IUT de Paris - Rives de Seine
Université de Paris

Traitement de données massives avec

Apache Spark

Master Informatique, 1ère année

Mourad.Ouziri@u-paris.fr

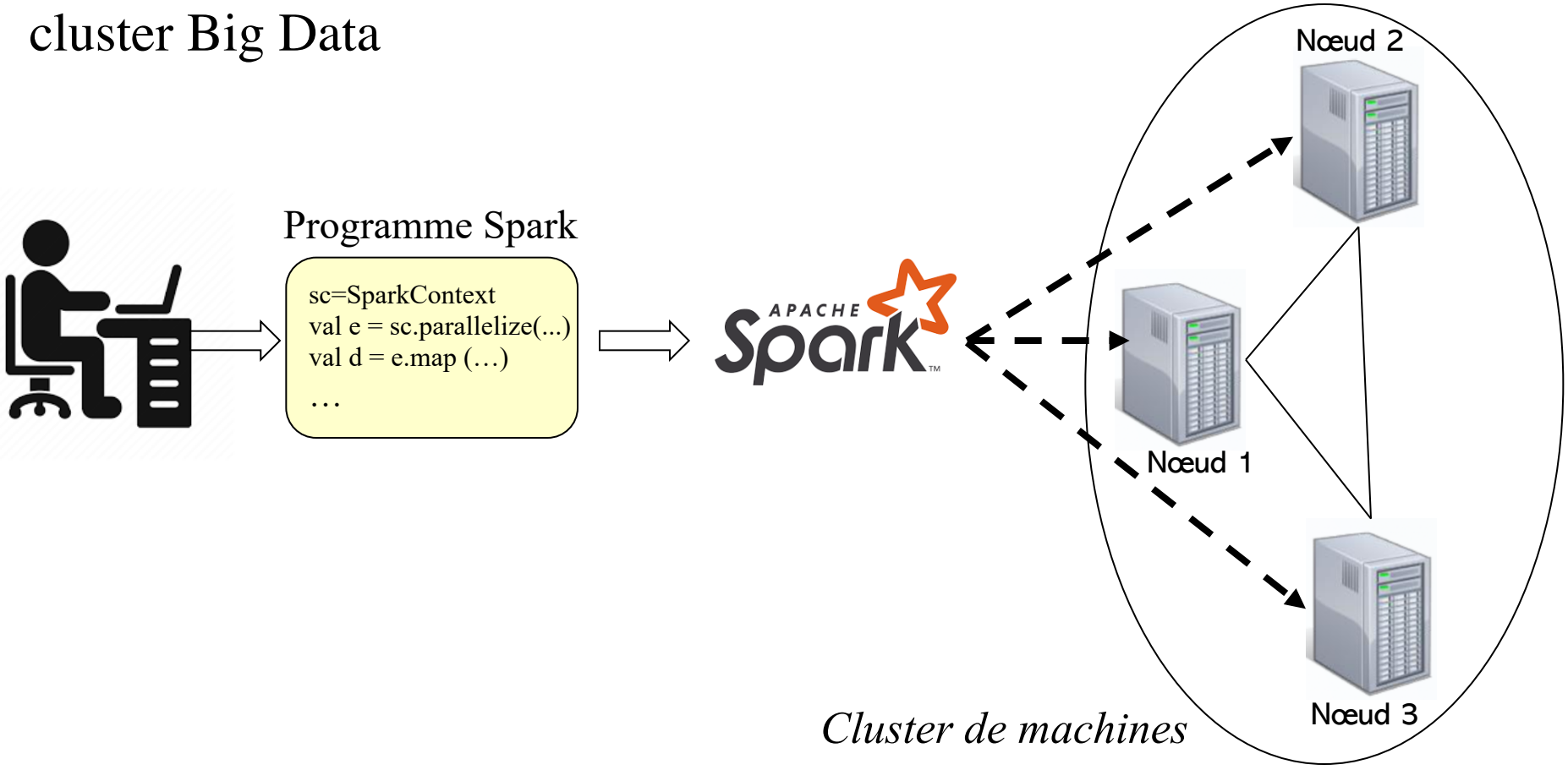
Traitement de données massives avec

Apache Spark *Core*

Apache Spark

Définition

- Spark est un framework de développement et d'exécution de programmes de traitement de données sur plusieurs machines d'un cluster Big Data



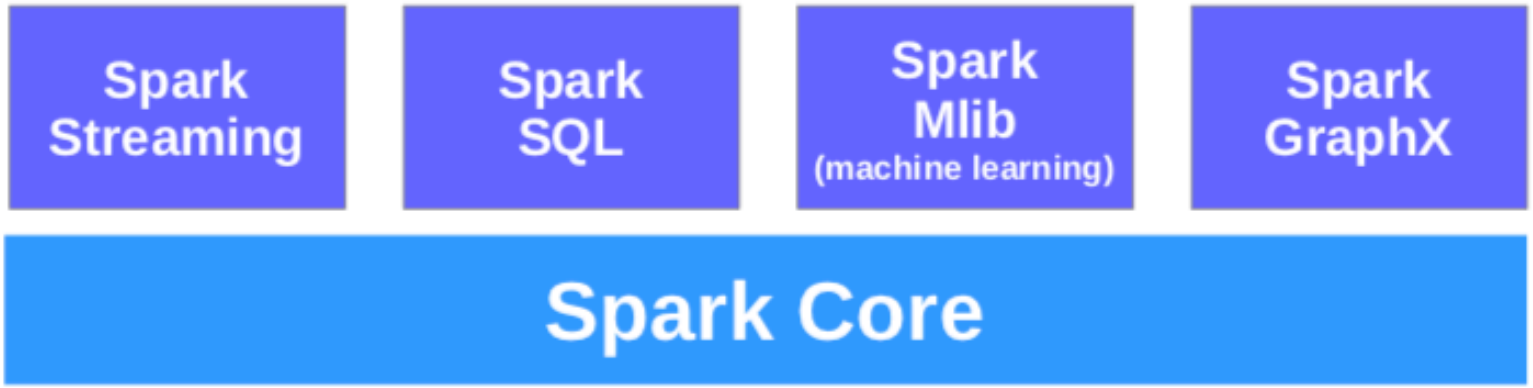
Apache Spark
Définition

- Spark fournit au programmeur :
  - Structure de données (tableau) abstraite pour accéder aux données distribuées sur plusieurs machines du cluster Spark (abstraction de la distribution des données)
  - Ensemble d'opérations applicables sur la structure de données abstraite pour y effectuer des traitements de manière distribuée sur plusieurs machines (abstraction de la distribution des traitements)
  - Framework permettant de gérer l'infrastructure : lancement des programmes, suivi d'exécution, récupération des résultats, gestion des pannes

Apache Spark

Définition

- Il inclut différentes librairies pour le traitement et l'analyse de données en modes batch et en streaming (temps réel)
- Spark unifie la multiplicité et la diversité des frameworks de big data



Apache Spark

Définition

☞ Diversité et hétéroénéité de outils de big data…

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

Apache Spark

Avantages par rapport à Hadoop/MapReduce

- Spark améliore le paradigme MapReduce par :
  - Une API de traitement de données plus riche (que *map()* et *reduce()*)

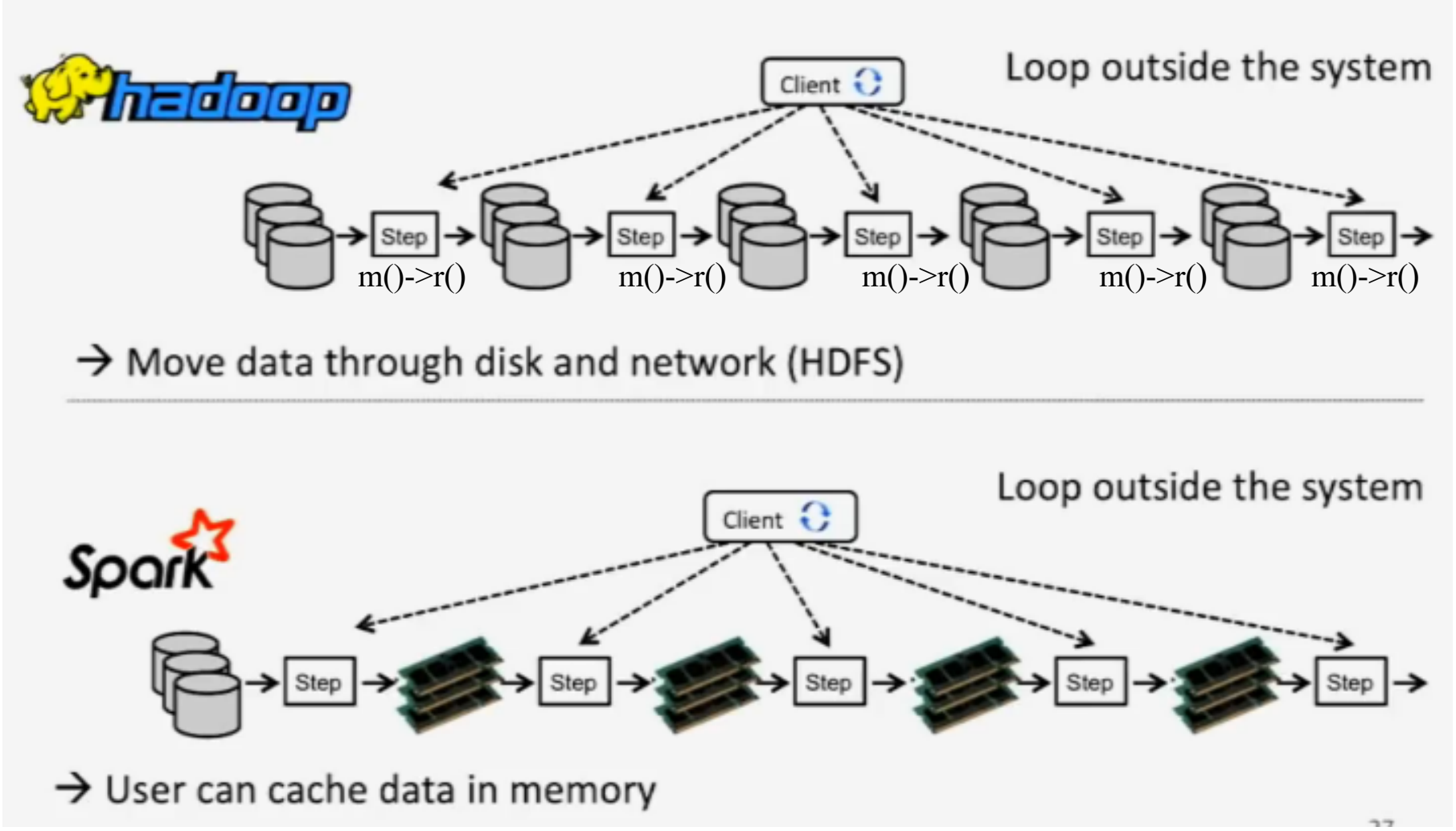
map(func), flatMap(func), filter(func), grouByKey(), reduceByKey(func), mapValues(func), distinct(), sortByKey(func) join(other), union(other), …	reduce(func), collect(), first(), take(), foreach(func), count(), countByKey(), saveAsTextFile(), etc.

  - Les données/résultats en sortie de chaque opération sont maintenues en mémoire (RAM) dans machines du cluster pour être traités par les opérations suivantes
  - Permet de programmer en Java, Scala, Python, R

Apache Spark *vs.* MapReduce

Partage de données dans une application

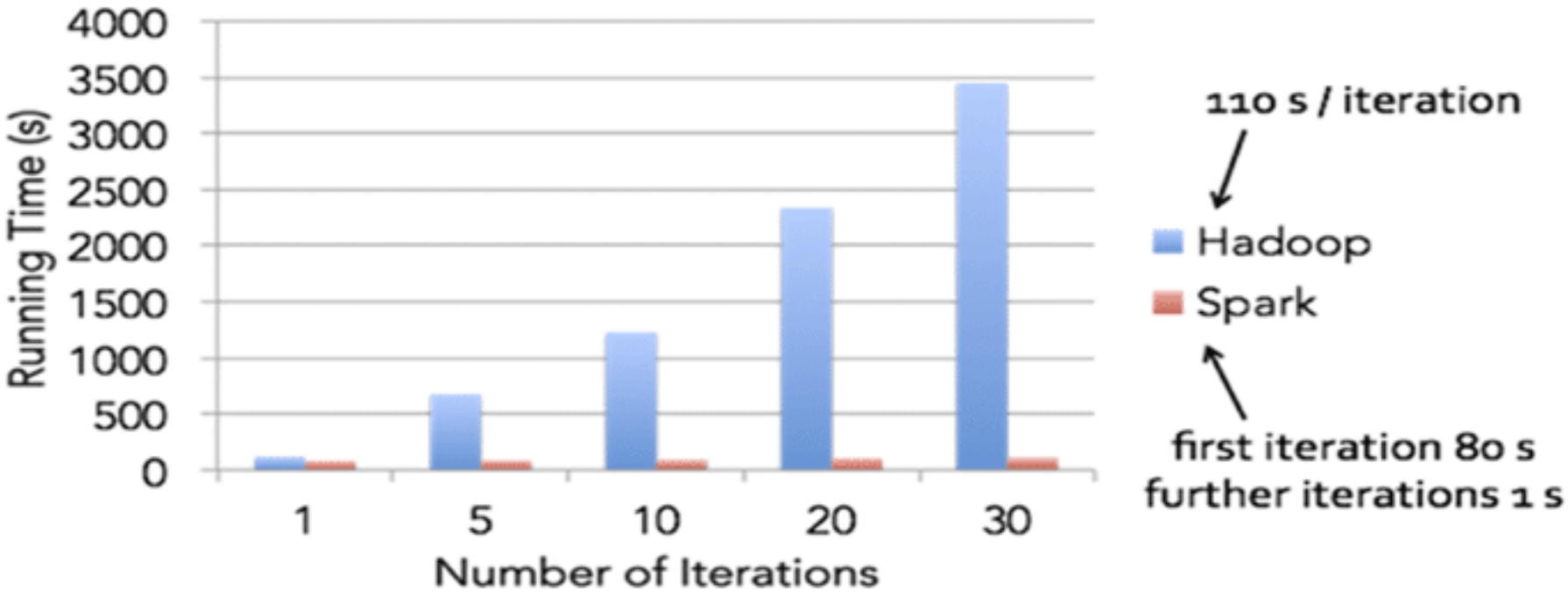
☞ Spark est un framework *in-memory* (contrairement à Hadoop-MapReduce)



Apache Spark

Performances

- Il peut être jusqu'à 100 fois plus rapide que Hadoop-MapReduce dans certaines situations



Apache Spark *vs.* MapReduce

API riche et programmation simplifiée

Son API de haut niveau permet de simplifier le code des applications Big Data

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Spark

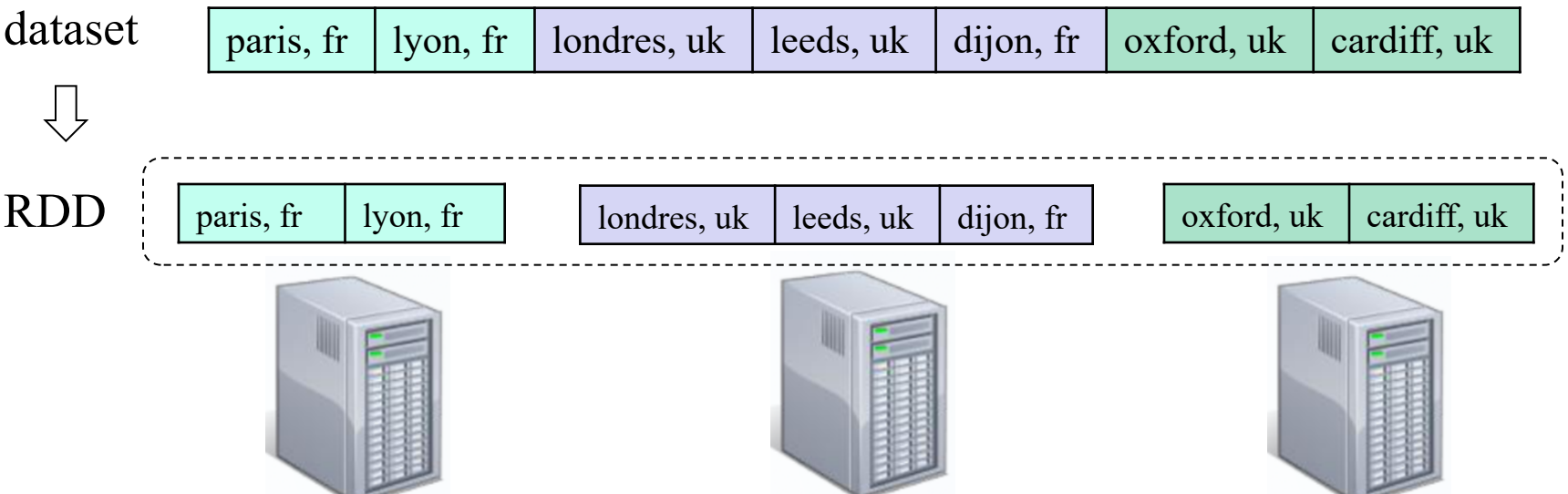
API de programmation de Spark *Core*

- L'API de programmation de Spark *Core* :
  - RDD : structure (collection) de données abstraite distribuée sur les RAM des machines du cluster
  - Opérations : traitements applicables (de manières distribuée et parallèle) sur les partitions d'une RDD

Spark

API de programmation de Spark *Core*

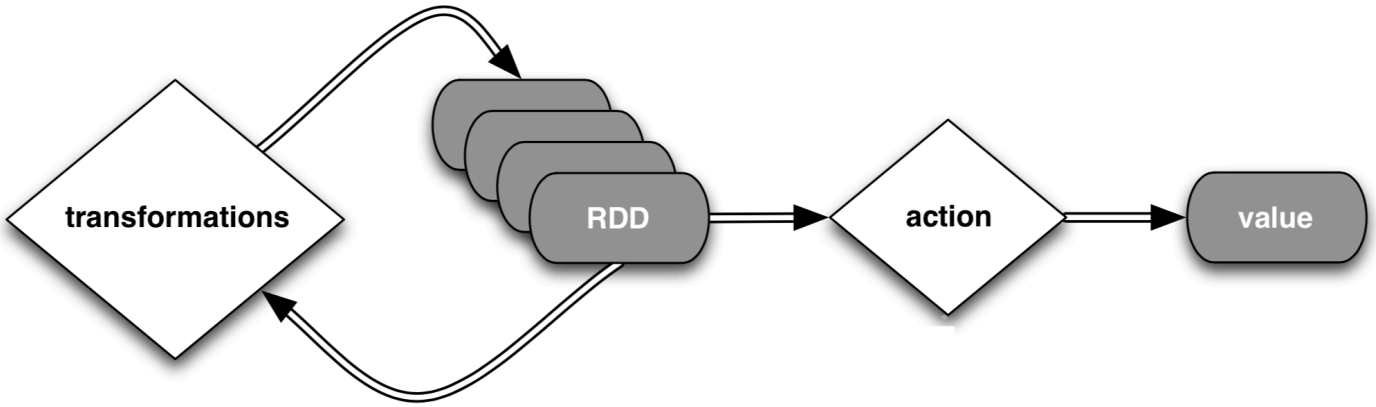
- Resilient Distributed Dataset (RDD) :
  - *Dataset* : collection (tableau) de données, typée, ordonnée et immuable
  - *Distributed* : distribuée sur plusieurs machines avec un niveau de persistance (RAM, cache, disque)
  - *Resilient* (résistante aux pannes machines) : recalculable en cas de défaillance entrainant la perte (totale ou une partie) de la collection



Spark

API de programmation

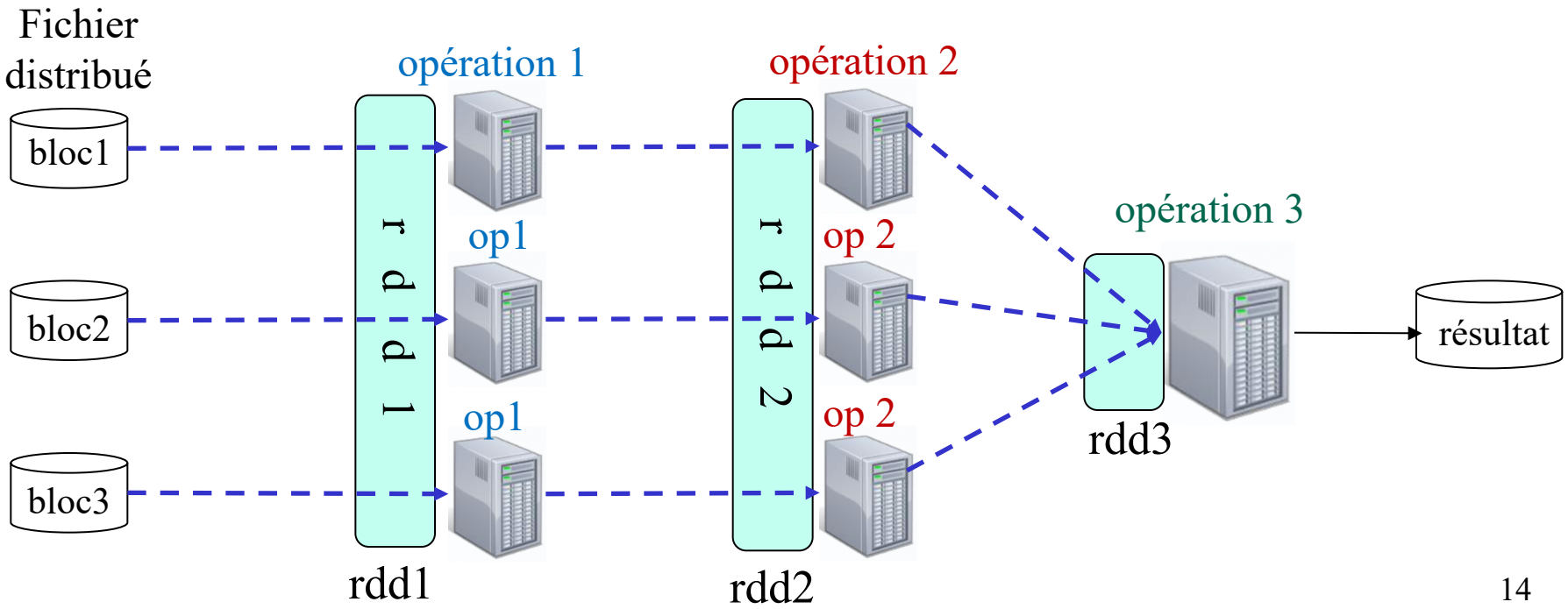
- Opérations sur RDD
  - Ensemble d'opérations exécutées sur les RDD de manière distribuée
  - Elles sont de deux types : transformations et actions



Spark

Structure d'un programme

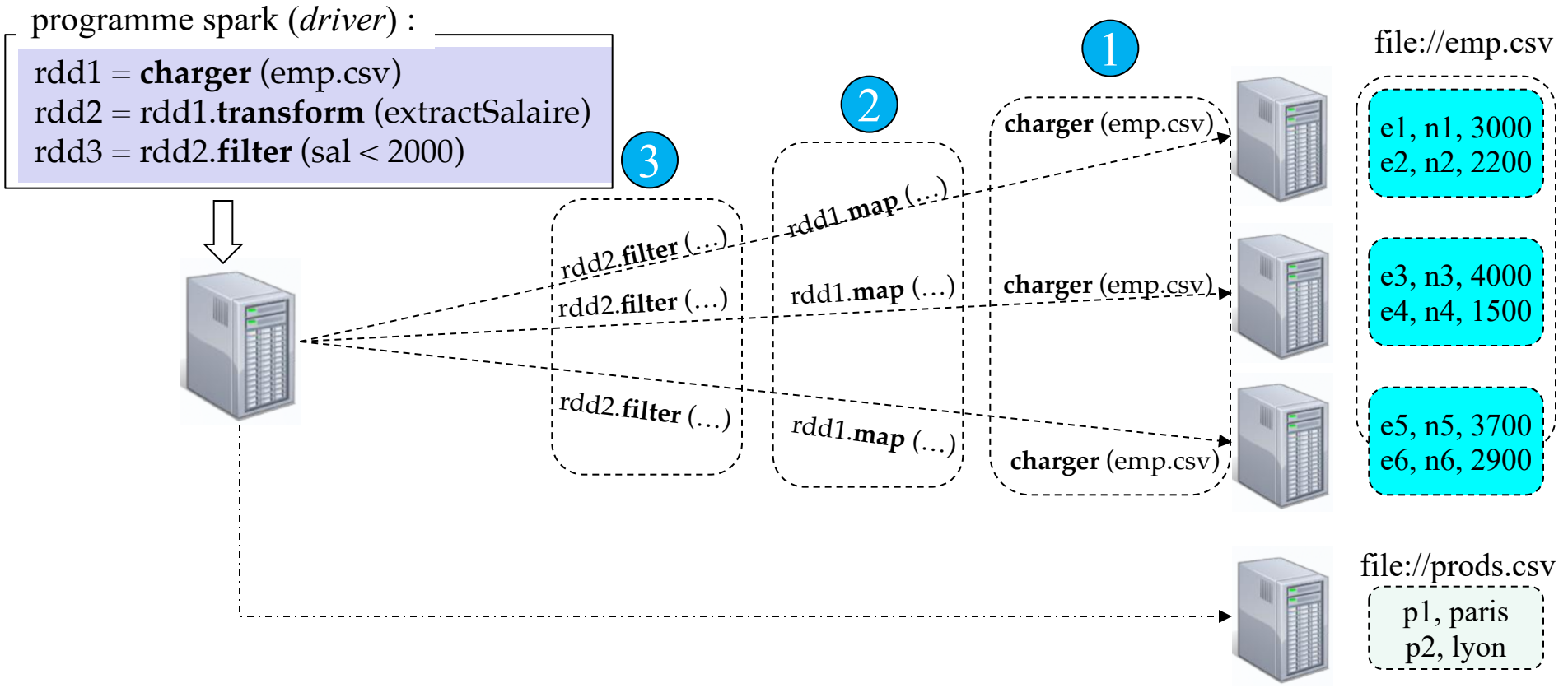
- Schéma de traitement de données en Spark
  - Les données sont chargées sous forme de RDD (collection distribuée)
  - Chaque opération sur RDD est appliquée de manière distribuée sur l'ensemble des partitions du RDD concerné



Apache Spark

Principe de distribution des traitements : *data locality*

Data locality : chaque opération est envoyée et exécutée par les machines qui détiennent les partitions de données concernées



Apache Spark

Opérations

☞ Opérations de transformation de RDD

- Elles appliquent une transformation sur les éléments du RDD traité et génèrent le résultat dans un nouveau RDD de sortie
- Elles sont évaluées de manière passive

map(func), flatMap(func),
filter(func), grouByKey(),
reduceByKey(func),
mapValues(func), distinct(),
sortByKey(func)
join(other), union(other), ...

☞ Les actions

- Elles calculent un résultat qui sera renvoyé
  au *Driver* (*main* de l'application Spark)
  ou stocké sur disque
- Elles déclenchent l'évaluation de toutes les transformations passives en attente

reduce(func), collect(), first(),
take(), foreach(func), count(),
countByKey(),
saveAsTextFile(), etc.

Programmation Spark

Structure d'un programme

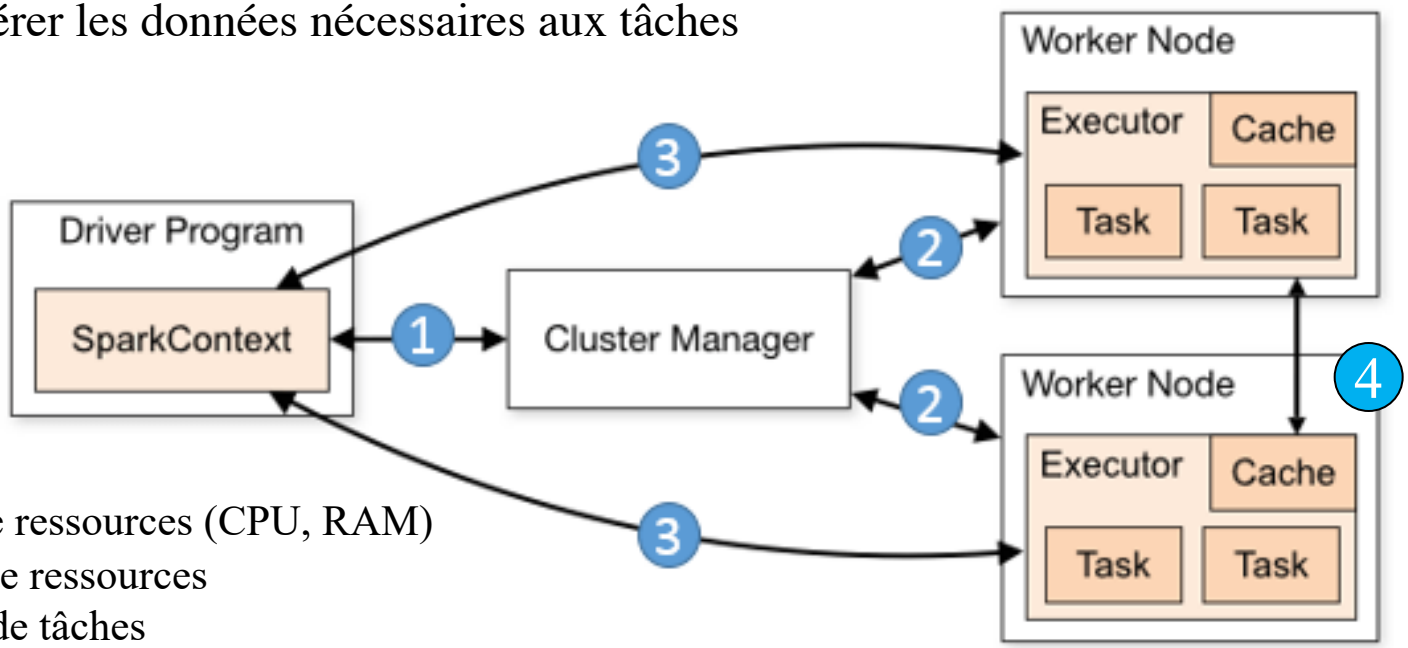
☞ Structure d'un programme Spark : le programme dit *Driver*

1. Etablir la connexion avec le Spark Master : création d'un objet *SparkContext*
2. Charger les données dans les RAM des machines workers : création de RDD
3. Appliquer des opérations de transformation sur les données des RDD : transformations passives
4. Appliquer les actions terminales sur les données des RDD pour calculer et retourner le résultat final (pour affichage à l'écran ou stockage sur disque)

Programmation Spark

Connection à Spark

- Un programme Spark utilise un objet *SparkContext* permettant d'envoyer les opérations/traitements aux noeuds du cluster. *SparkContext* a pour rôle de :
  - Négocier les ressources auprès du cluster (*Master – Resource Manager, Hadoop-Yarn*)
  - Découper le programme en tâches et les envoyer aux *Executors* (processus permettant l'exécution et le suivi de tâches sur les noeuds de calculs et de stockage)
  - Transférer les données nécessaires aux tâches



1. Demande de ressources (CPU, RAM)
2. Allocation de ressources
3. Attribution de tâches
4. Echange de données

Programmation Spark

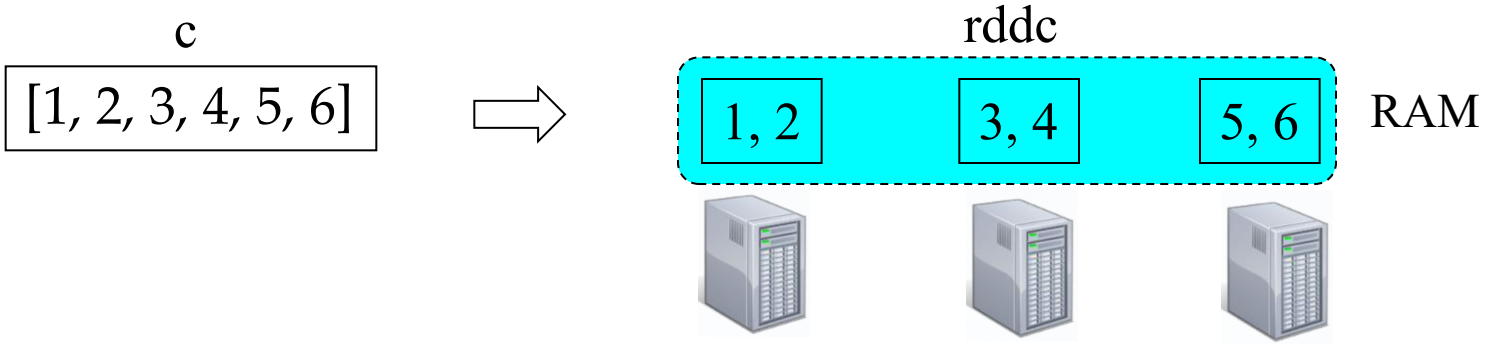
Opérations de transformation

- Création de RDD
  - Par distribution d'une collection/tableau de données (déjà en mémoire)

```
def parallelize[T](seq: Seq[T], nbPartitions: Int): RDD[T]
```

  - Exemple

```
var c = List (1, 2, 3, 4, 5, 6)
val rddc = sc.parallelize (c, 3)
```



Programmation Spark

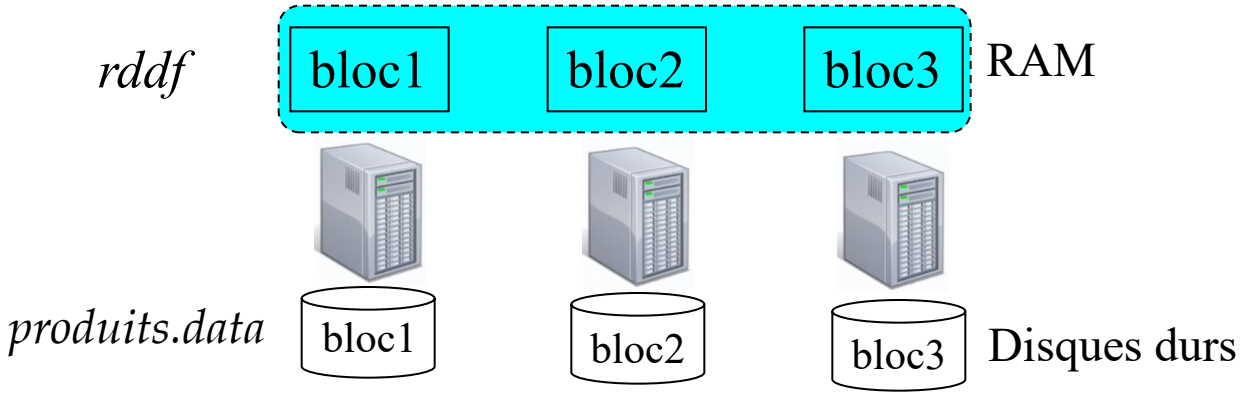
Opérations de transformation

- Création de RDD
  - Parallélisation d'une collection en mémoire

```
def textFile(path: String, minPartitions: Int = 2): RDD[String]
```

  - Exemple

```
var fichier = "file:///data/produits.data"
val rddf = sc.textFile (fichier)
```

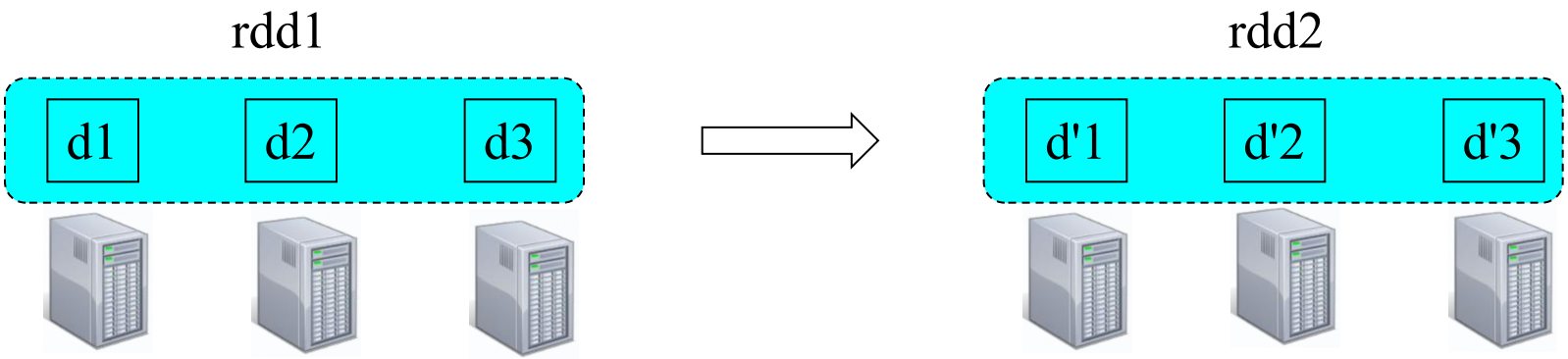


Programmation Spark

Opérations de transformation

- Opération de type *transformation* de RDD
  - Génère une nouvelle RDD par application d'une fonction de transformation des éléments d'une RDD
  - Exécutée de manière passive : n'est exécutée que si la RDD générée est requise par une action terminale

rdd2 = **rdd1**.**spark_tranform_funct** (**user_funct**)



Programmation Spark

Opérations de transformations

☞ Filtrage de données : sélectionner seulement les éléments du RDD vérifiant une certaine condition donnée par une fonction utilisateur

```
def filter (f: (T) => Boolean) : RDD[T]
```

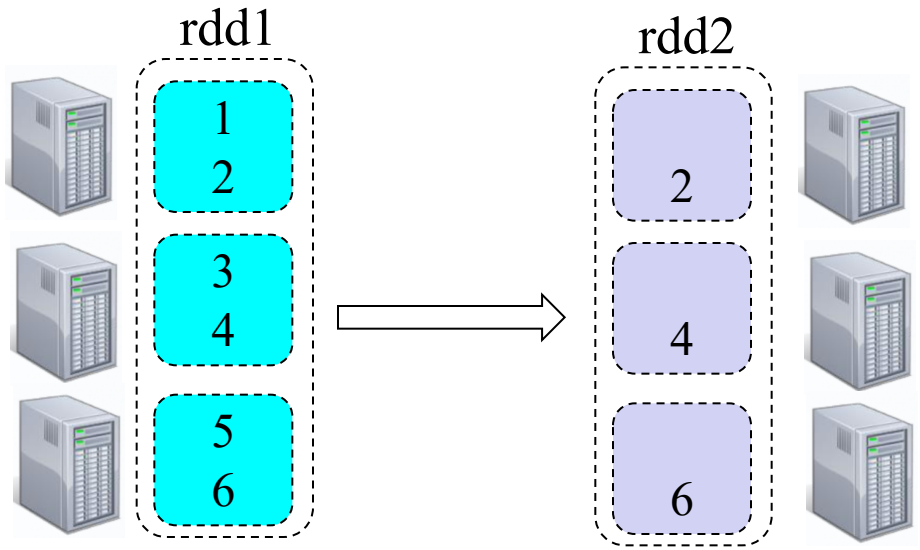
☞ Exemple :

1) Avec une lambda expression :

```
rdd2 = rdd1.filter (e => e%2 == 0)
```

2) Avec une fonction nommée :

```
def pair (x:Int) : Boolean = {
        return x % 2 == 0;
}
rdd2 = rdd1.filter (pair)
rdd2 = rdd1.filter (e => pair(e))
```



Programmation Spark

Opérations de transformations

☞ Transformation de données : transforme les éléments du RDD par application d'une fonction sur chacun de ces élémennts

```
def map (f: (T) => U ) : RDD[U]
```

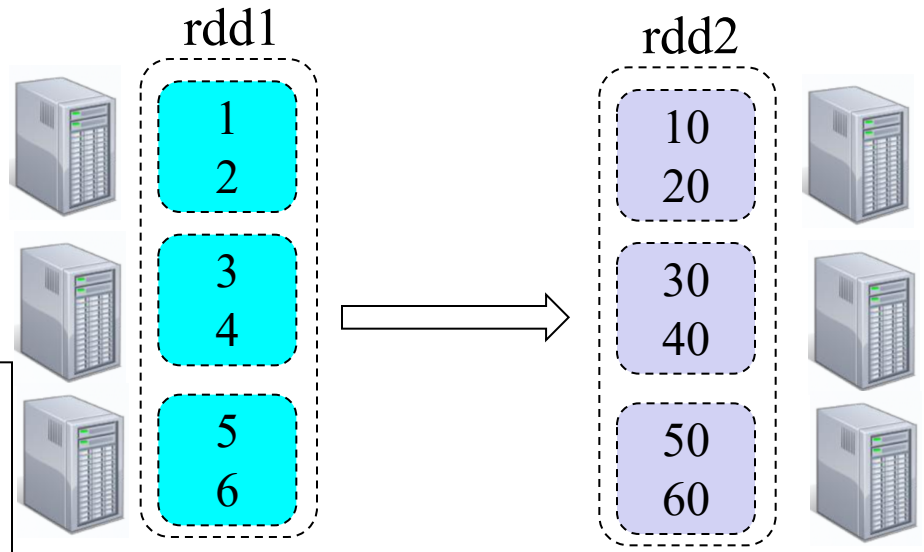
☞ Exemple :

1) Avec une lambda expression :

```
rdd2 = rdd1.map (e => e*10)
```

2) Avec une fonction nommée :

```
def multiplier (x:Int, f:Int) : Int = {
        return x * f;
}
rdd2 = rdd1.map (e => multiplier(e, 10))
rdd2 = rdd1.map (multiplier(_, 10))
```

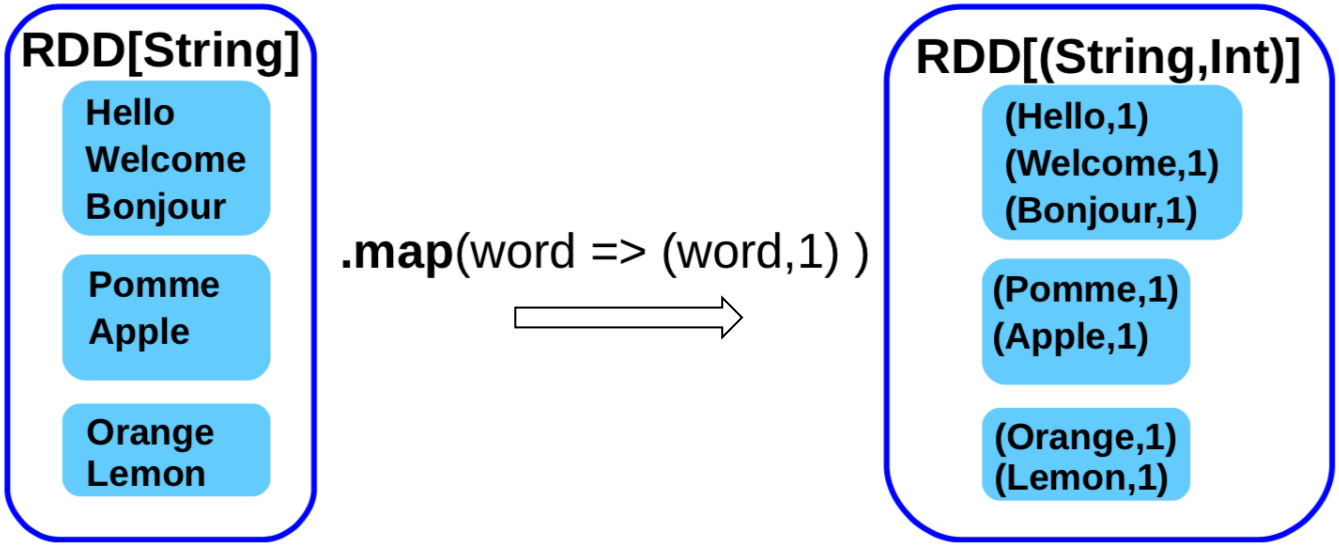


Programmation Spark

Opérations de transformations

☞ Transformation de données : transforme les éléments du RDD par application d'une fonction sur chacun de ces élémennts

```
def map (f: (T) => U ) : RDD[U]
```

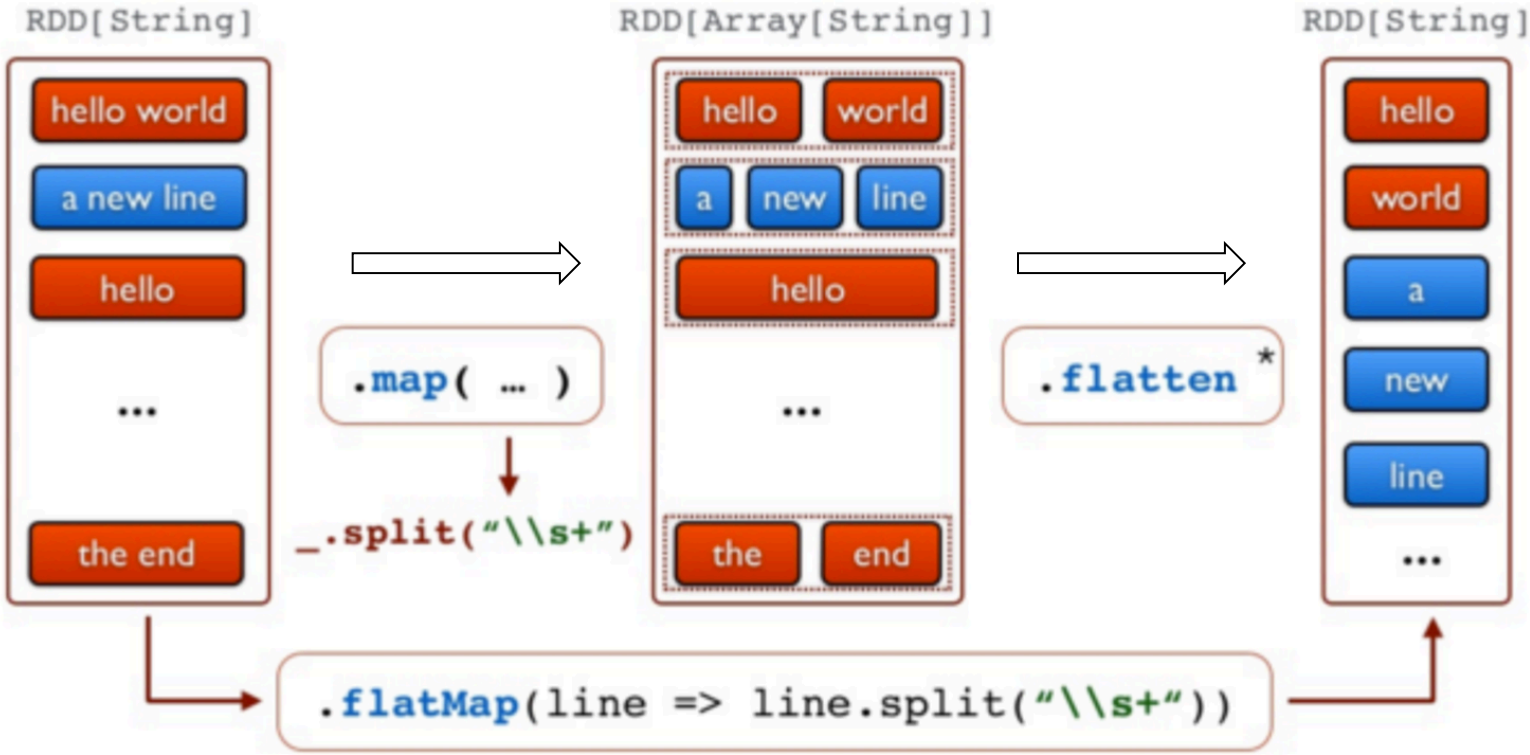


Programmation Spark

Opérations de transformations

☞ Flatmap : crée de nouveaux éléments du RDD en appliquant une fonction des éléments du RDD d'entrée

```
def flatMap[U](f: (T) =>TraversableOnce[U]): RDD[U]
```

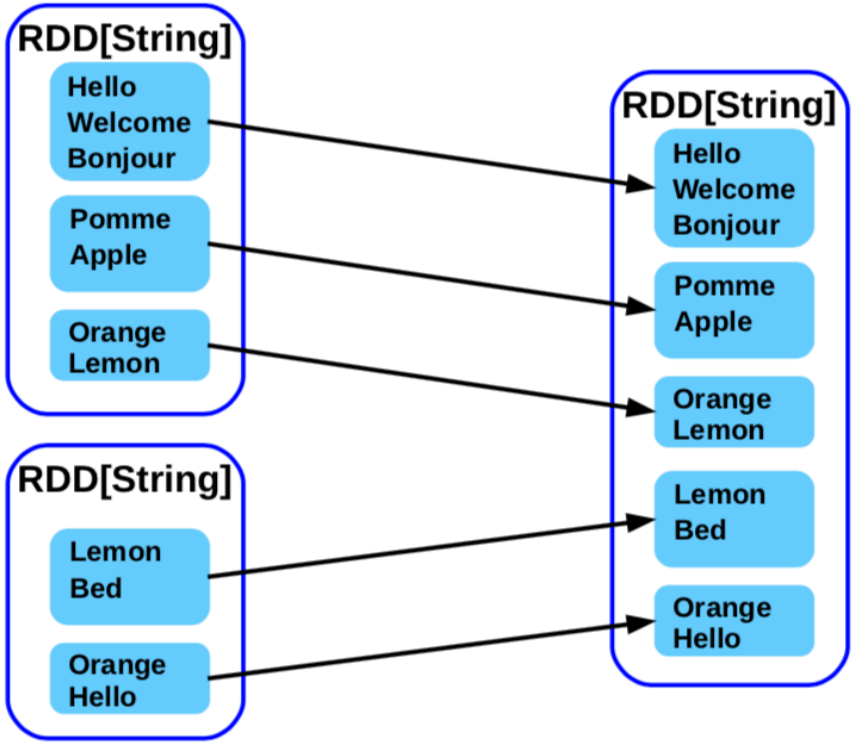


Programmation Spark

Opérations de transformations

☞ Union de RDD : créer un nouvel RDD par union des RDD en entrée en conservant les doublons

```
def union (other: RDD[T]): RDD[T]
```

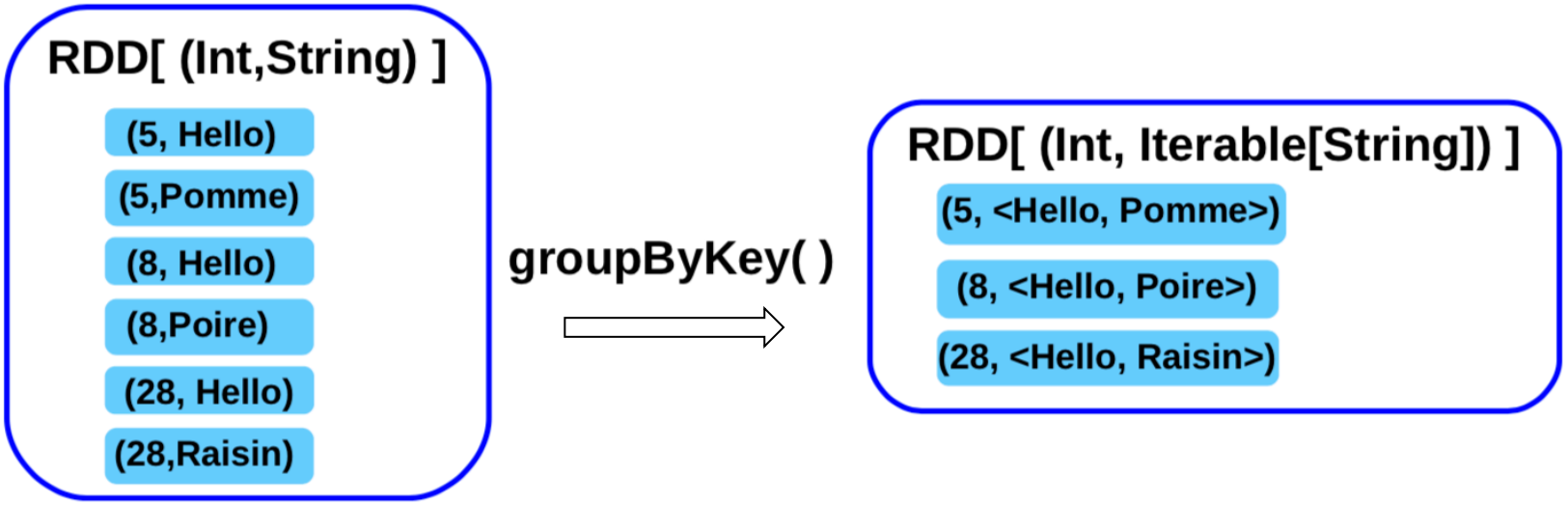


Programmation Spark

Opérations de transformations

☞ Regroupement de RDD par clé (sorte d'auto jointure) : regrouper les éléments d'une RDD structurées en (clé,valeur) ayant la même clé

```
def groupByKey(): RDD[(K, Iterable[V])]
def groupByKey(numPartitions:Int): RDD[(K, Iterable[V])]
```

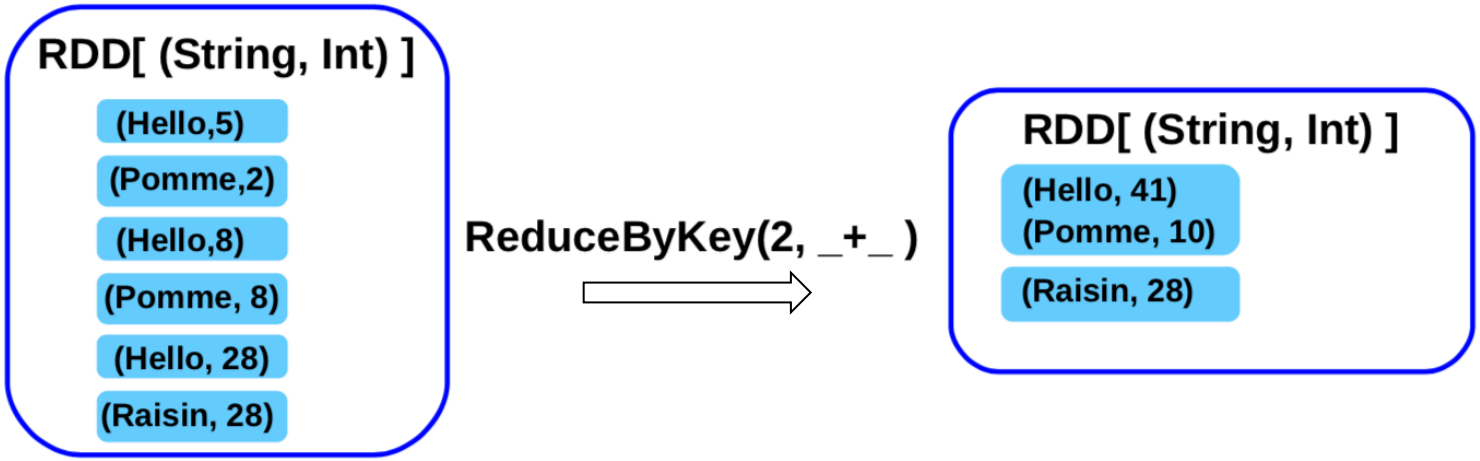


Programmation Spark

Opérations de transformations

☞ Agrégation de RDD par clé : agréger les valeurs d'une même clé d'une RDD de type (clé, valeur) avec une fonction d'agrégation associative

```
def reduceByKey (func: (V, V) =>V) : RDD[(K, V)]
def reduceByKey (numPartitions:Int, func : (V, V) =>V) : RDD[(K, V)]
```

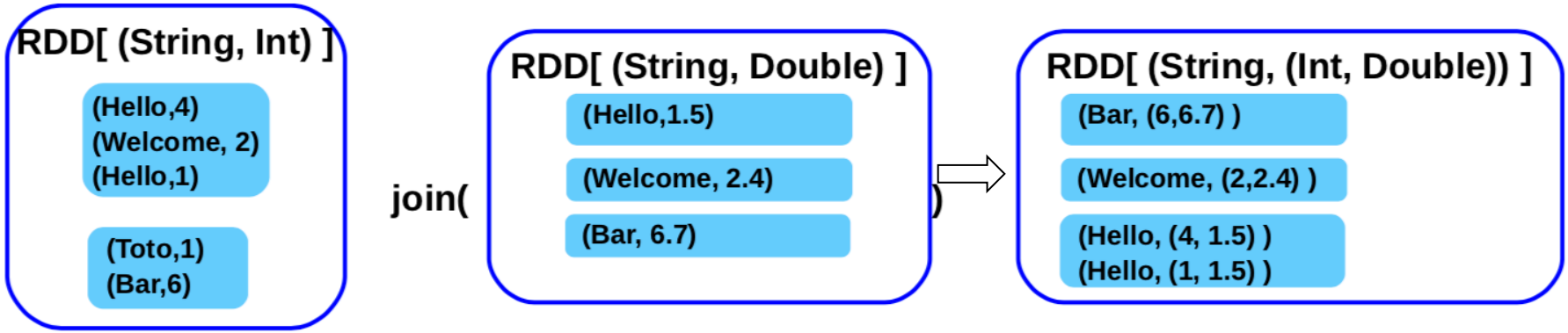


Programmation Spark

Opérations de transformations

☞ Jointure de deux RDD : réalise la jointure des éléments de deux RDD de type (clé, valeur) ayant la même clé

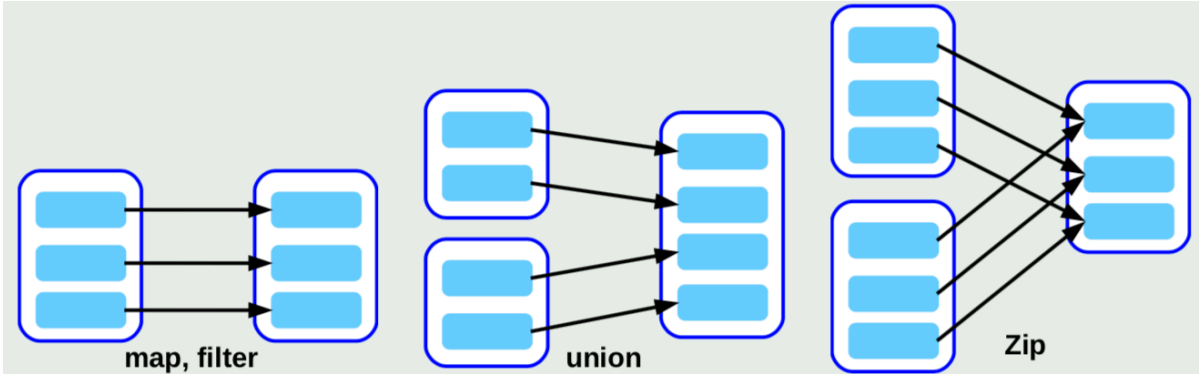
```
def join[W] (other: RDD[(K, W)]) : RDD[(K, (V, W))]
def join[W] (other: RDD[(K, W)], numPartitions : Int) : RDD[(K, (V, W))]
```



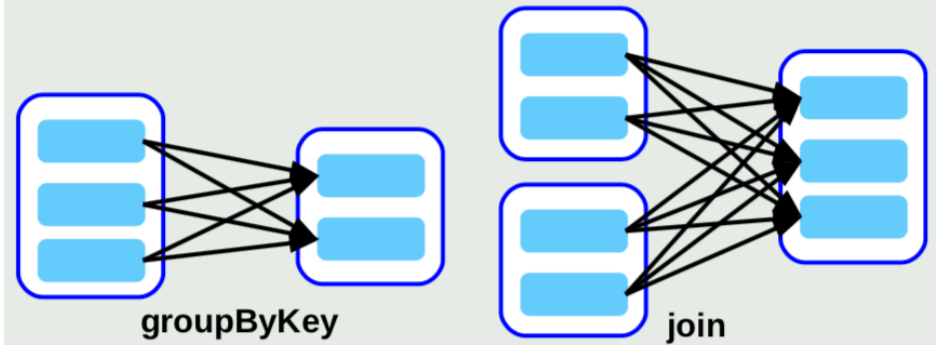
Programmation Spark

Types d'opérations de transformations

- Opérations étroites (peu coûteuses) : chaque partition du RDD résultat est calculée à partir d'une seule partition de chaque RDD source



- Opérations étendues (coûteuses) : chaque partition du RDD résultat est calculée à partir de plusieurs (voire de toutes) partitions des RDD sources

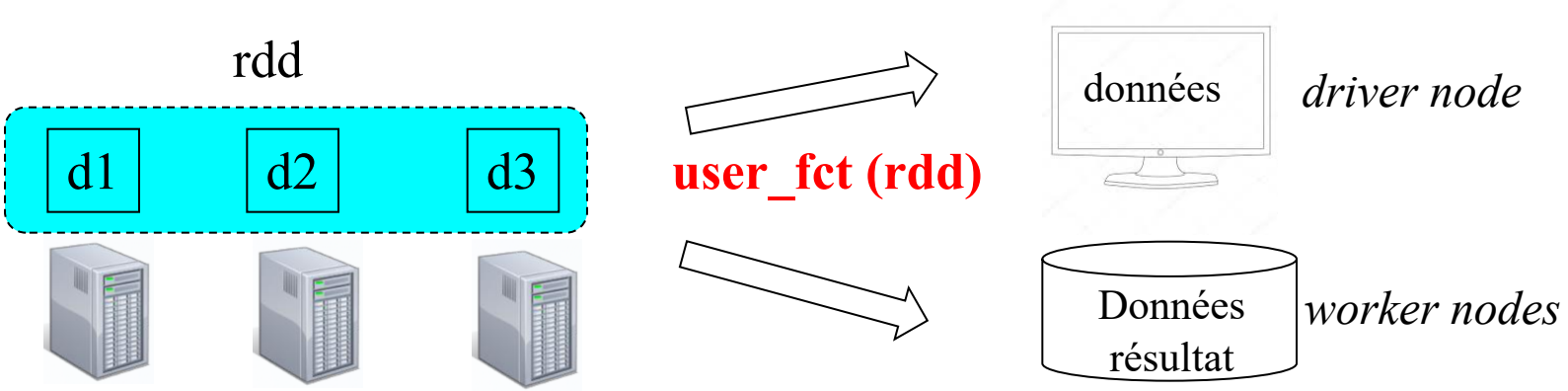


Programmation Spark

Opérations de type *action*

- Opération de type *action* sur les éléments d'une RDD
  - Calcule un résultat à partir des éléments d'une RDD qui sera soit retourné au programme *Driver* soit stocké sur le disque des workers (ou dans une BD)
  - Exécutée de manière active : elle déclenche l'exécution de toutes les transformations passives nécessaires au calcul de la RDD utilisée par l'action

résultat = **rdd**.**spark_action_funct** (**user_funct**)

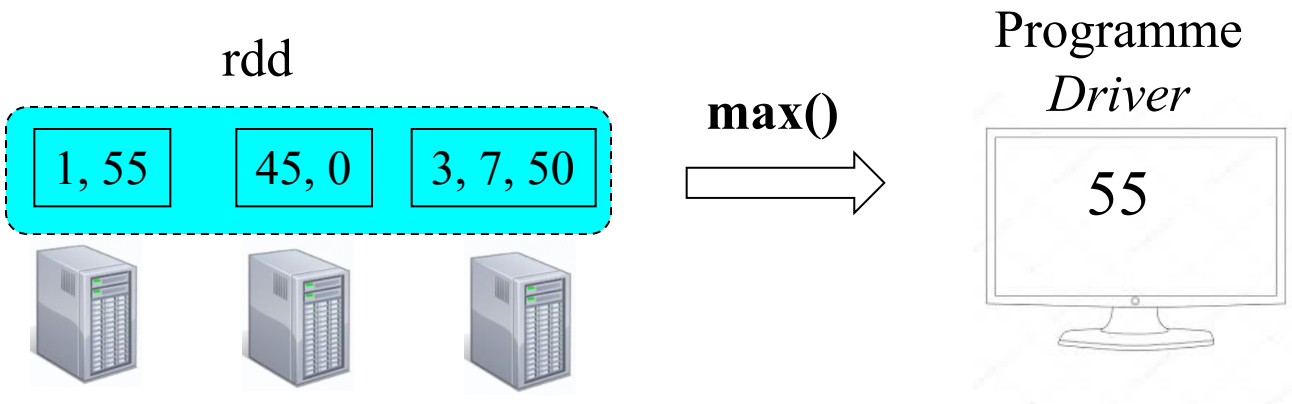


Programmation Spark

Opérations de transformations

☞ Quelques opérations de type *action* sur RDD

```
def max()(implicit ord: Ordering[T]): T
def min()(implicit ord: Ordering[T]): T
def isEmpty(): Boolean
def first(): T
def count():Long
def collect(): Array[T]
def take(num: Int): Array[T]
…
```

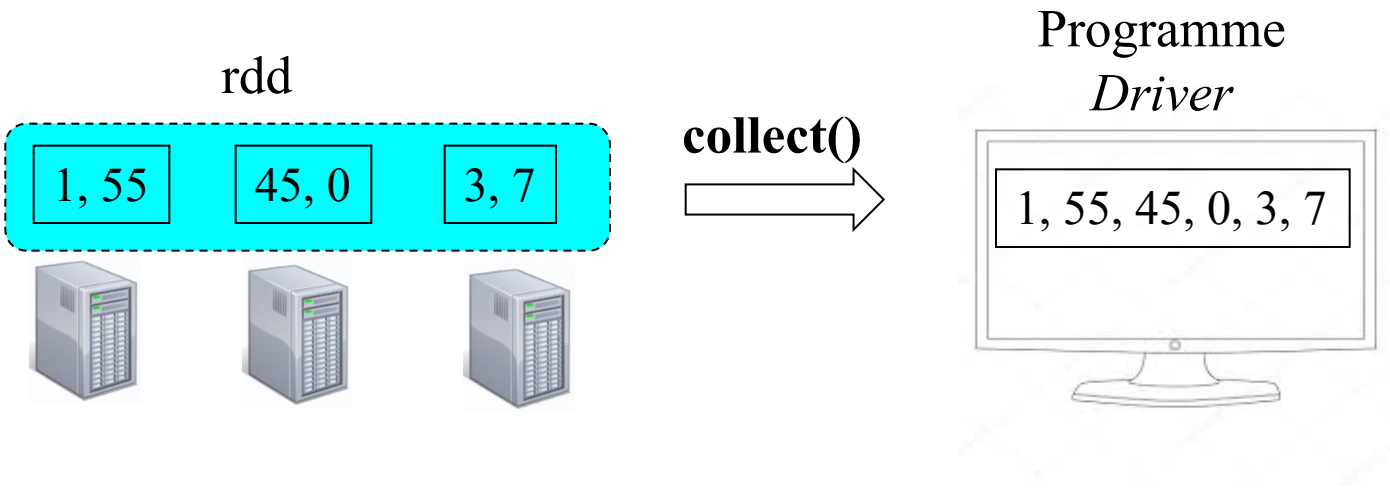


Programmation Spark

Opérations de transformations

- Collecter les éléments d'une RDD partitionnés sur les machies du cluster en seul tableau puis renvoyé au *Driver*

def **collect**(): Array[T]



Programmation Spark

Opérations de transformation

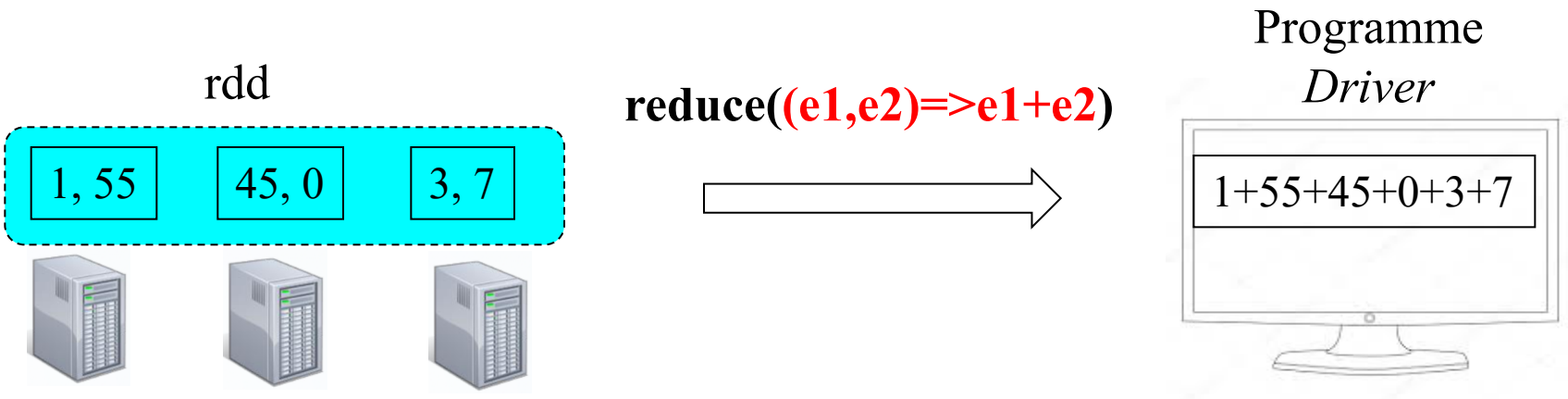
- Appliquer une fonction *f* sur chaque élément d'une RDD sans renvoi de résultat (utilité : affichage, notification, publication web service, etc.)

```
def foreach(user_fct: (T) =>Unit): Unit
```

- Agrégation de tous les éléments d'une RDD

```
def reduce(user_fct: (T, T) =>T): T
```

user_fct : fonction associative

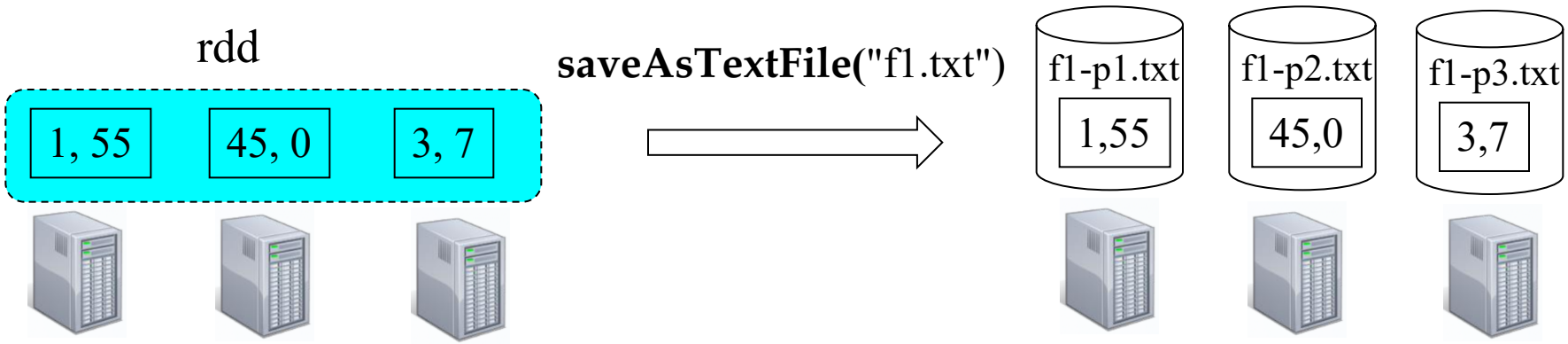


Programmation Spark

Opérations de transformations

☞ Sauvegarder une RDD sur disque (fichier distribué)

def **saveAsTextFile** (path : String): Unit



Programmation Spark

Résumé de quelques opérations d'action et de transformation

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

Apache Spark

Opérations

☞ Exemple de script *WordCount* sur Spark-shell en mode interactif

```
val rdd1 = sc.textFile("file:///data.txt")
val rdd2 = rdd1.flatMap(l => l.split(" "))
val rdd3 = rdd2.map(w => (w, 1))
val rdd4 = rdd3.reduceByKey((v1,v2) => v1+v2)
rdd4.saveAsTextFile("hdfs ://outputs/result.txt")
```

Apache Spark

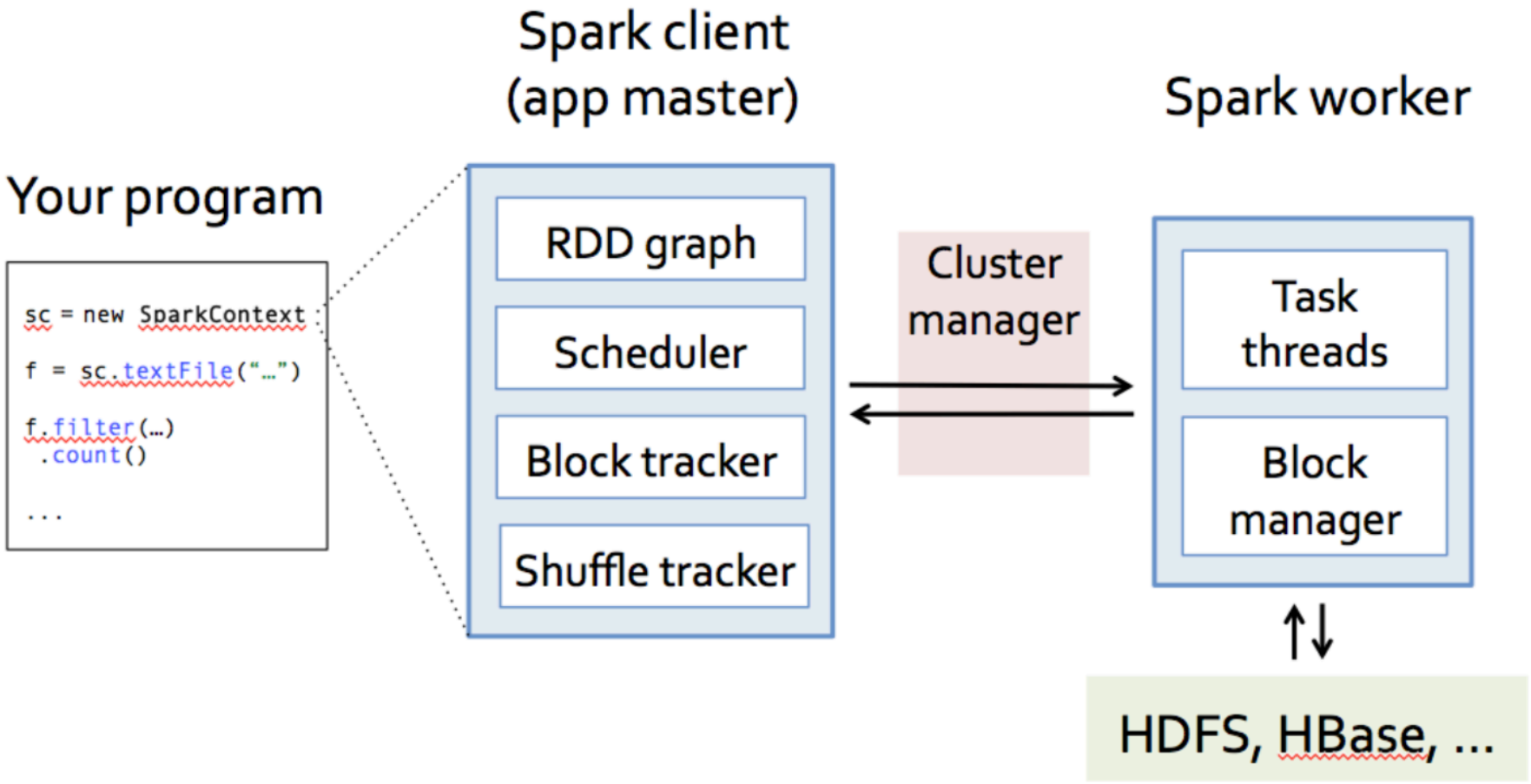
Programmation

☞ Exemple de programme objet en scala de *WordCount*

```
object Programme {
  def main(args: Array[String]):Unit= {
     val conf = new SparkConf().setAppName("Mon␣Programme␣Spark");
     val spark = new SparkContext ( conf )
     val textFile = spark.textFile("hdfs:/ /home/data/data.txt")
     val rdd = textFile.flatMap(line => line.split("␣"))
                              .map(x => (x, 1))
                              .reduceByKey ((v1, v2) => v1 + v2)
     rdd.saveAsTextFile("hdfs :/ /home/data/outputs/resf1.txt")
} }
```

Apache Spark

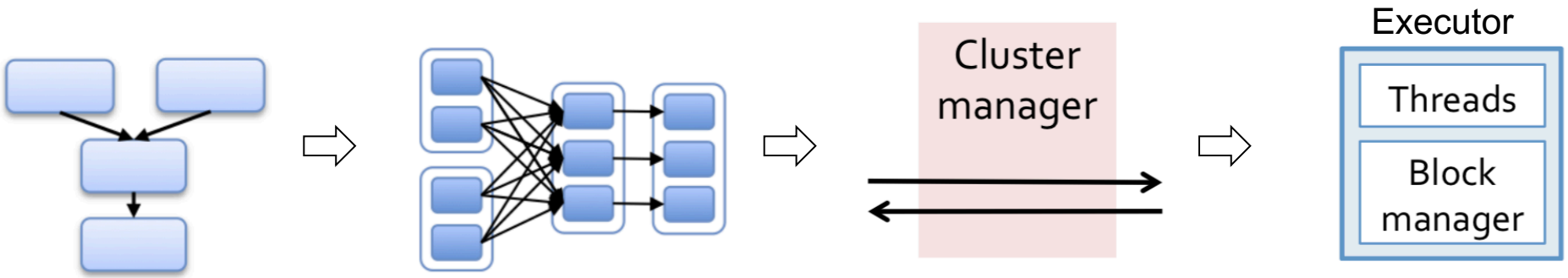
Structure interne d'un programme



Exécution de programme

Cycle de vie d'un programme

☞ Processus d'exécution d'un programme



Représenter le pgme par un DAG (Directed Acyclic Graph) de RDD

Découper le DAG en *tâches* parallèles

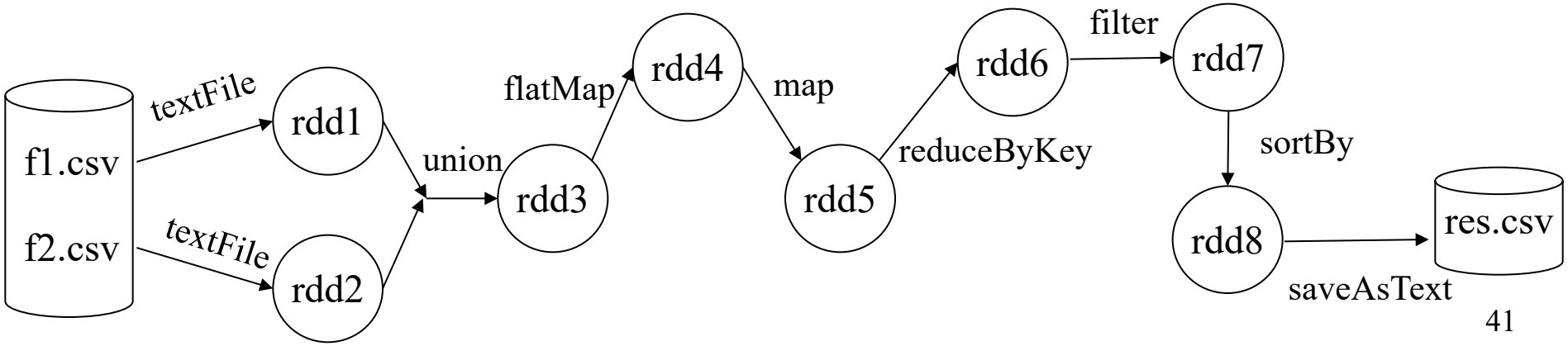
Demander les ressources de calculs au Master (*Executors*)

Envoyer les tâches aux *Executors* pour exécution

Exécution de programme

Représentation en DAG : Graphe orienté acyclique

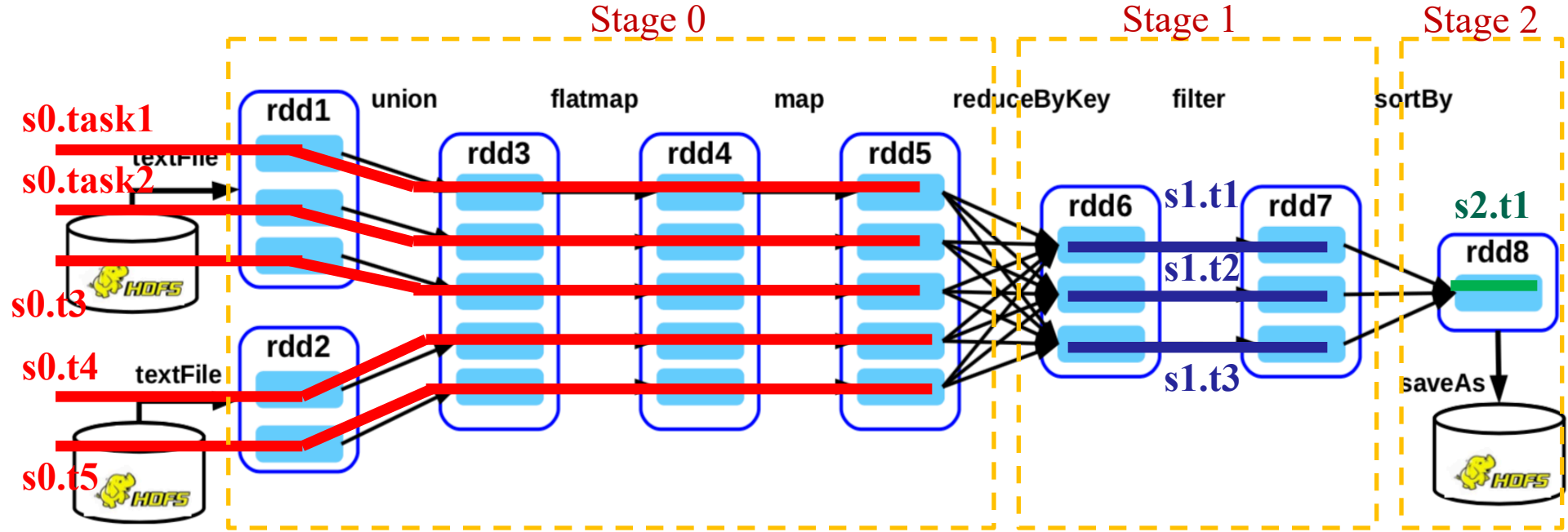
```
val rdd1= sc . textFile ("hdfs :/ /data/f1.csv")
val rdd2 = sc.textFile("hdfs:/ /data/f2.csv")
val rdd3 = rdd1.union(rdd2);
val rdd4 = rdd3.flatMap(_.split(" "))
val rdd5 = rdd4.map((_,1))
val rdd6 = rdd5.reduceByKey(_+_, 3)
val rdd7 = rdd6.filter(_._2 > 1)
val rdd8 = rdd7.sortBy(_._2, true, 1)
rdd8.saveAsTextFile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



Exécution de programme

Décomposition du DAG en tâches

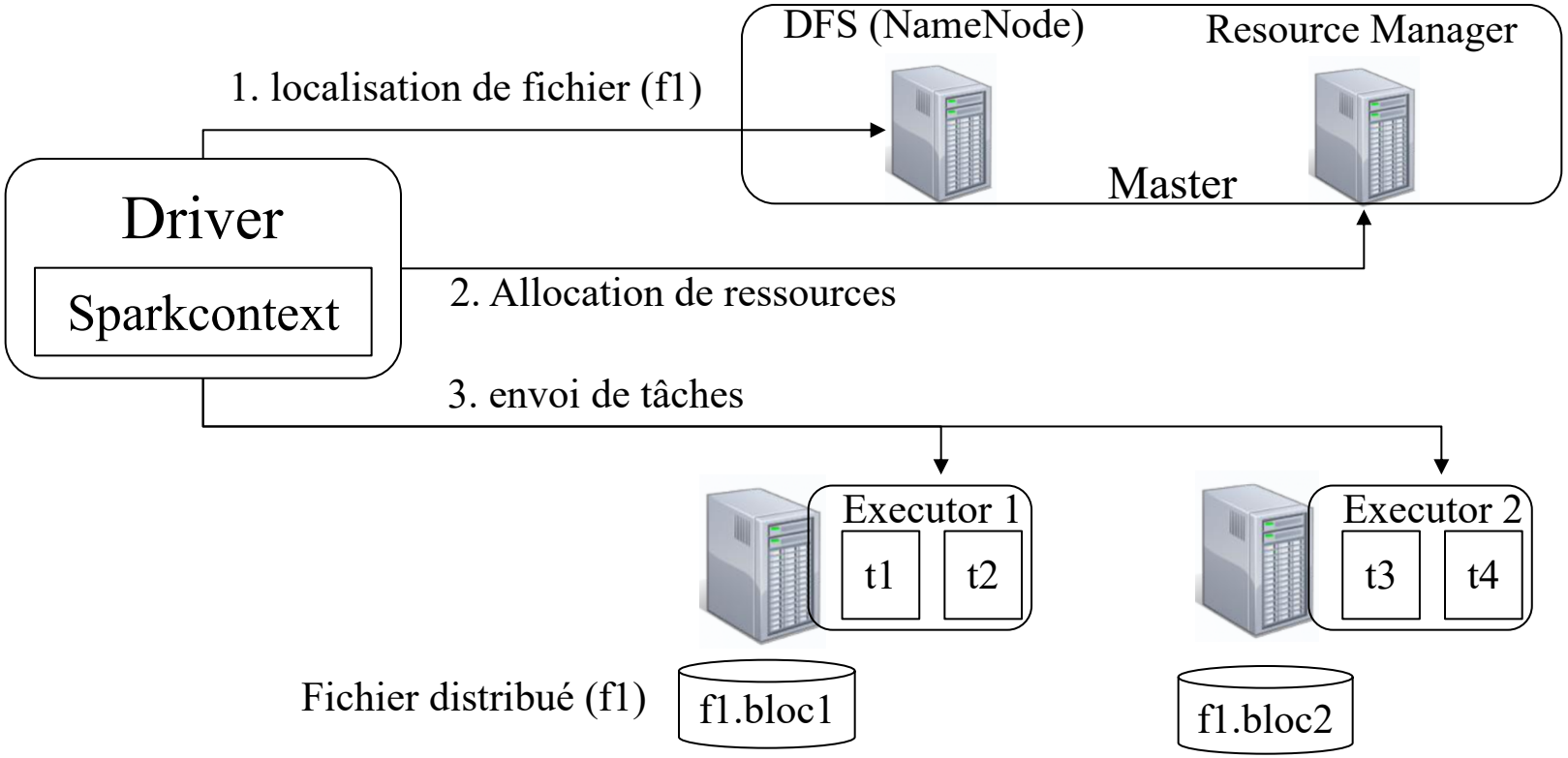
- Job : suite d'opérations de *transformation* terminées par une *action*
- Tâche :
  - Enchainement continu d'opérations sur une partition indépendamment des autres
  - Permet de paralléliser les opérations sur les cœurs (1 partition = 1 cœur)
- Stage : regroupement de tâches portant sur l'ensemble des partitions (terminé par une opération étendue ou une action)



Exécution de programme

Soumission des tâches

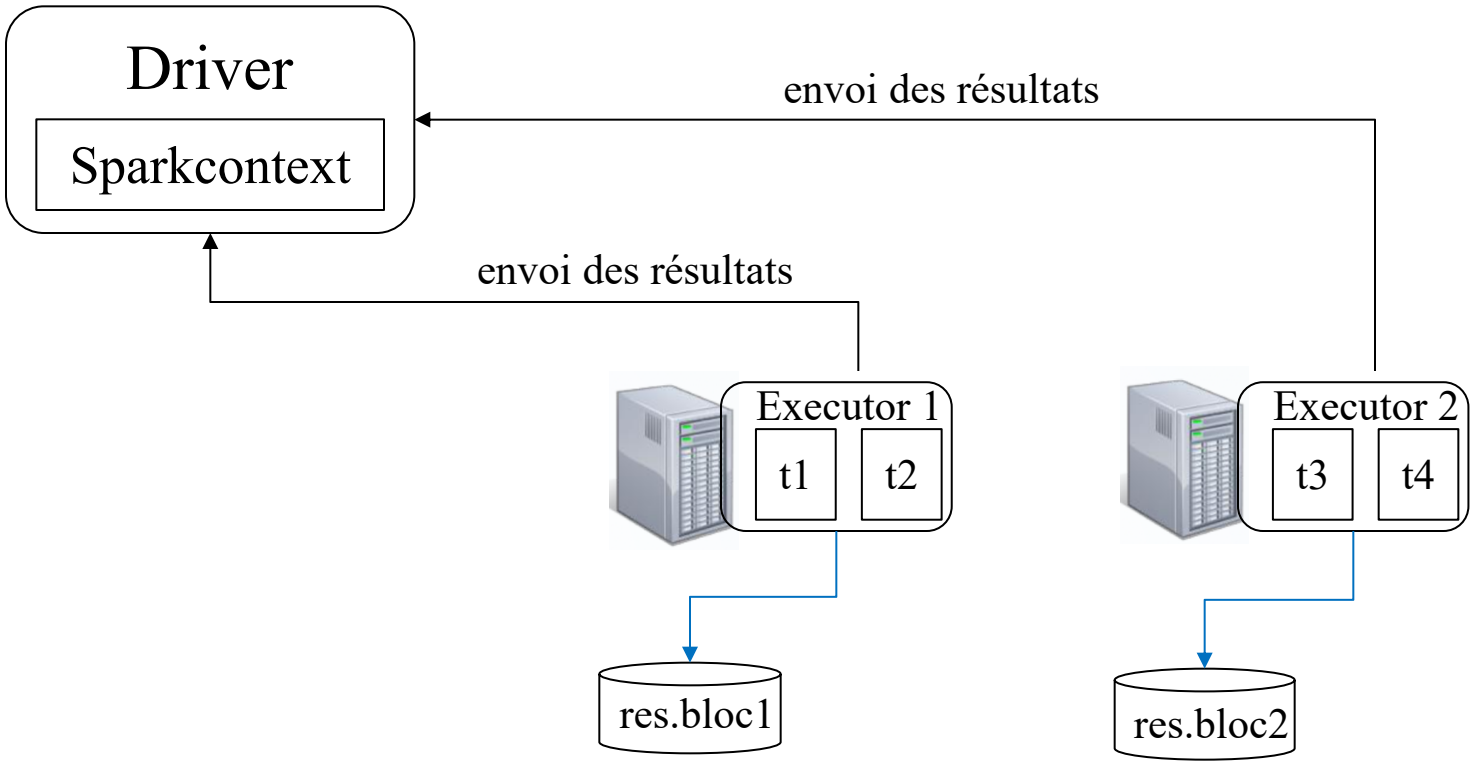
- Soumission de tâches du programme
  - Le *Driver* localise les données sur les nœuds de stockage (*datanodes* de *hadoop*)
  - Il demande au Ressource Manager d'allouer des containers sur ces nœuds (*data locality*)
  - Il lance les *executors* sur les nœuds de calculs et leur envoie les tâches à exécuter



Exécution de programme

Récupération des résultats

- Finalisation du programme : deux types de tâches terminales
  - Renvoi des résultats au *Driver* (actions : *collect, take, foreach, etc.*)
  - Stockage des résultats sur les disques distribués (actions: *saveAsTextFile, etc.*)



Persistance des RDD

Opérations

- Les RDD sont retirés dès que l'opération dessus s'est terminée

```
val rdd1 = sc.textFile("data.txt")
val rdd2 = rdd1.flatMap(l => l.split(" "))
val rdd3 = rdd2.map(w => (w, 1))
val rdd4 = rdd2.filter (e => e.equals ("paris"))
…
```

- Persister le RDD utilisé par plus d'une opération (c'est le cas de *rdd2* ci dessus) pour de meilleures performances :
  - En mémoire (*cache*) : rdd2.**cache()**
  - Désérialisé en mémoire ou sur disque : rdd2.**persist(**niveau de persistance**)**

Persistance des RDD

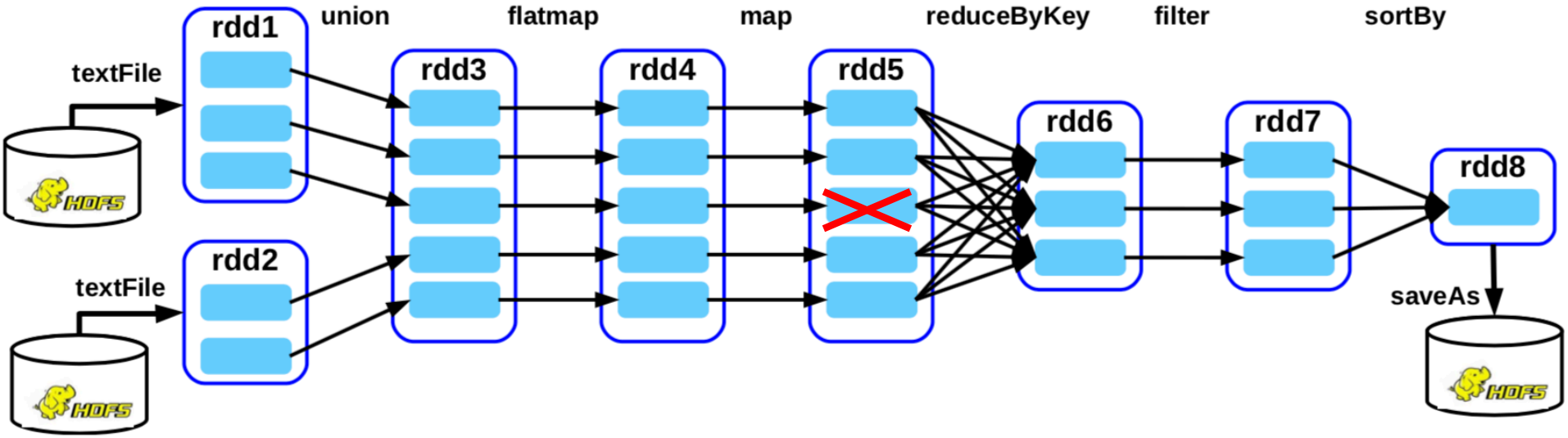
Niveaux de persistance

- MEMORY_ONLY et MEMORY_ONLY_SER : stockage du RDD en mémoire vive
  - Récupéré rapidement et occupe moins d'espace RAM si sérialisé
  - Risque de stockage sur disque si le RDD ne tient pas en mémoire
- MEMORY_AND_DISK et MEMORY_AND_DISK_SER : stockage du RDD en mémoire vive et sur le disque
  - Temps de stockage et de récupération plus élevé
  - Permet de récupérer les RDD en cas de perte (panne machine)
- DISK_ONLY : stockage du RDD en mémoire vive et sur le disque
  - récupération la moins performante
  - stockage de RDD volumineux possible
- Possibilité de répliquer les partitions sur plusieurs nœuds : MEMORY_ONLY_<N>, MEMORY_AND_DISK_<N>, DISK_ONLY_<N>, etc.

Tolérance aux pannes

Résilience de **R**DD

- Reconstruire les RDD en cas de perte d'une de ses partitions



- Récupération se fait comme suit :

  - Le recalculer à partir d'un RDD parent précédent gardé en cache ou sur disque
  - Tout recalculer à partir de la source si pas de persistance

Déploiement de programme Spark

Modes de déploiement de programmes (*Driver*)

- Polyvalence de Spark : exécution sur la machine locale ou sur cluster
- En local sur la machine cliente
  - Le programme s'exécute entièrement sur la machine du *driver* (le programme est parallélisé sur les cœurs du microprocesseur)
- En distribué sur un cluster de machines géré par :
  - Spark Master (*standalone*) : le programme s'exécute sur un cluster géré par Spark (Spark Ressource Manager)
  - Yarn Master (sur Hadoop) : le programme s'exécute sur un cluster Hadoop géré par Yarn (gestionnaire de ressources)
  - Mesos Master : le programme s'exécute sur un cluster géré par Mesos (gestionnaire de conteneurs)