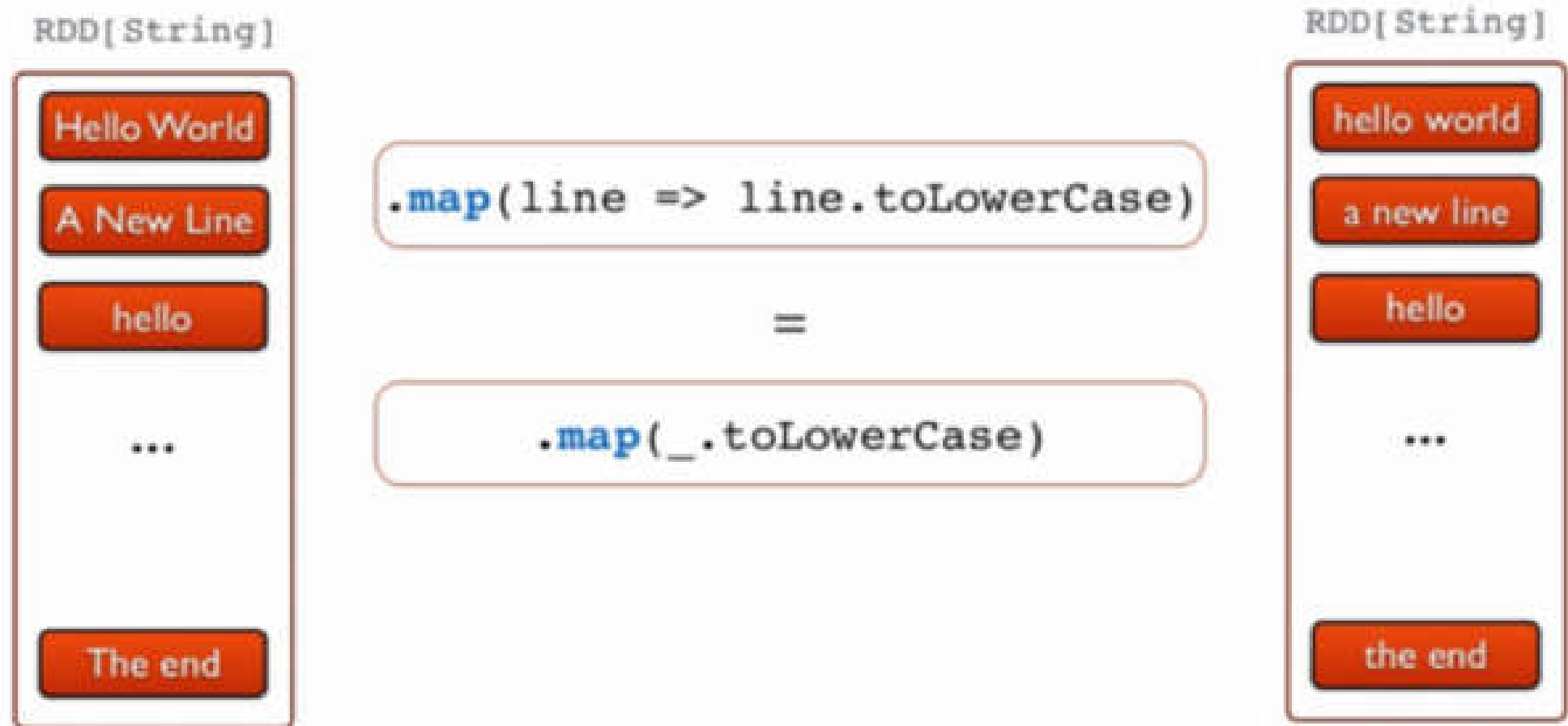
91



```
//Etape 1 - Créer un RDD à partir d'un fichier texte de Hadoop   
val docs = spark.textFile("/docs")
```

Application 1: WordCount

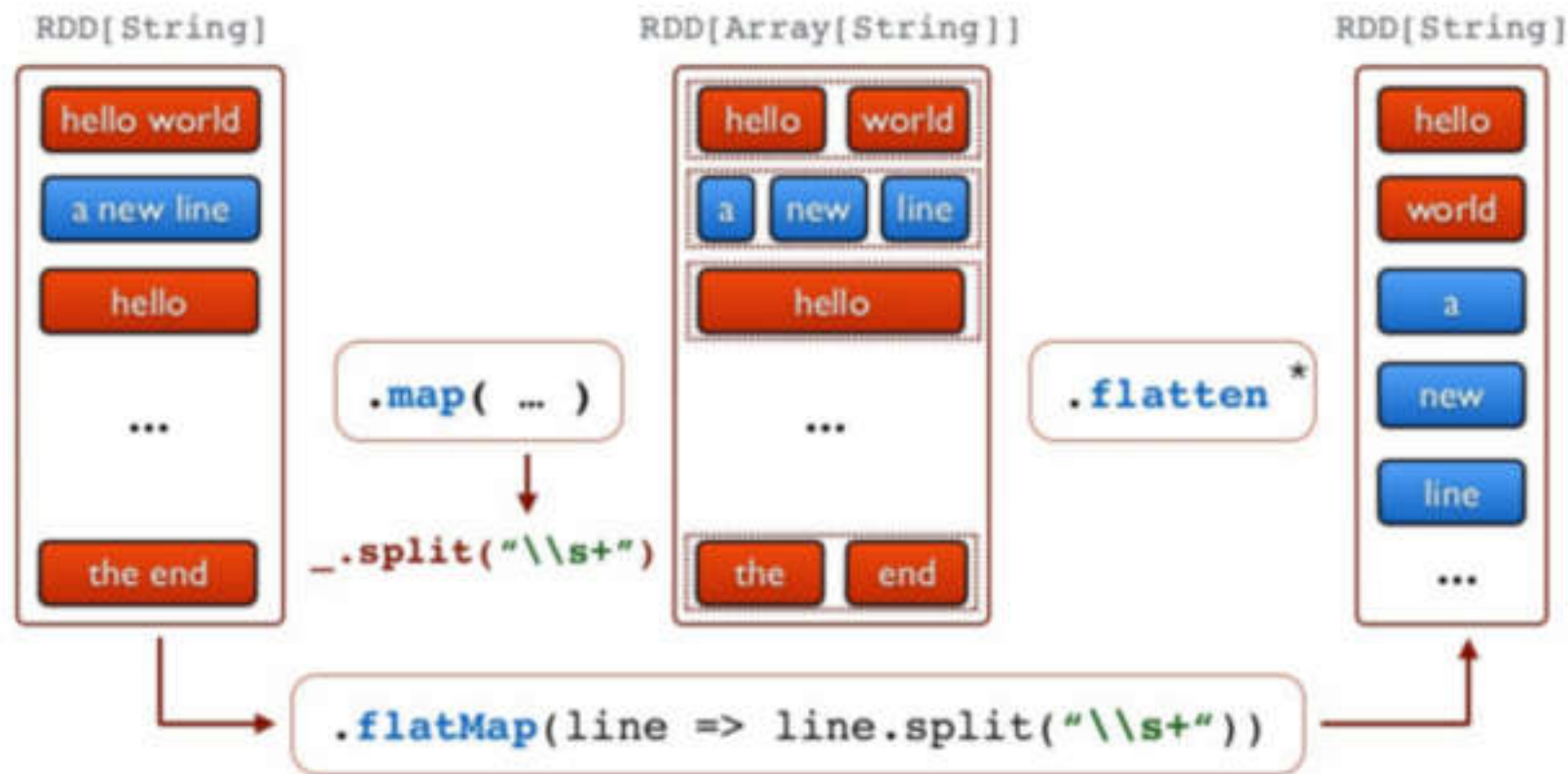
92



```
//Etape 2 - Convertir les lignes en minuscule  
val lower = docs.map(line => line.toLowerCase)
```

Application 1: WordCount

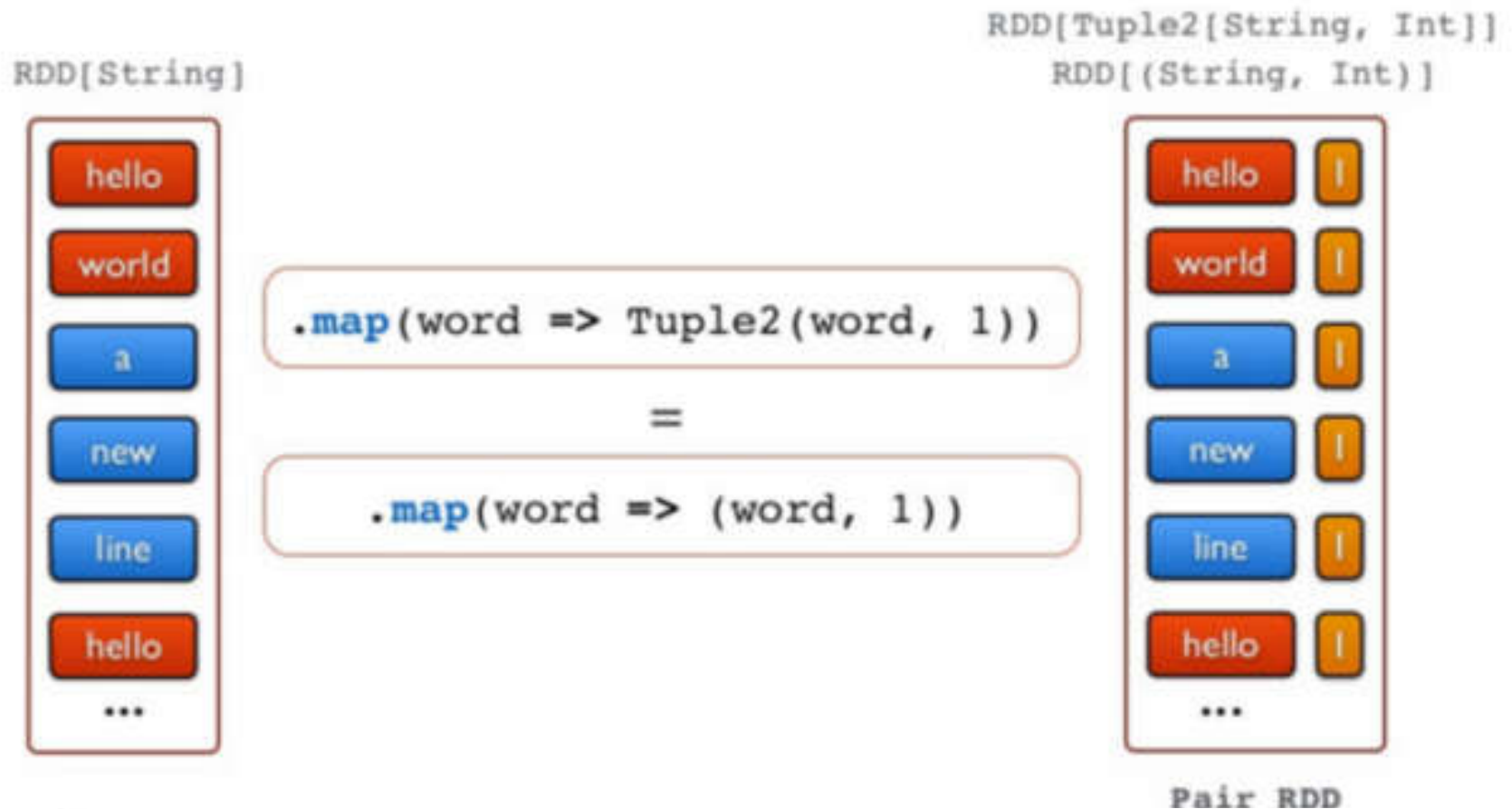
93



```
//Etape 3 - Séparer les lignes en mots  
val words = lower.flatMap(line => line.split("\\s+"))
```

Application 1: WordCount

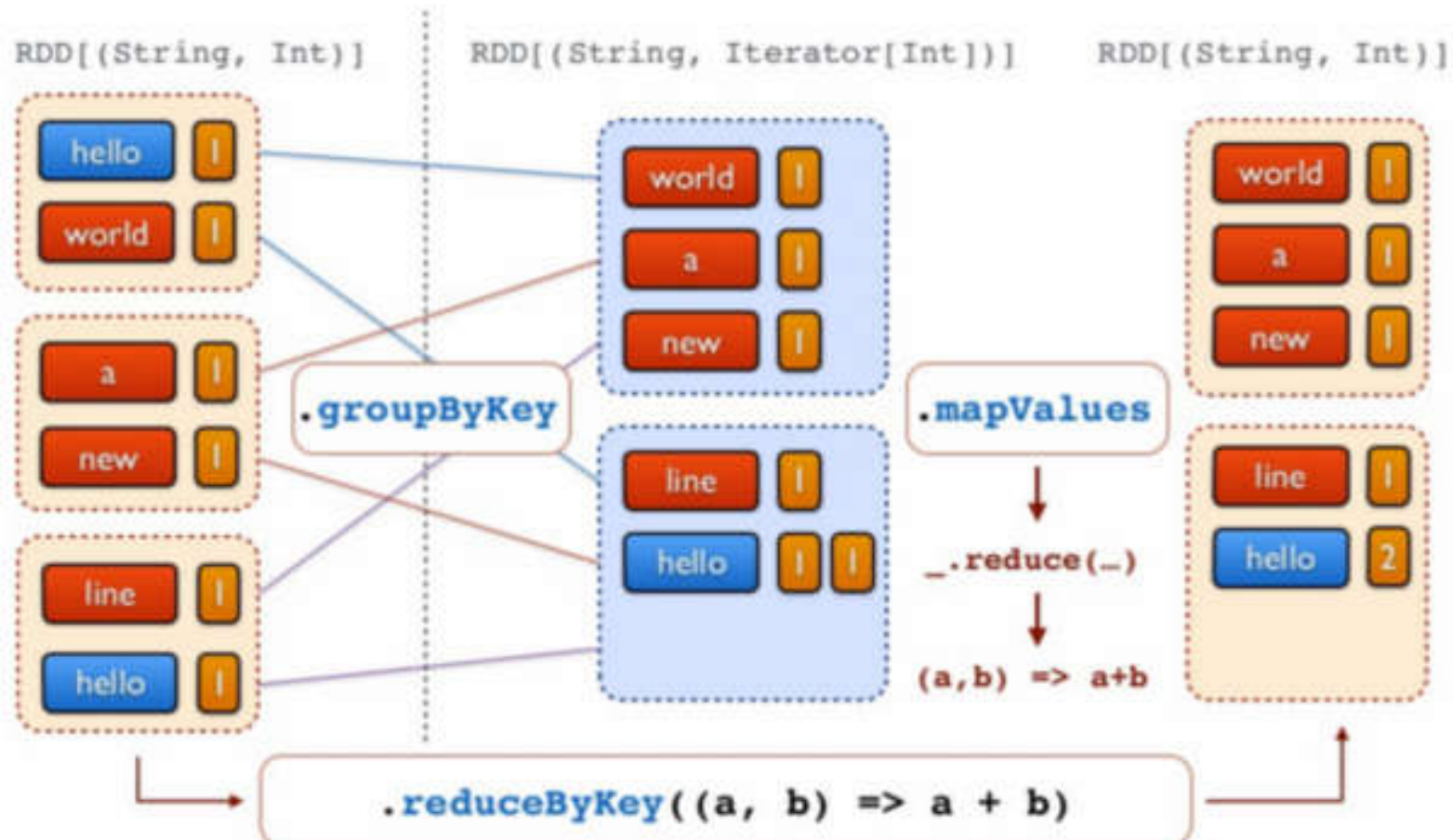
94



```
//Etape 4 - produire les tuples (mot, 1)  
val counts = words.map(word => (word,1))
```

Application 1: WordCount

95



```
//Etape 5 - Compter tous les mots  
val freq = counts.reduceByKey(_ + _)
```



Application 2: PageRank

PageRank: C'est quoi?

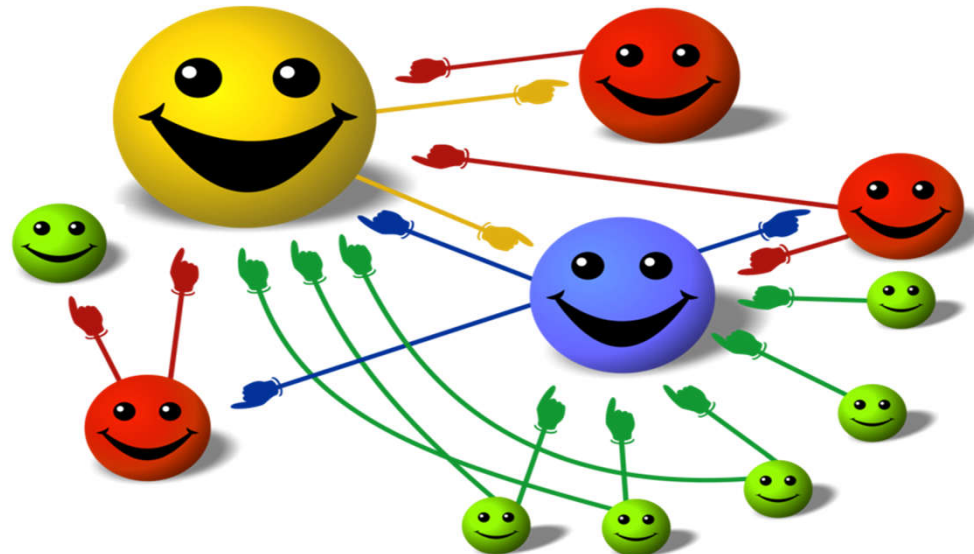
97

- Algorithme proposé par Larry Page and Sergey Brin, les fondateurs de Google, pour mesurer l'importance d'une page web.
- Il est utilisable pour les graphes et les réseaux.
- C'est un bon exemple d'algorithme complexe:
 - ▣ Plusieurs phases de map & reduce
- Il tire profit de la mise en cache de Spark
 - ▣ Il effectue plusieurs iterations sur les mêmes données.

Page Rank: Idée

98

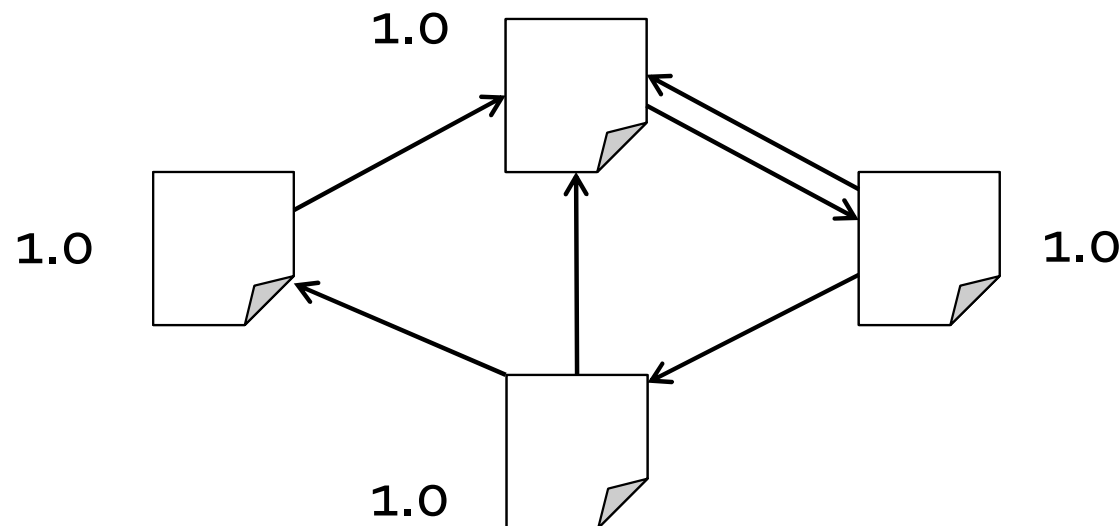
- Donner aux pages Web des scores (Ranks) sur la base des liens entrants à ces pages.
- ▣ Liens à partir de plusieurs page → score élevé
- ▣ Lien à partir d'une page avec un score élevé → score élevé.



Page Rank: algorithme

99

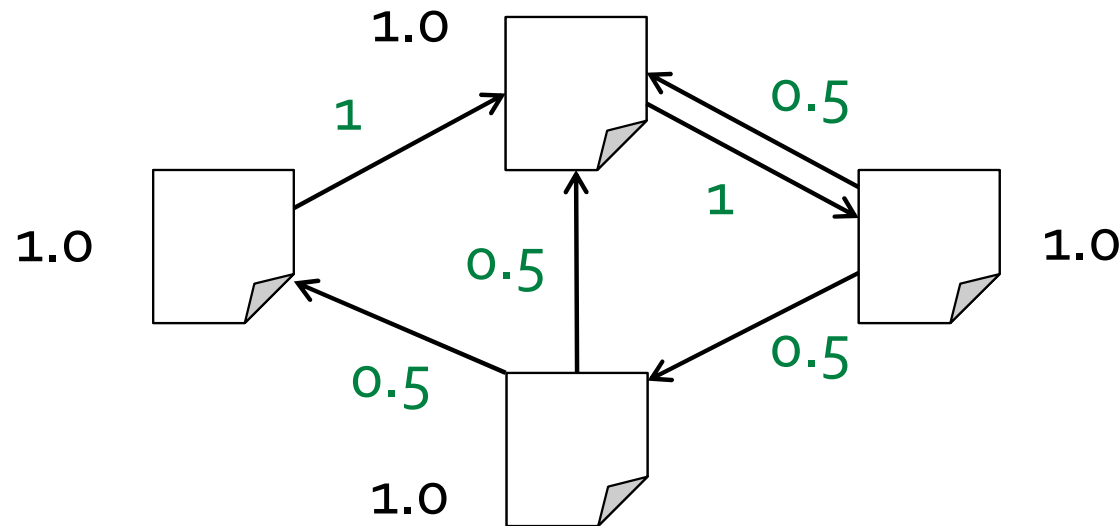
1. Le score de chaque page est initialisé à **1**.
2. À chaque itération, une page **p** envoie le score *contrib(p)*
$$\text{contrib}(p) = \text{rank}(p) / |\text{Sortants}(p)|$$
3. Le score de chaque page **q** est calculé comme étant:
$$\text{rank}(q) = 0.15 + 0.85 * \sum \text{contibs}(p), p \in \text{Entrants}(q)$$



Page Rank: algorithme

100

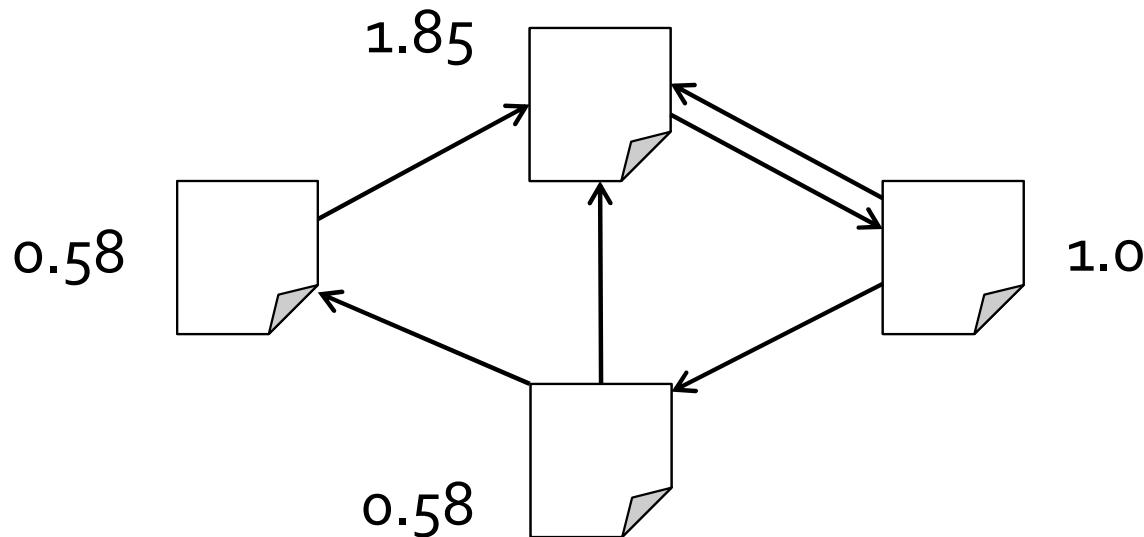
1. Le score de chaque page est initialisé à **1**.
2. À chaque itération, une page **p** envoie le score *contrib(p)*
$$\text{contrib}(p) = \text{rank}(p) / |\text{Sortants}(p)|$$
3. Le score de chaque page **q** est calculé comme étant:
$$\text{rank}(q) = 0.15 + 0.85 * \sum \text{contibs}(p), p \in \text{Entrants}(q)$$



Page Rank: algorithme

101

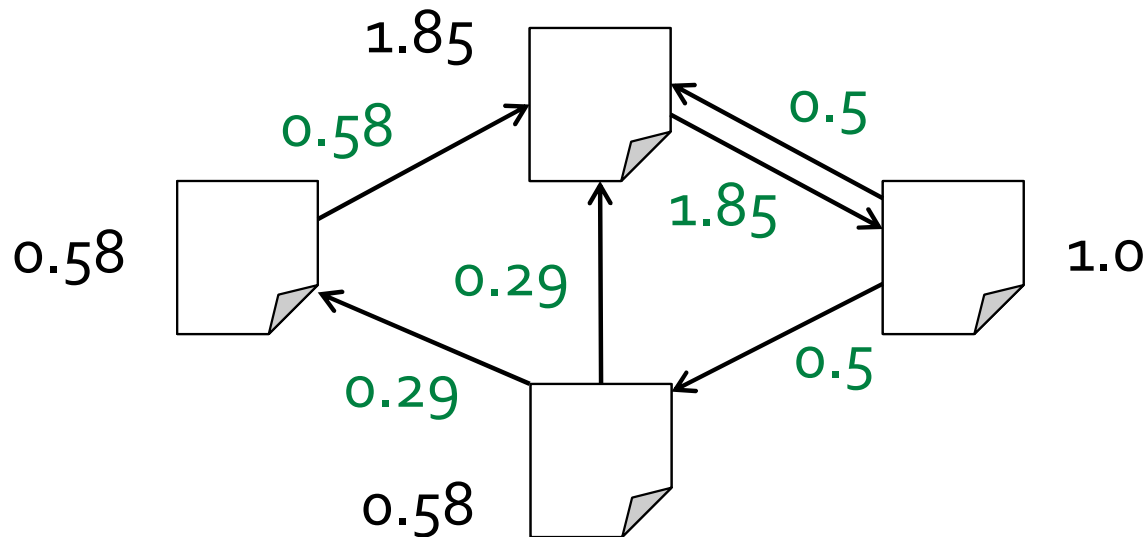
1. Le score de chaque page est initialisé à **1**.
2. À chaque itération, une page **p** envoie le score *contrib(p)*
$$\text{contrib}(p) = \text{rank}(p) / |\text{Sortants}(p)|$$
3. Le score de chaque page **q** est calculé comme étant:
$$\text{rank}(q) = 0.15 + 0.85 * \sum \text{contibs}(p), p \in \text{Entrants}(q)$$



Page Rank: algorithme

102

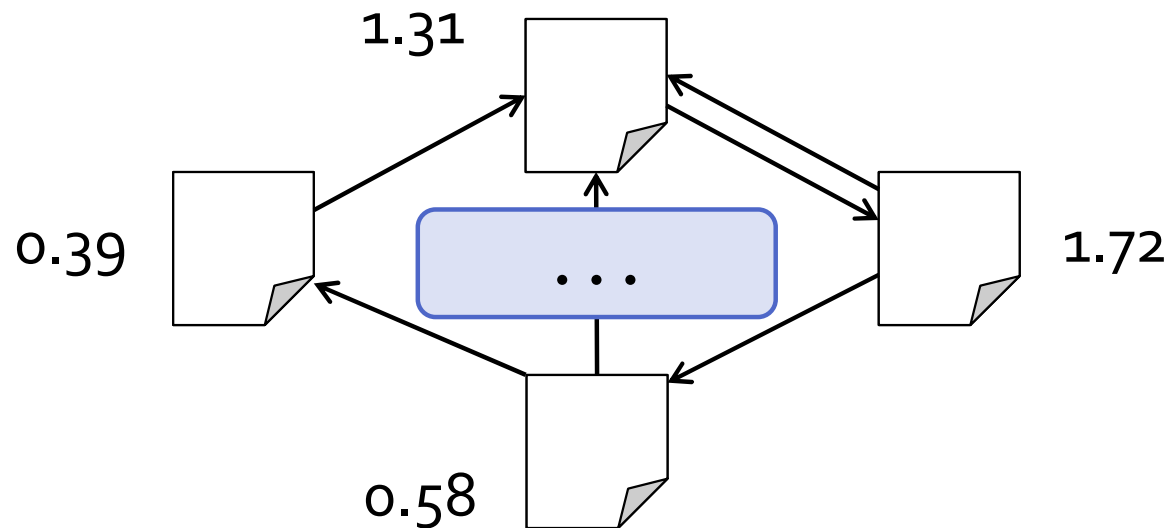
1. Le score de chaque page est initialisé à **1**.
2. À chaque itération, une page **p** envoie le score *contrib(p)*
$$\text{contrib}(p) = \text{rank}(p) / |\text{Sortants}(p)|$$
3. Le score de chaque page **q** est calculé comme étant:
$$\text{rank}(q) = 0.15 + 0.85 * \sum \text{contibs}(p), p \in \text{Entrants}(q)$$



Page Rank: algorithme

103

1. Le score de chaque page est initialisé à **1**.
2. À chaque itération, une page **p** envoie le score *contrib(p)*
$$\text{contrib}(p) = \text{rank}(p) / |\text{Sortants}(p)|$$
3. Le score de chaque page **q** est calculé comme étant:
$$\text{rank}(q) = 0.15 + 0.85 * \sum \text{contibs}(p), p \in \text{Entrants}(q)$$

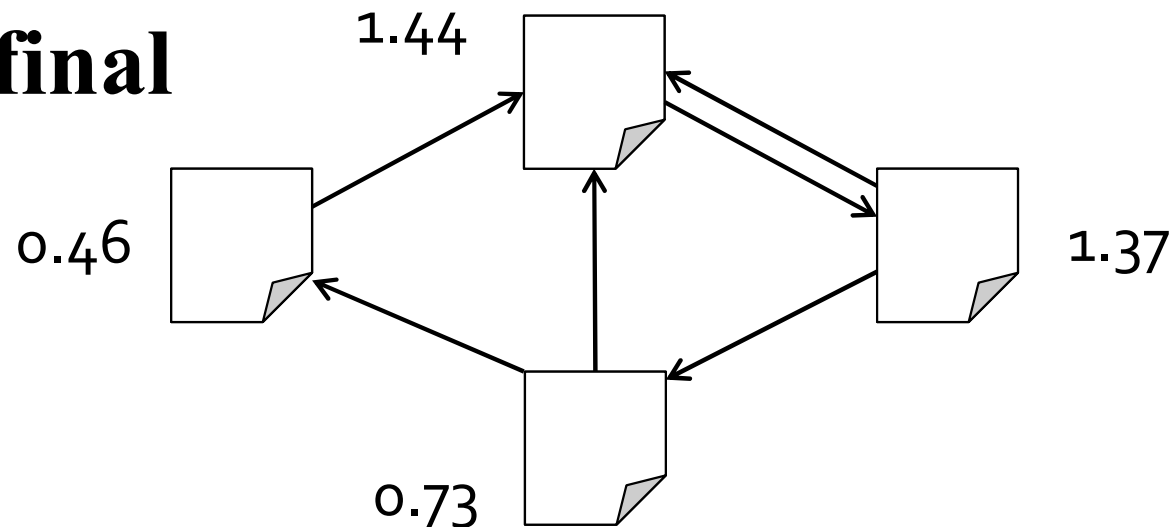


Page Rank: algorithme

104

1. Le score de chaque page est initialisé à **1**.
2. À chaque itération, une page **p** envoie le score *contrib(p)*
$$\text{contrib}(p) = \text{rank}(p) / |\text{Sortants}(p)|$$
3. Le score de chaque page **q** est calculé comme étant:
$$\text{rank}(q) = 0.15 + 0.85 * \sum \text{contibs}(p), p \in \text{Entrants}(q)$$

Etat final



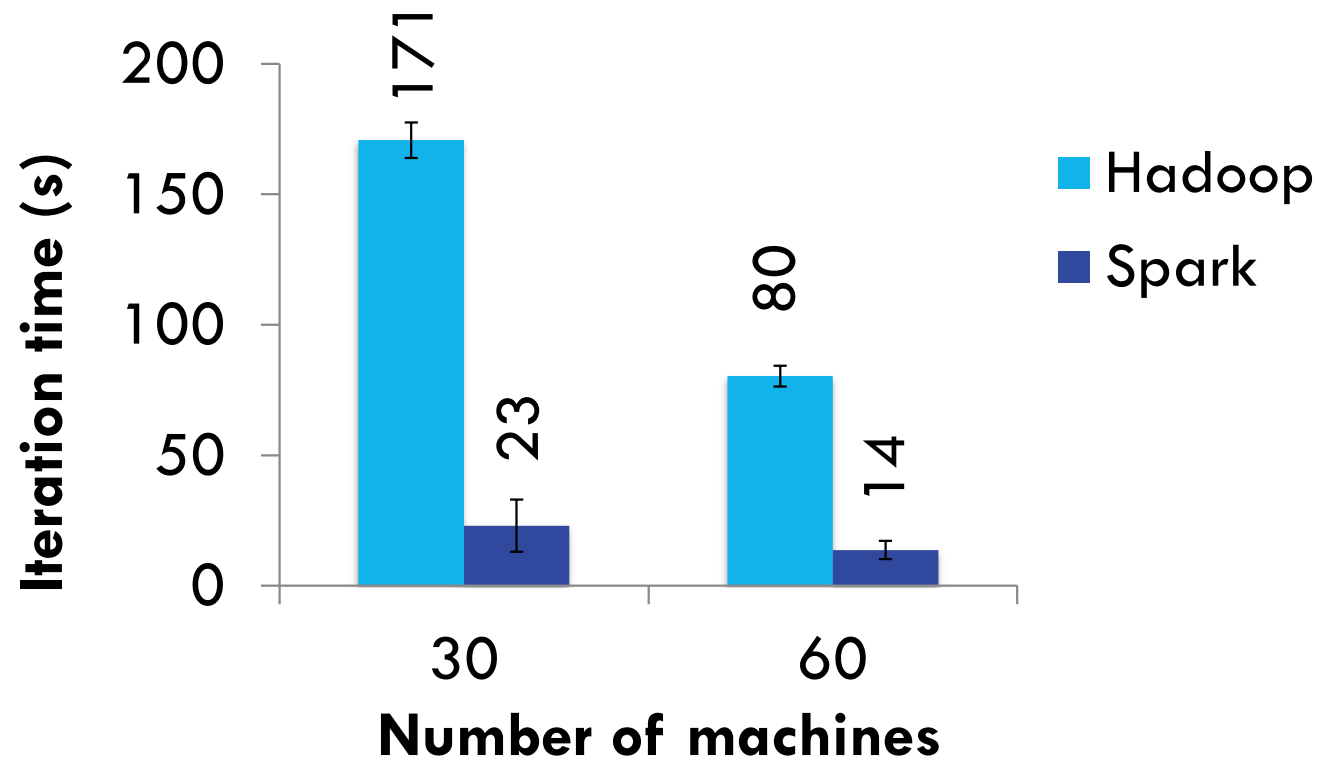
Page Rank : programme Spark

105

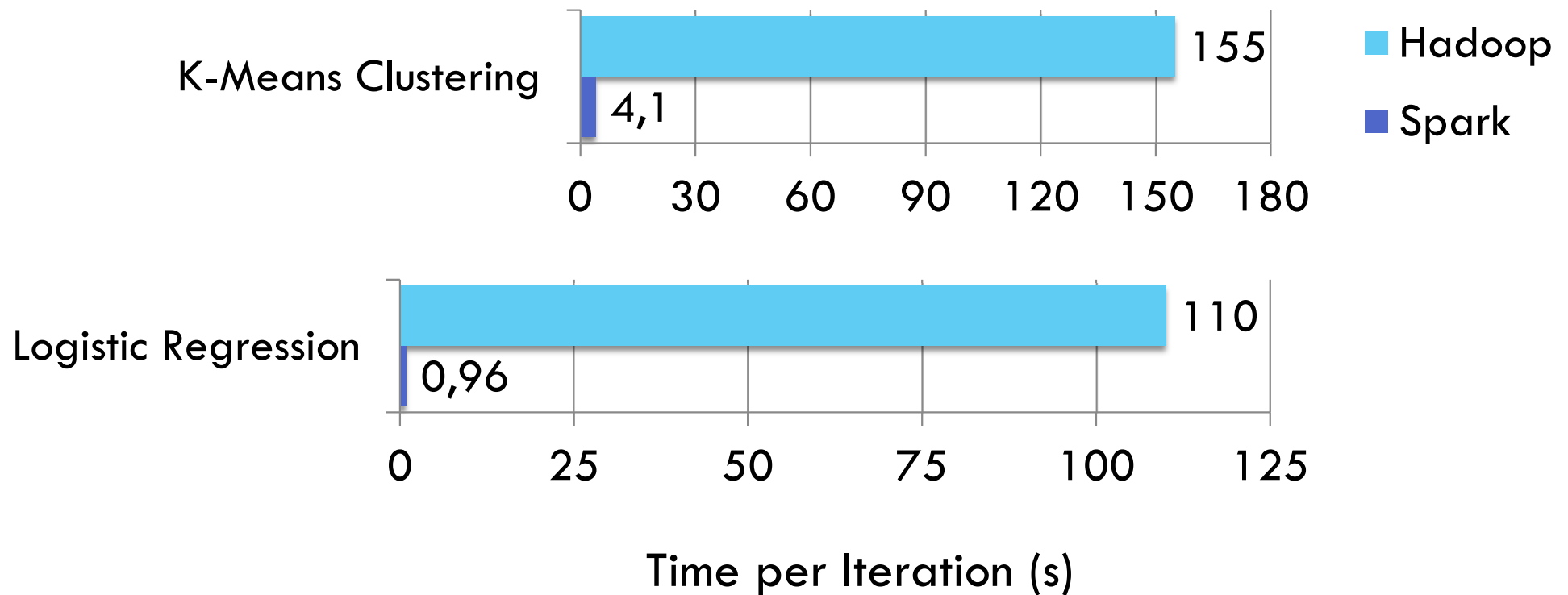
- ☐ Donner l'implémentation de l'algorithme PageRank en PySpark.

Performance de PageRank

106



Performances d'autres algorithmes itératifs



108

Architecture Spark

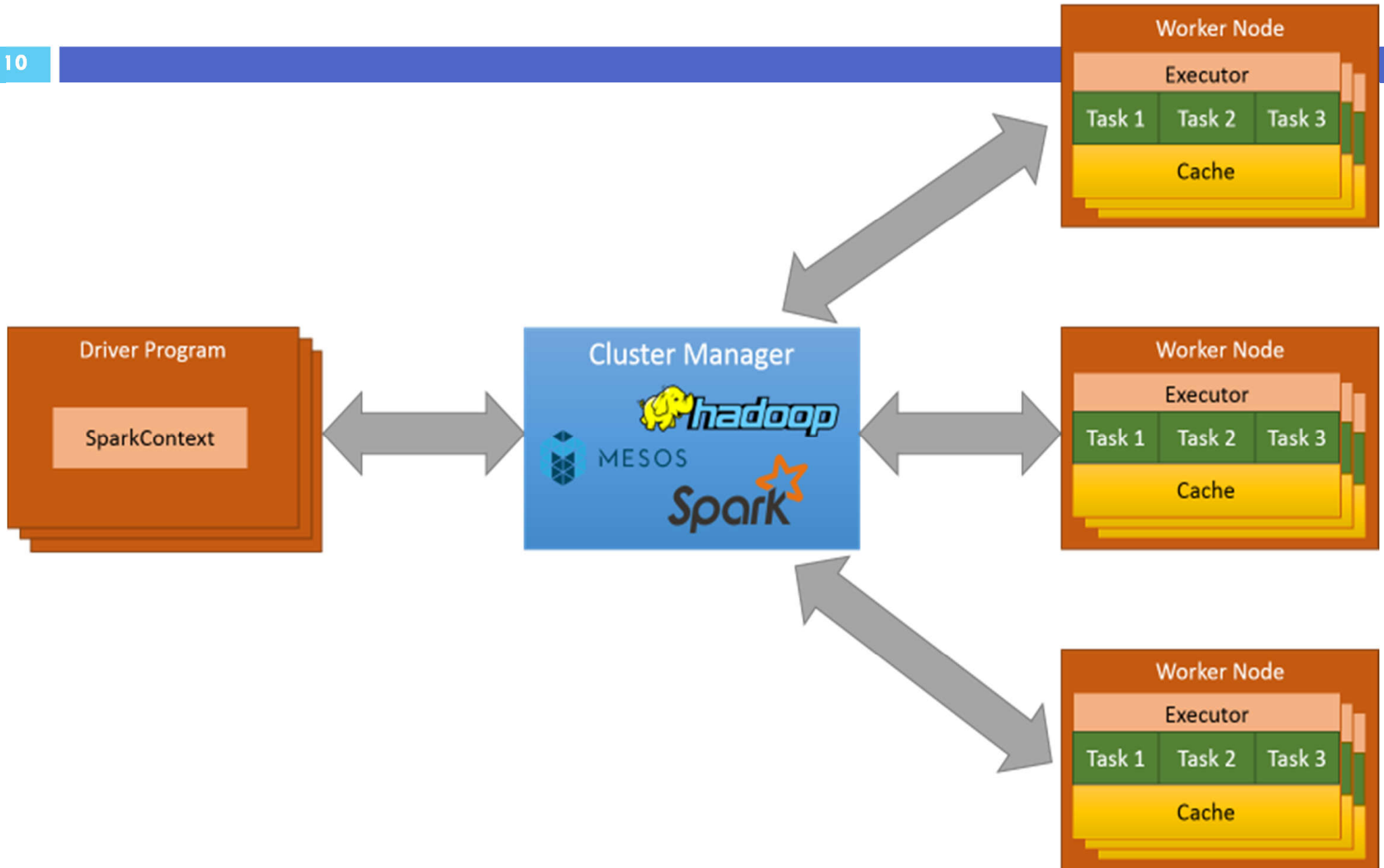
Spark: Architecture

109

- 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

110



Spark: Architecture

111

□ Driver

- ▣ Point d'entrée du Spark Shell (Scala, Python, R)
- ▣ L'endroit où SparkContext est créé
- ▣ Traduit RDD en graphe d'exécution
- ▣ Divise le graphe en stages
- ▣ Planifie les tâches et contrôle leur exécution
- ▣ Stocke des métadonnées sur tous les RDD et leurs partitions
- ▣ Lance Spark WebUI avec des informations sur les tâches

Spark: Architecture

112

□ Executor:

- ▣ Stocke les données dans le cache dans la JVM ou sur les disques durs (HDD).
- ▣ Lit les données à partir de sources externes.
- ▣ Écrit les données dans des sources externes.
- ▣ Exécute tout le traitement de données

Spark: Architecture

113

- 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.

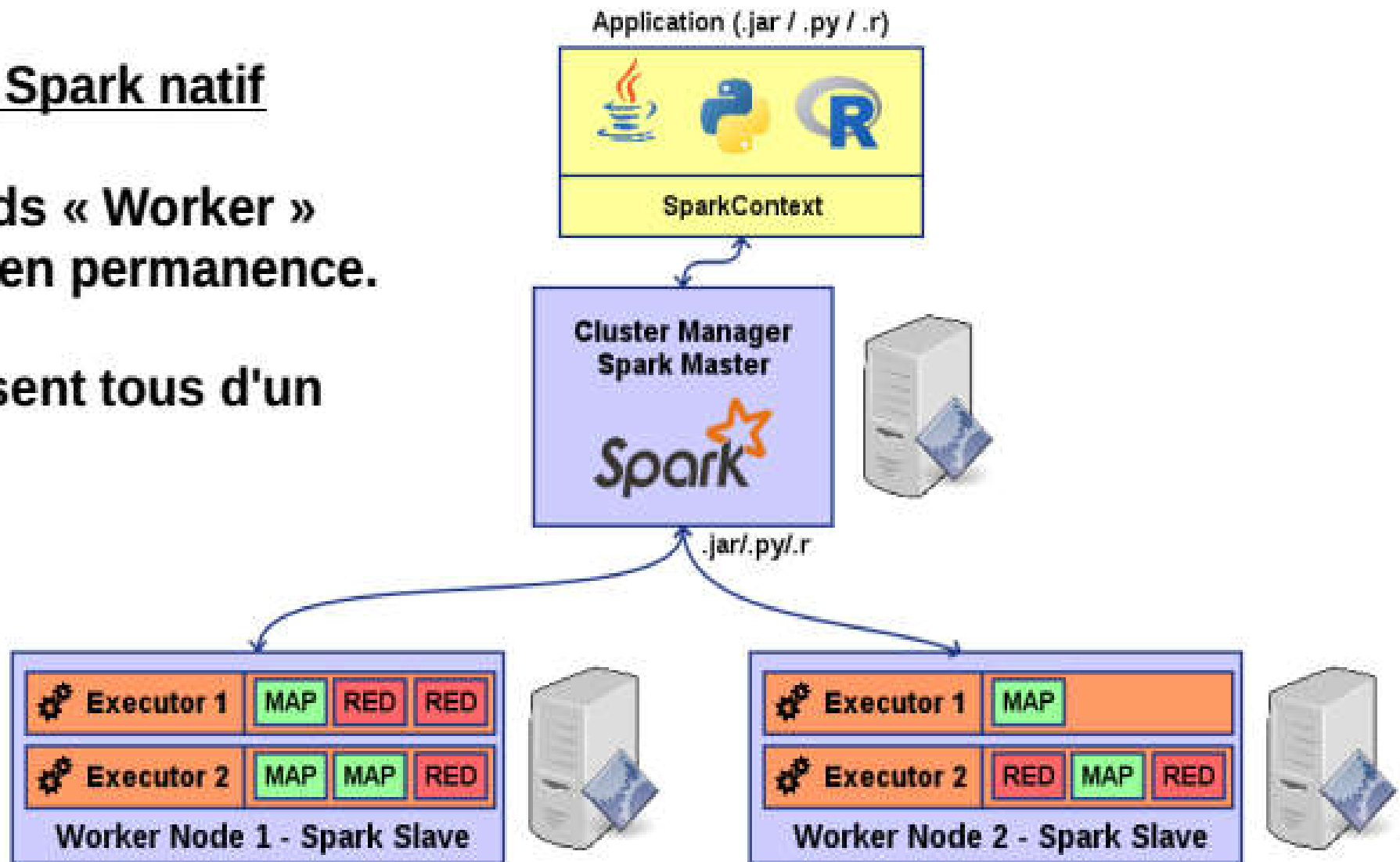
Spark: Architecture

114

En mode Spark natif

Les nœuds « Worker »
tournent en permanence.

Ils disposent tous d'un
cache.



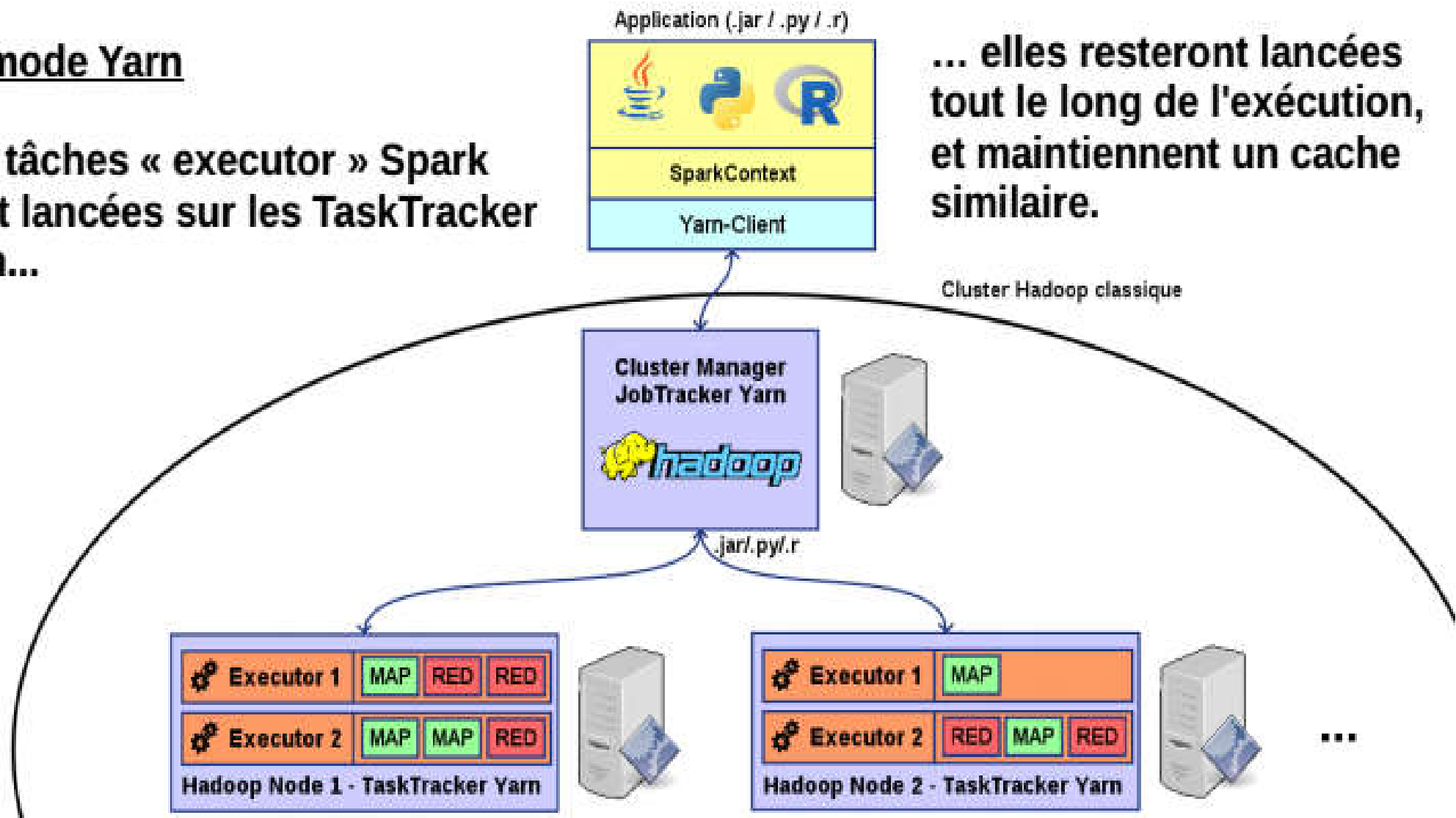
Spark: Architecture

115

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

116

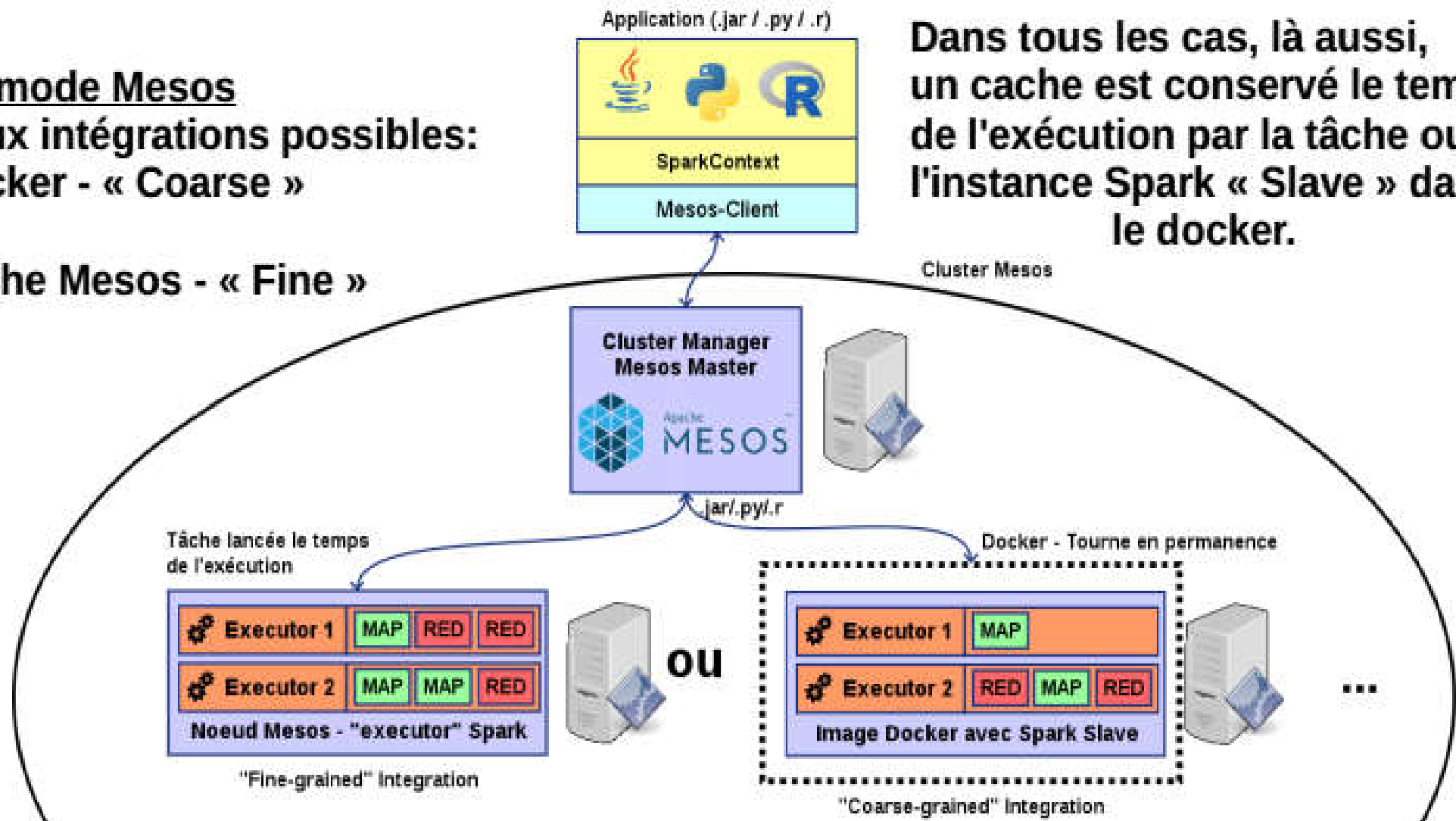
En mode Mesos

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 : Mode de fonctionnement

117

- 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-distribué** : le gestionnaire de ressource est entièrement déployé sur l'ensemble des machines du cluster.

120

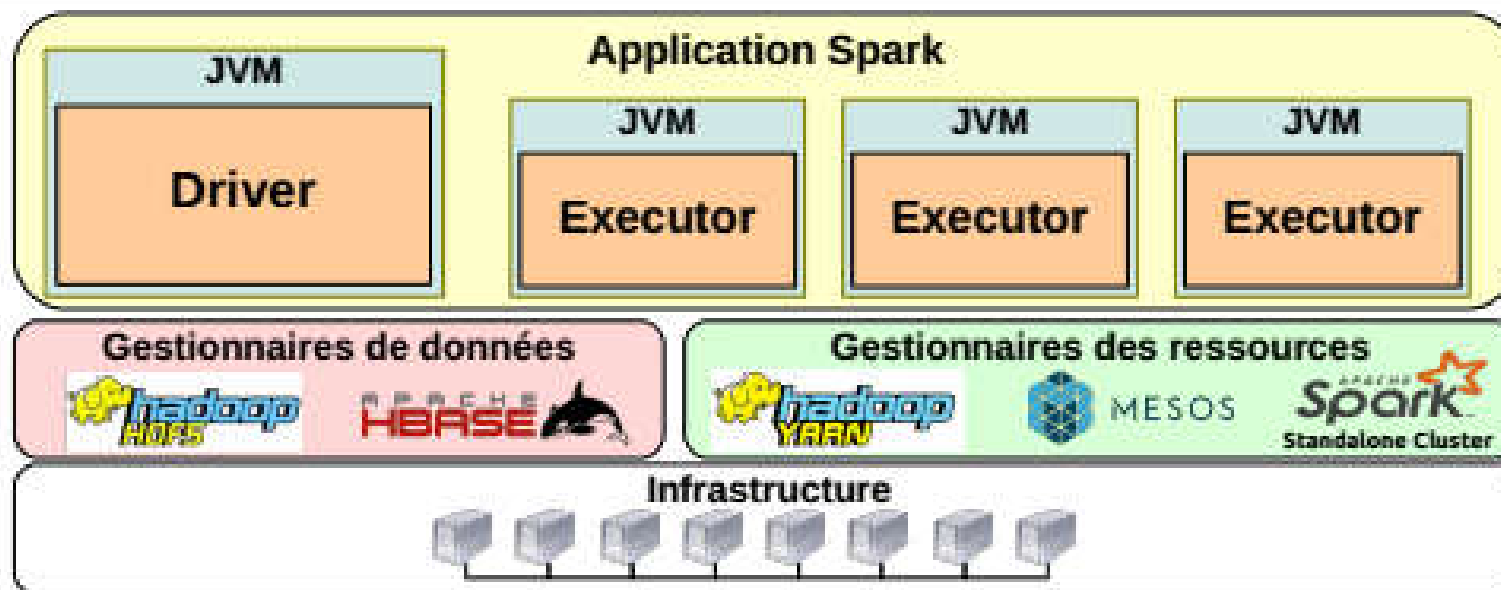
Exécution d'une application Spark

Application Spark

121

Caractéristiques

- Exécution d'un programme spark sur l'infrastructure
- Associée à un sparkcontext
- Mise en place de plusieurs JVM :
 - une JVM maître : **le driver**
 - des JVM esclaves : **les executors**
- Utilisation de gestionnaires de ressources et de données



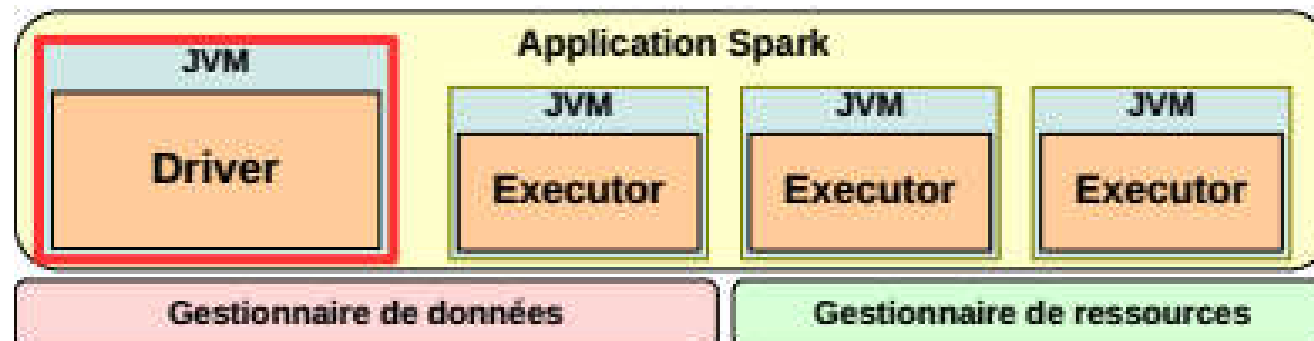
Application Spark : Le Driver

122

Objectifs

JVM maître exécutant le main de l'application

- Interaction avec les gestionnaires de ressources et données
- Définition des tâches :
 - code
 - placement
 - dépendances (transfert de données)
- Orchestration de l'exécution des tâches :
 - Affectation sur les executors
 - Surveillance des tâches terminées ou défaillante

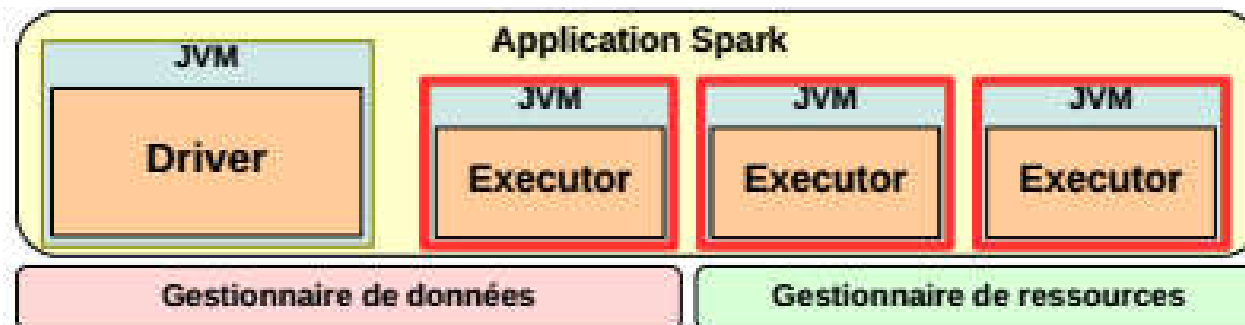


Application Spark : les exécuteurs

123

Caractéristiques

- JVM esclave exécutant les tâches de l'application
- Communication Driver → Executor :
 - affectation de nouvelles tâches
 - annulation de tâches
- Communication Executor → Driver :
 - notification de l'avancement des tâches
- Communication Executor-Executor
 - échange de données entre tâches dépendantes



Exécution: Etape 1 (SparkContexte)

124

Initialisation du sparkcontext

- Prise en compte de la configuration
- Construction des méta-données de l'application

Application Spark

JVM Driver

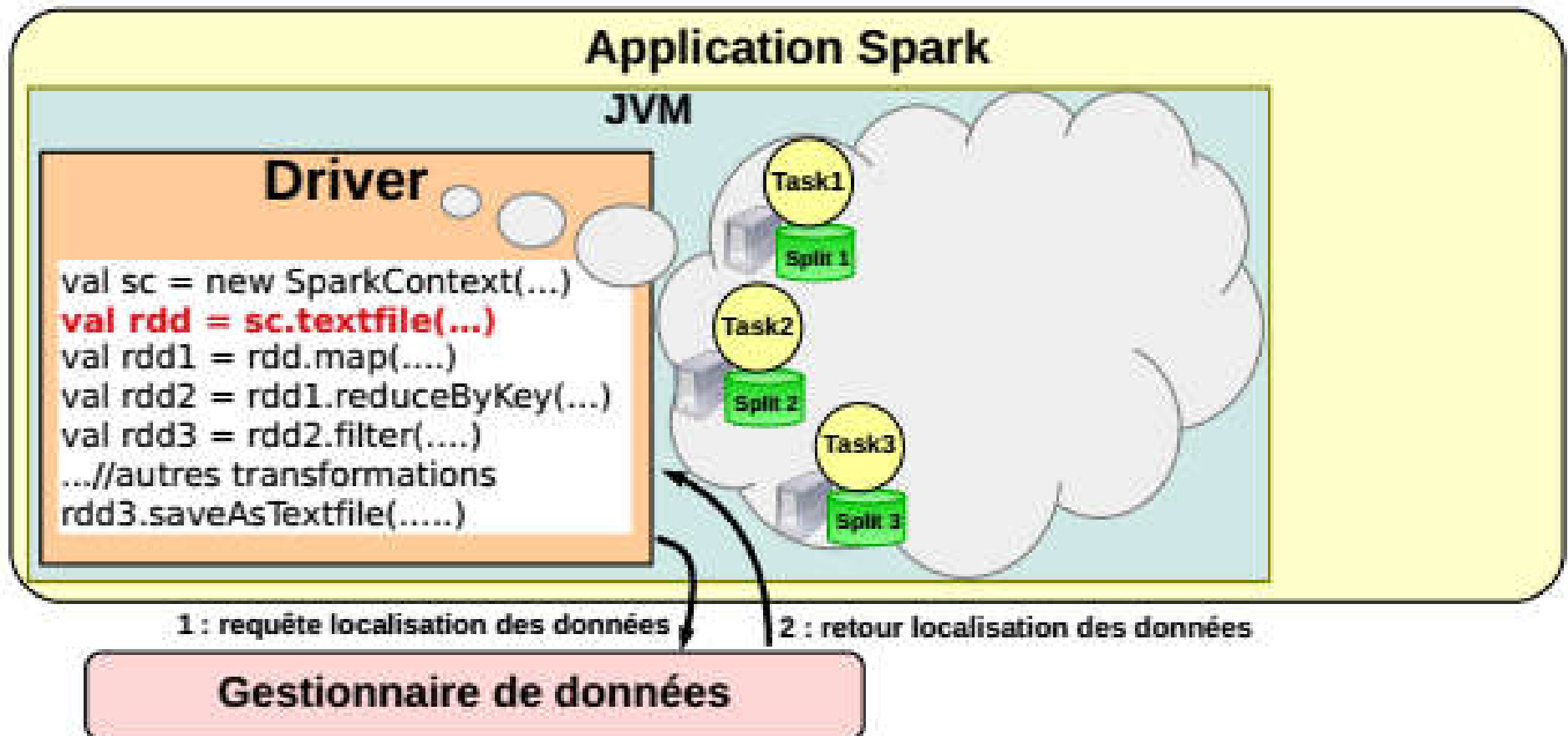
```
val sc = new SparkContext(...)
val rdd = sc.textfile(...)
val rdd1 = rdd.map(...)
val rdd2 = rdd1.reduceByKey(...)
val rdd3 = rdd2.filter(...)
...//autres transformations
rdd3.saveAsTextfile(...)
```


Exécution: Etape 2 (Création)

125

Création des premiers RDD

- Détermination des emplacements des tâches racines
- Prise en compte de la localisation des splits (1 tâche par split)

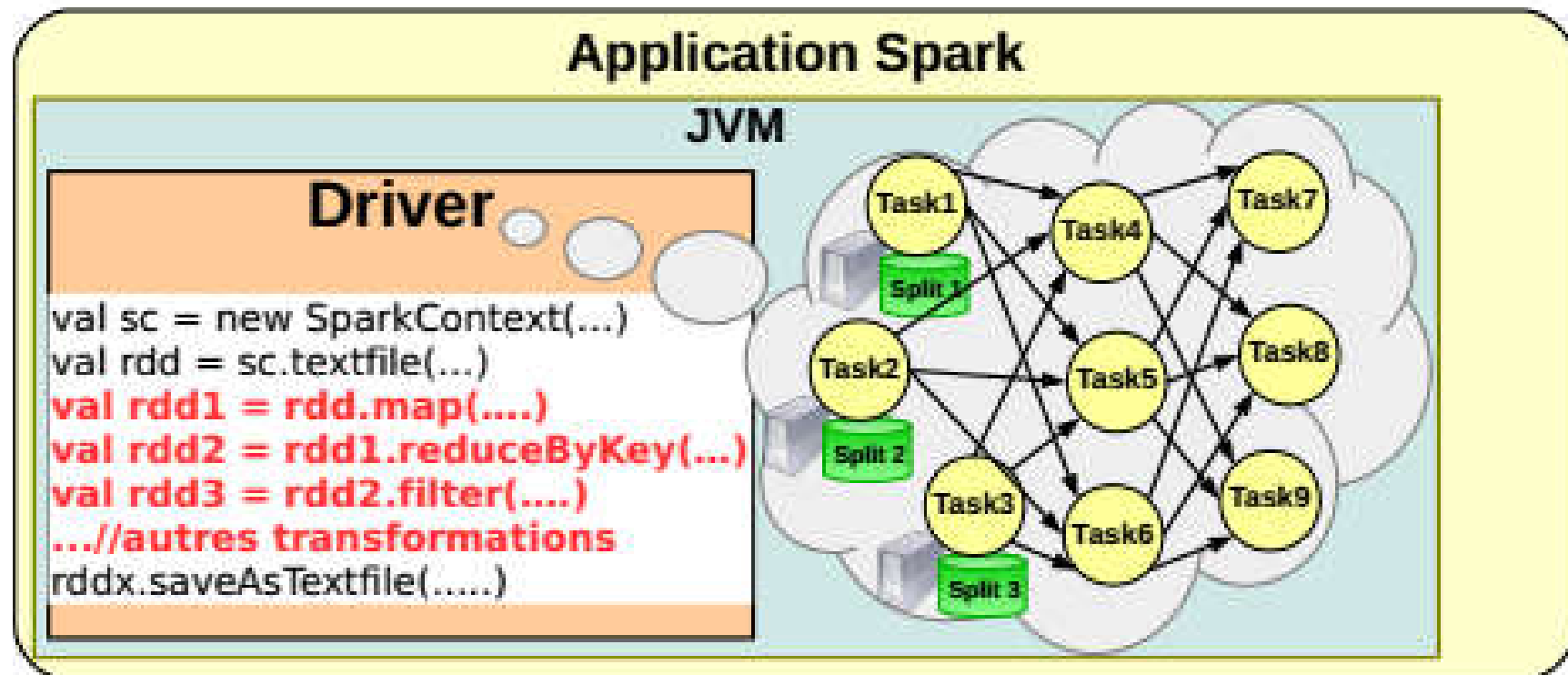


Exécution: Etape 3 (Transformations)

126

Transformations de RDDs

- Traduction des transformations en graphe dirigé acyclique de tâches
- Optimisation des communications inter-tâches

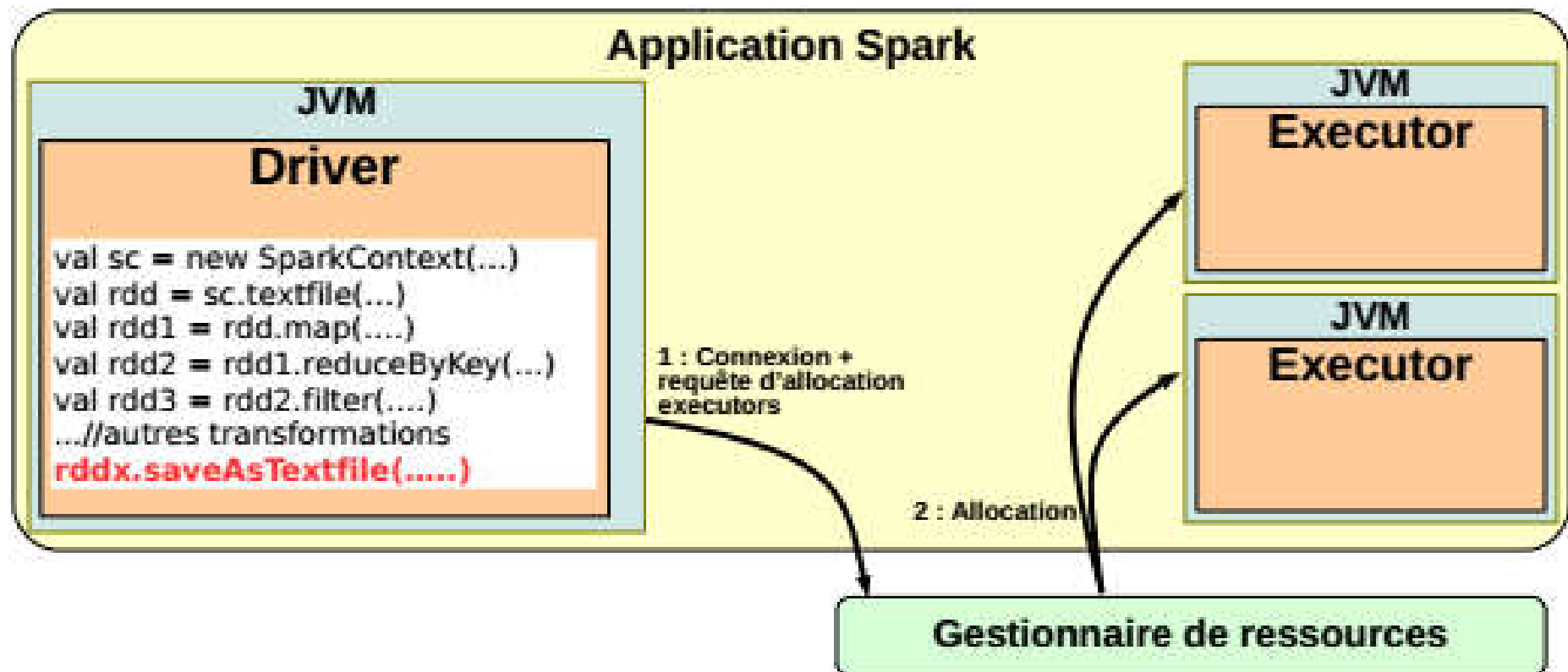


Exécution: Etape 4 (Action)

127

Exécution d'un Job Spark (1ère partie)

- Connexion au gestionnaire de ressources
- Requête d'allocation des executors sur l'infrastructure

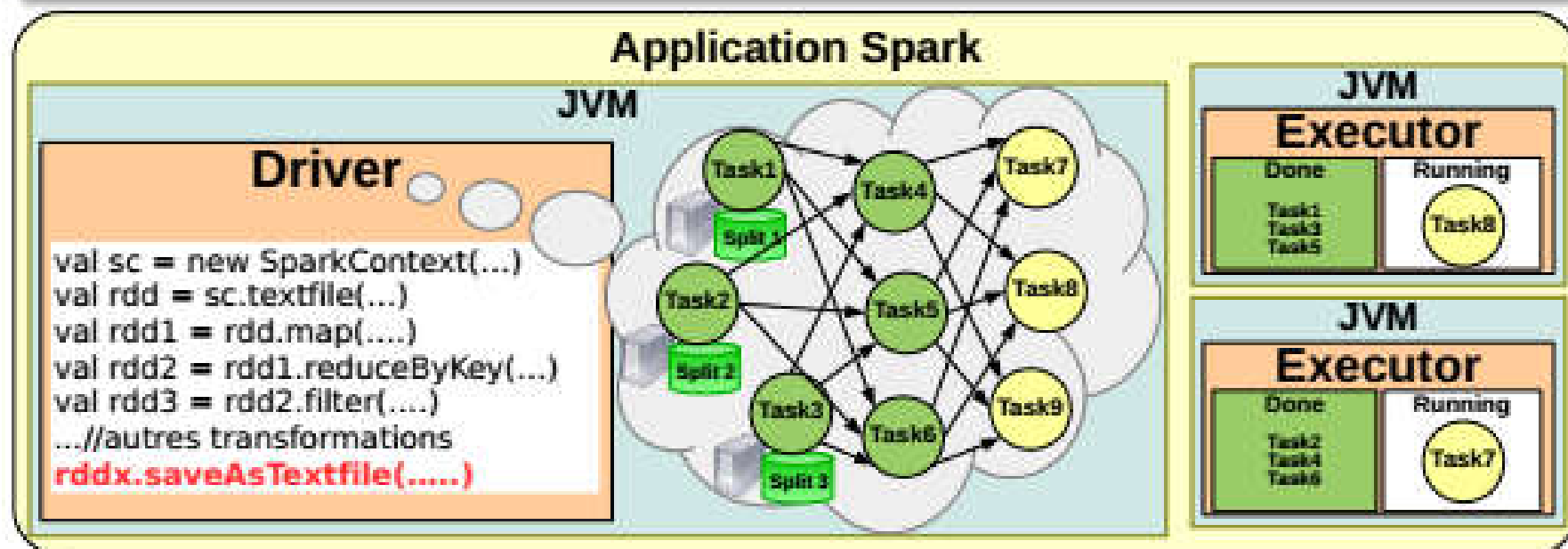


Exécution: Etape 4 (Action)

128

Exécution d'un Job Spark (2ème partie)

- Attribution des tâches aux executors et surveillance de l'avancement
- Une tâche est déployée :
 - si l'ensemble de ses tâches parentes ont terminé leur calcul
 - si elle a été défaillante (redéploiement)
- Les données sont éventuellement mise en cache ou persister sur le executor (persistance des RDD)



Exécution: Etape 4 (Action)

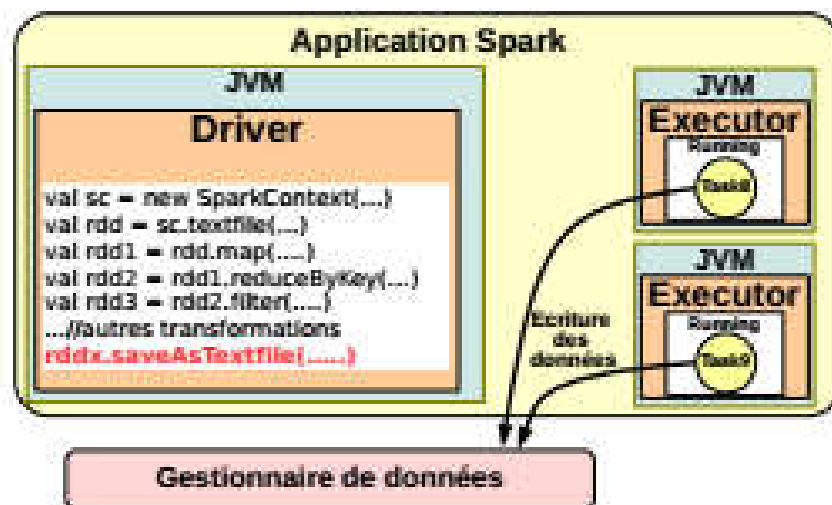
129

Exécution d'un Job Spark (3ème partie)

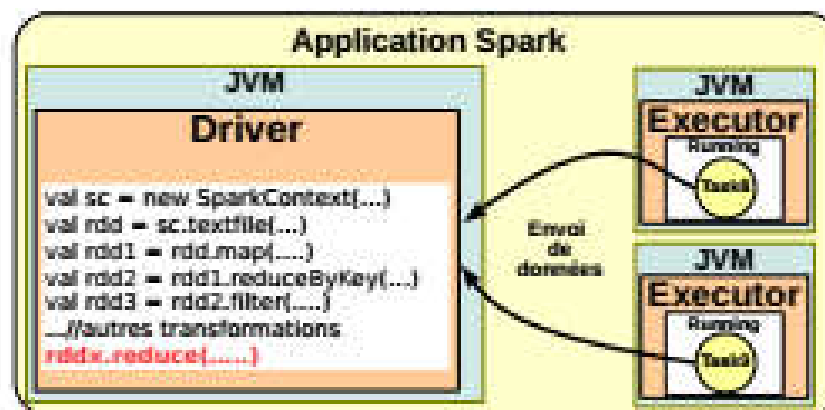
Les données finales du flux sont envoyées :

- soit au service de stockage si c'est une action de sauvegarde (`saveAs...`)
- soit au driver si c'est une action retournant un résultat (`reduce`, etc.)

Action de sauvegarde



Action retournant un résultat



Construction d'un DAG de tâches

130

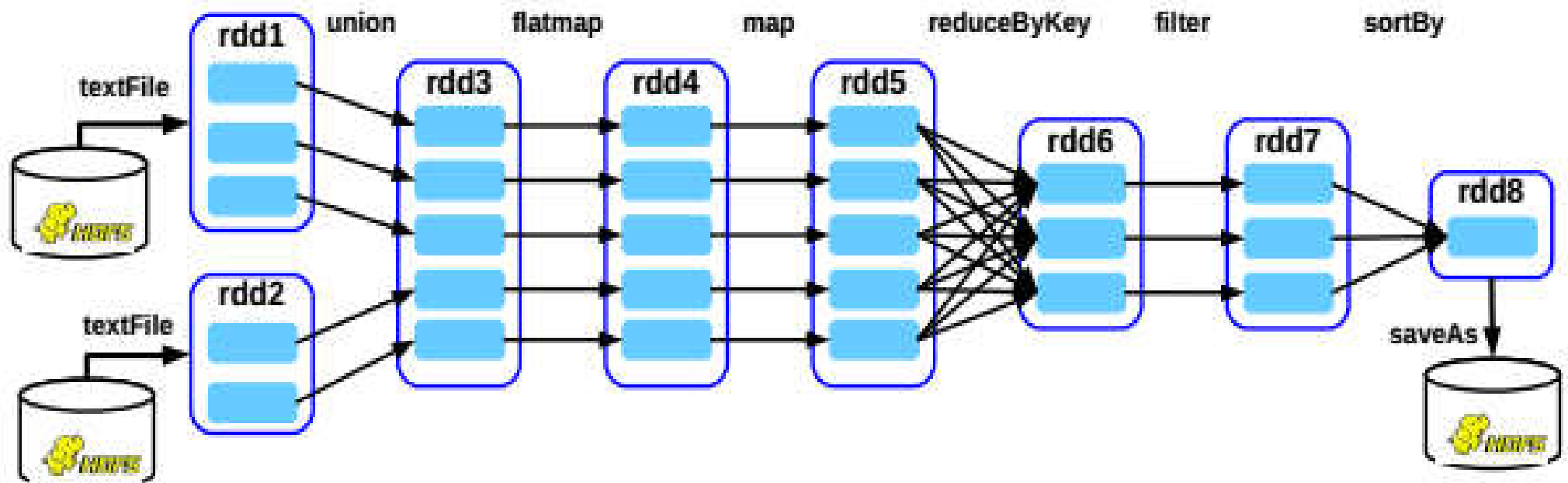
```
val sc = new SparkContext(new SparkConf())
val rdd1= sc.textFile("hdfs://namenode/f1")//implique 3 splits
val rdd2 = sc.textFile("hdfs://namenode/f2")//implique 2 splits
val rdd3 = rdd1.union(rdd2);
val rdd4 = rdd3.flatMap(_.split("_"))
val rdd5 = rdd4.map((_,1))
val rdd6 = rdd5.reduceByKey(_+_ )
val rdd7 = rdd6.filter(_._2 > 1)
val rdd8 = rdd7.sortBy(_._2, true, 1)
rdd8.saveAsTextFile("hdfs://namenode/out")
```

Comment transformer ce programme en graphe de tâches ?

Construction d'un DAG de tâches

131

```
val rdd1= sc.textFile("hdfs://namenode/f1")
val rdd2 = sc.textFile("hdfs://namenode/f2")
val rdd3 = rdd1.union(rdd2);
val rdd4 = rdd3.flatMap(_._split(" "))
val rdd5 = rdd4.map(_._1)
val rdd6 = rdd5.reduceByKey(_+_ , 3)
val rdd7 = rdd6.filter(_._2 > 1)
val rdd8 = rdd7.sortBy(_._2, true, 1)
rdd8.saveAsTextFile("hdfs://namenode/out")
```

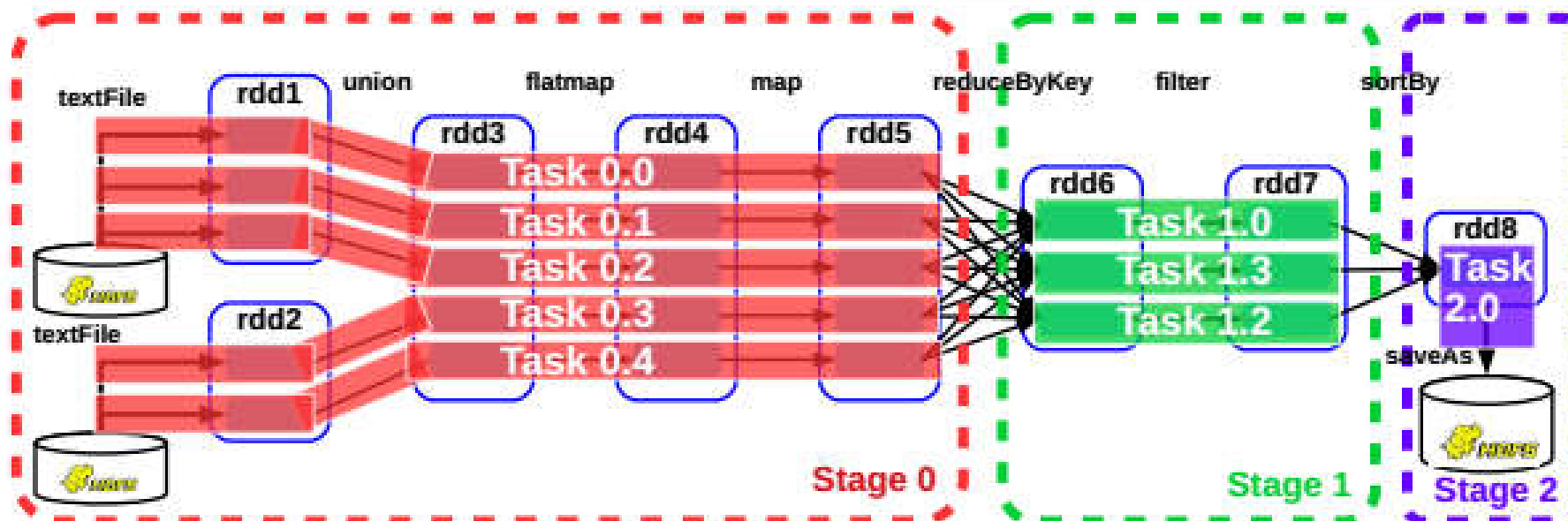


Construction d'un DAG de tâches

132

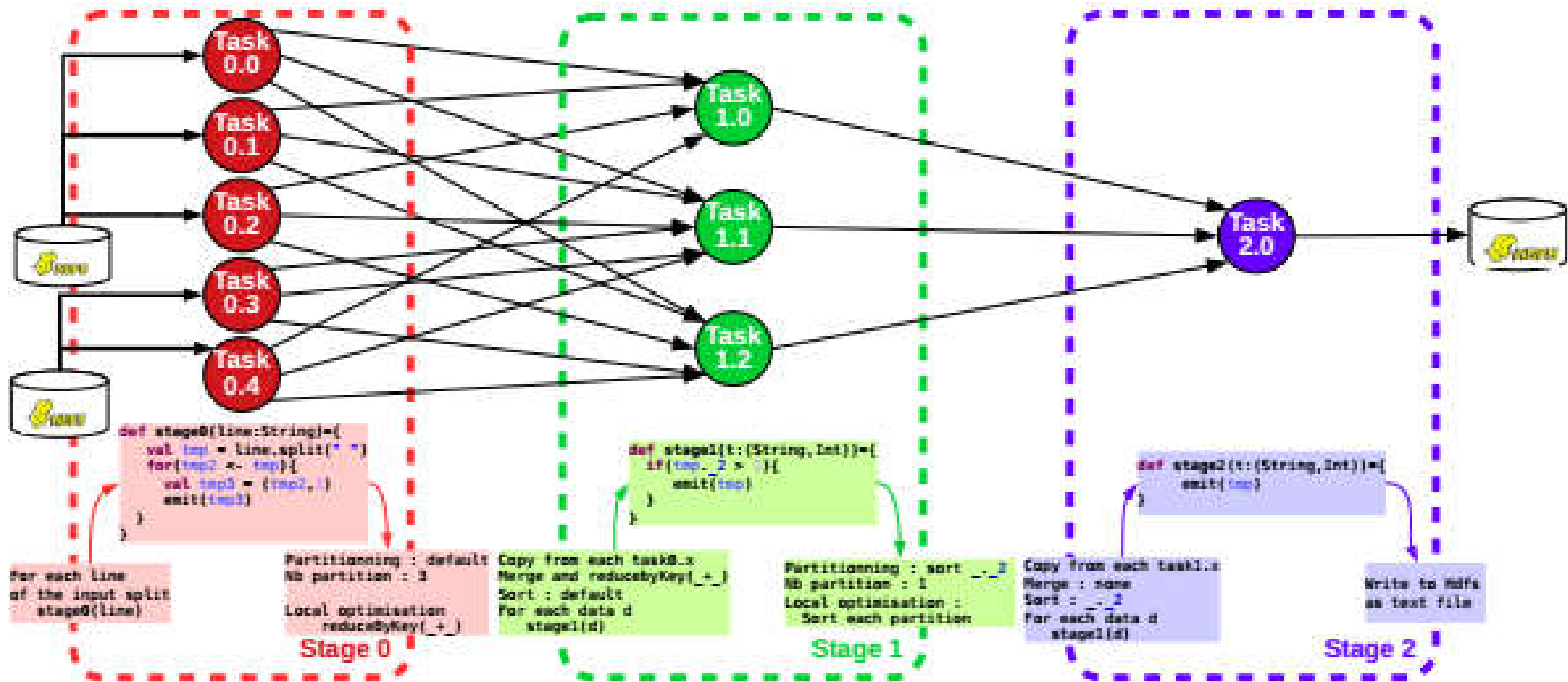
Définition

- Enchaînement continu de transformations simples de RDD compris entre
 - soit deux transformations shuffle
 - soit une transformation de shuffle et une action
- Définit un ensemble de tâches indépendantes et parallèle exécutant le même code mais sur des partitions différentes



Construction d'un DAG de tâches

133



Récapitulatif

134

- ❑ Apache Spark Rapide:
 - ❑ 10 fois plus rapide que Hadoop sur disque
 - ❑ 100 fois plus rapide en mémoire que Hadoop
- ❑ Facile à développer
 - ❑ Riche en terme d'opérations
 - ❑ Ecriture rapide des programmes
 - ❑ Mode interactif
- ❑ Déploiement flexible : Yarn, Standalone, Local, Mesos
- ❑ Stockage : HDFS, S3, Openstack Swift, MapR FS, Cassandra
- ❑ Modèle de développement unifié : Batch, streaming, interactif
- ❑ Multi-langages : Scala, Java, Python, R
- ❑ Plusieurs API: Streaming, MLIB, Graphix, SQL.

135

FIN