Apache Spark : un système de traitement de données à large échelle

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

sources :

Contexte

Rôle des plate-formes de calcul d'analyse de données

- Écrire des programmes parallèles haut niveau (jobs Map Reduce)
- Abstraction de la distribution des données et des traitements
- Tolérance aux pannes
- Exécution performante des programmes

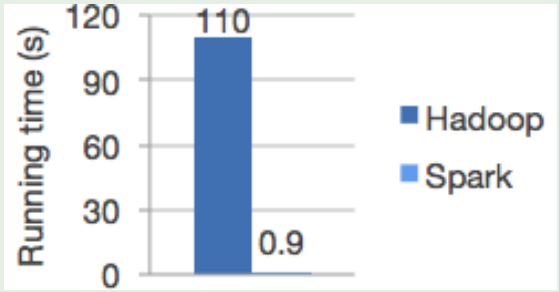
Besoins

- Éviter les I/O
  $\Rightarrow$ Privilégier la mémoire vive au disque
- Adresser facilement les données intermédiaire du calcul
  $\Rightarrow$ Produire une abstraction d'une mémoire partagée
- Exprimer facilement des transformations de données
  $\Rightarrow$ Utilisation du paradigme de programmation fonctionnelle
- Unifier les différentes API des traitements de données
  $\Rightarrow$ Avoir un outil tout en un

Spark : qu'est ce que c'est ?

Une plate-forme open-source pour le traitement massif de données

- Le projet top-level de Apache Software Foundation depuis 2014 (v 0.9)
- Une approche in-Memory : gain en performance

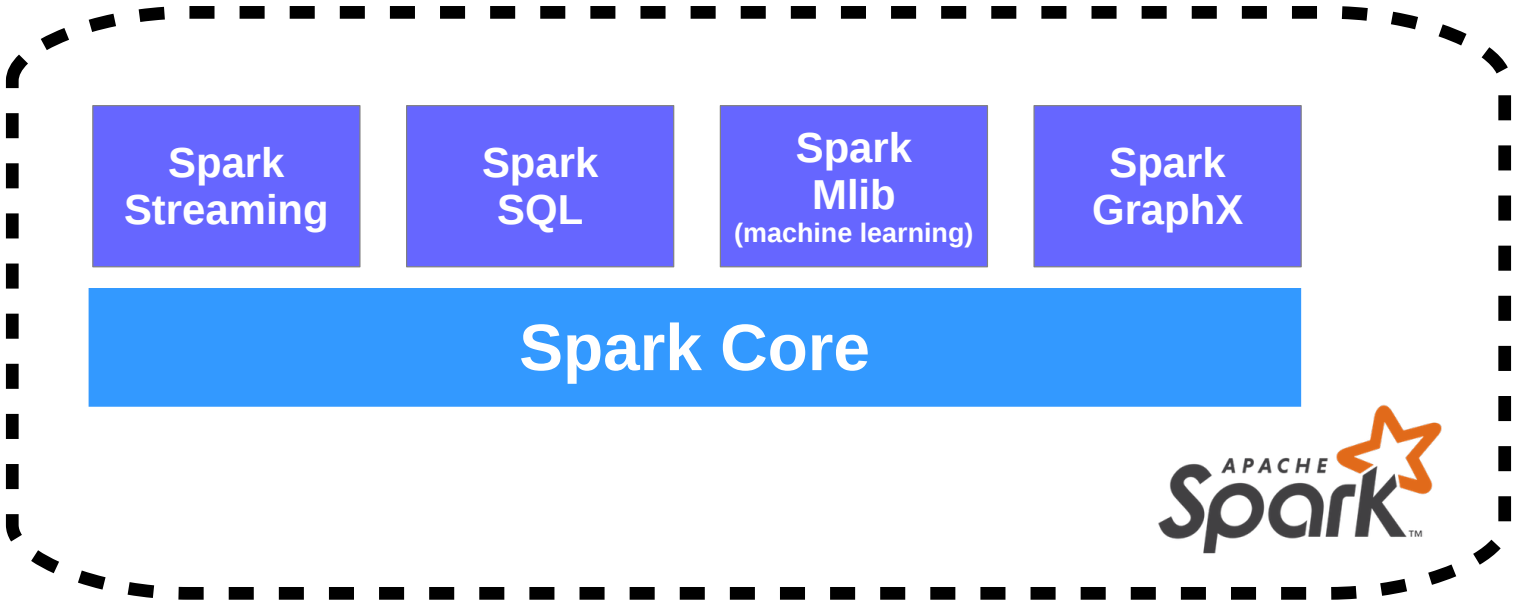


- Une généralisation du Map-Reduce avec une approche fonctionnelle
- Une abstraction des données : **Resilient Distributed Dataset (RDD)**
- Une API pour Scala, Java, Python et R
- Déploiement sur un cluster de machines
  $\Rightarrow$ utilisation d'un gestionnaire de ressources
- Pas de système de stockage propre
  mais très interfaçable avec HDFS, NFS, S3, etc..

Spark : qu'est ce que c'est ?

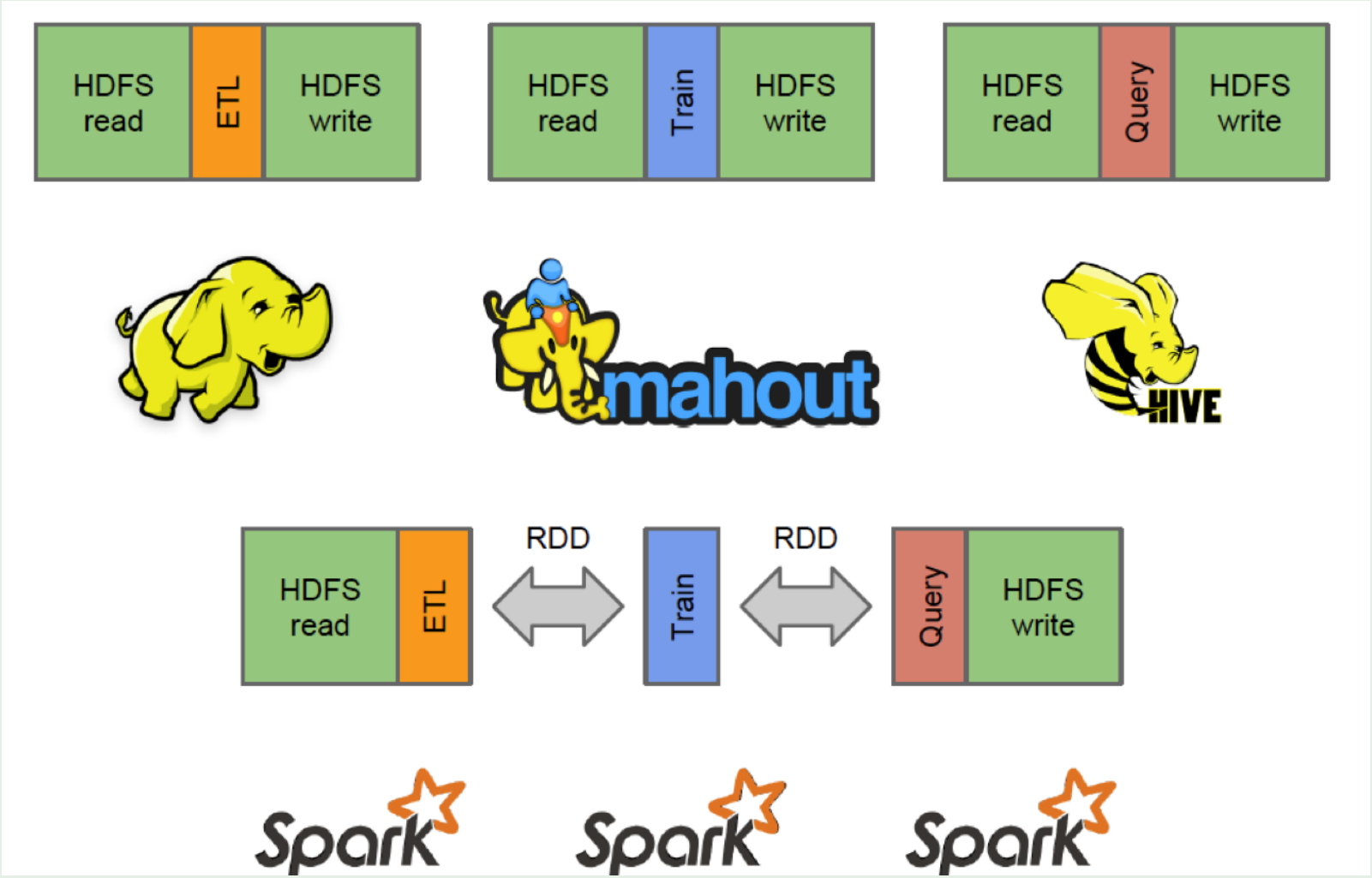
Une plate-forme unifiant plusieurs librairies

- **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 de données (machine learning)
- **Spark GraphX** : Librairie pour calcul de graphes

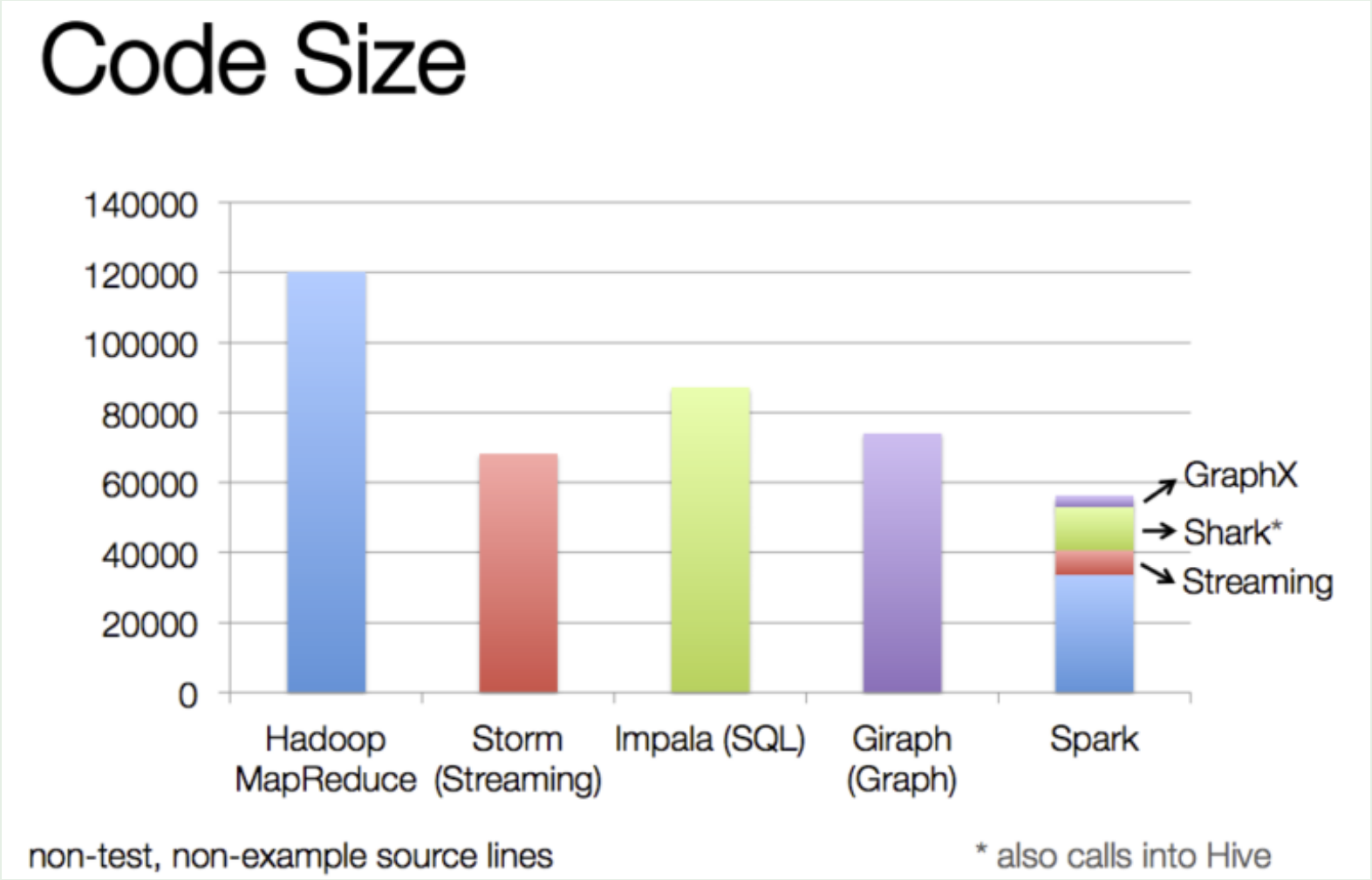


Spark : qu'est ce que c'est ?

Un flux de données simplifié

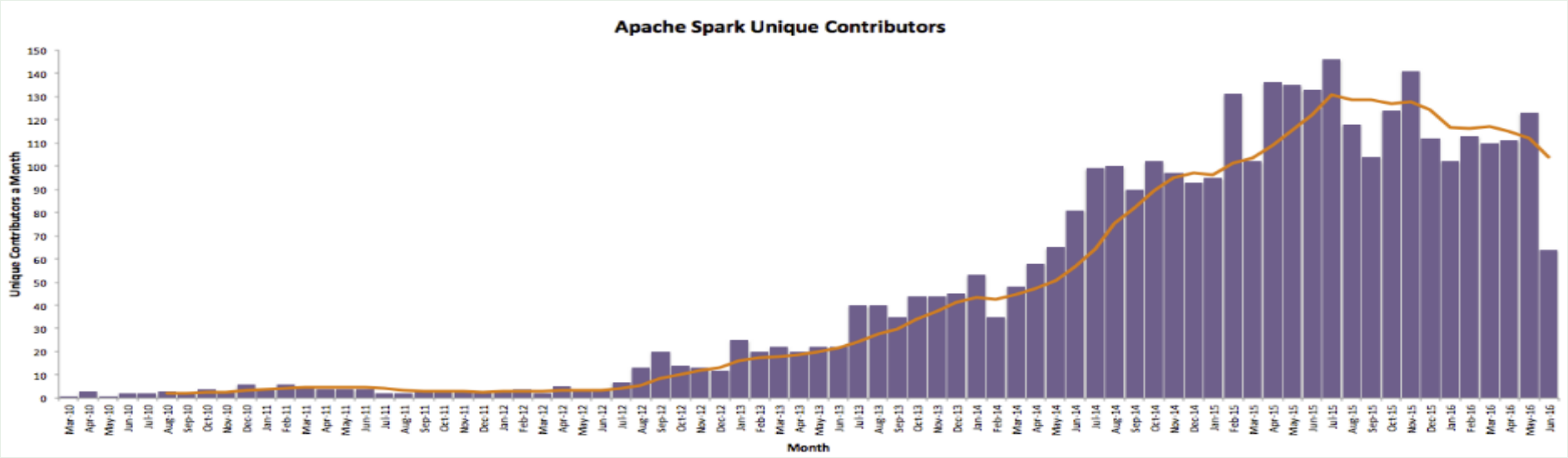


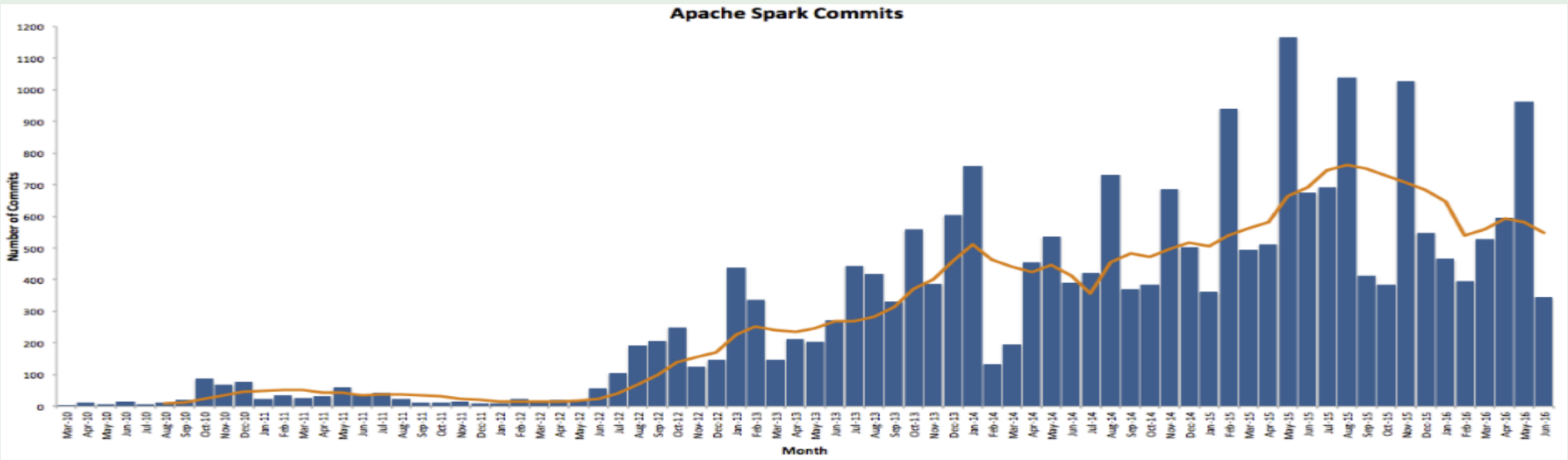
Une plate-forme écrite en Scala



Spark : qu'est ce que c'est ?

Une plate-forme très à la mode





Spark : qu'est ce que c'est ?

Un bref historique

- 2009 : conception initiale par Matei Zaharia en doctorat à Berkeley University.
- 2013 : reprise par la fondation Apache, devient l'un des projets les plus actifs
- 2014 : 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
- 2015 : plus de 1000 contributeurs venants de 200 entreprises
- Décembre 2017 : version 2.2.1

Spark : Qu'est ce que c'est ?

Les motivations pour l'apprendre

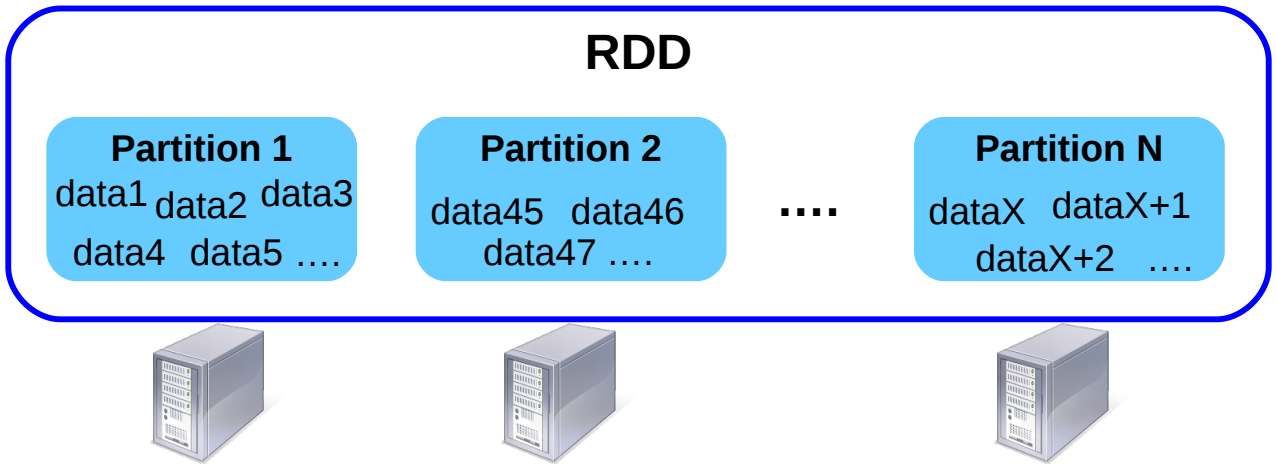
- Très compatible avec Hadoop
  - cohabitation possible sans surcoût
- Spark est beaucoup plus performant que Hadoop Map-Reduce
  - facteur 100
- Mieux adapté au "monde Big Data"
  - offre des outils performants de Machine Learning, de streaming, ..
- Multi-langage
  - Scala, Java, python, R
- Forte demande de développeurs Spark sur le marché du travail
- Des salaires très attrayants
  - $\simeq$ 100 K dollars aux USA en oct. 2017 (*src : Indeed.com*)

Resilient Distributed Dataset : le cœur de Spark

Definition

Un RDD est une collection de données :

- typée
- ordonnée (chaque élément a un index)
- partitionnée sur un ensemble de machines
- en lecture seule (immutabilité)
- résultante d'opérations déterministes.
- avec un niveau de persistance



L'API des RDD

Déclaration

```
abstract class RDD[T] extends Serializable with Logging
```

Attributs d'un RDD

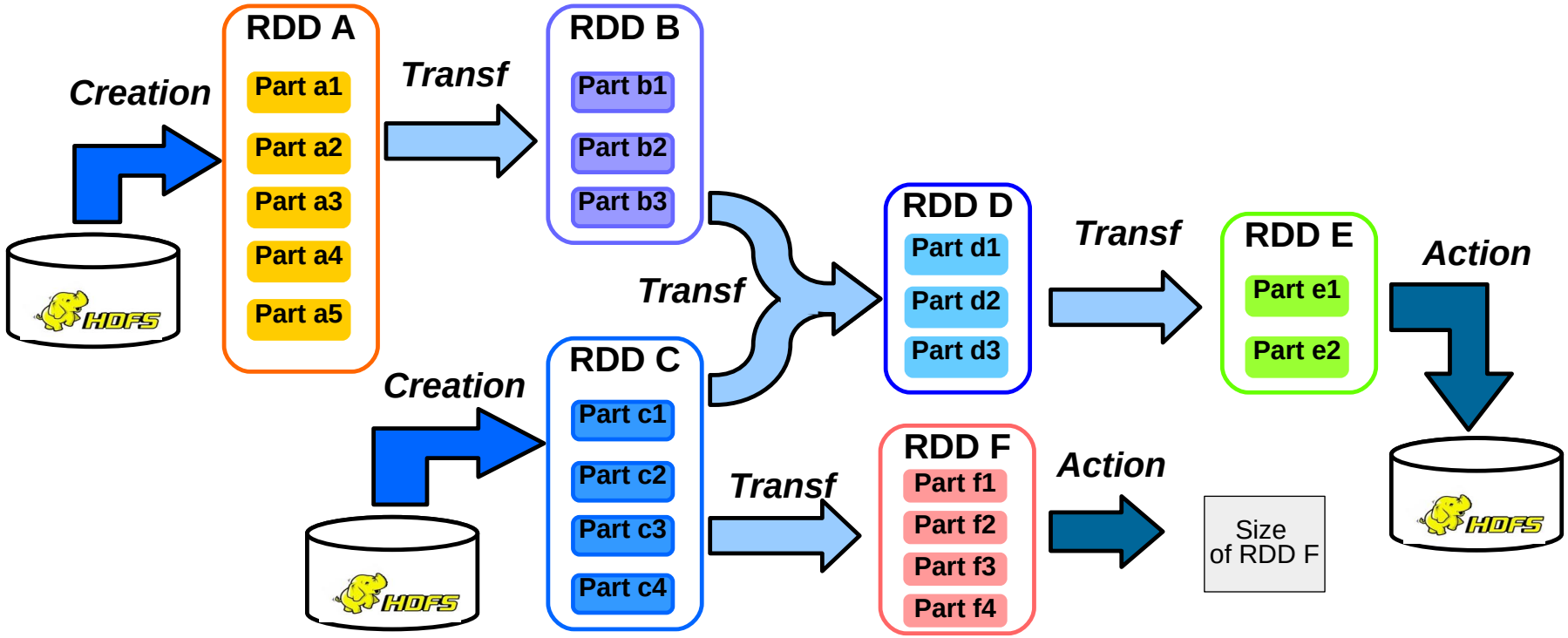
- un id : `val id: Int`, un nom : `var name: String`
- un ensemble de partitions :
  - `final def partitions: Array[Partition]`
  - `final def getNumPartitions: Int` : nombre de partitions
  - `final def preferredLocations(split: Partition): Seq[String]` : plans de distribution et l'emplacement des partitions (ex : localisation des blocks HDFS)
- un ensemble de dépendances aux RDDs parents :

  `final def dependencies: Seq[Dependency[_]]`
- Un partitionner : `val partitioner: Option[Partitioner]`
- une fonction de transformation calculant les données depuis les parents

Schéma d'un code Spark

Les 4 étapes

- 1) Initialisation d'un sparkcontext
- 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



Étape 1 du code : l'objet SparkContext

Définition

Point d'entrée principal pour les fonctionnalités de Spark :

- Connexions avec l'infrastructure distribuée
- Stockage des méta-données lors de l'exécution du programme
- Création de RDD
- Création d'accumulateurs ou variables de diffusion

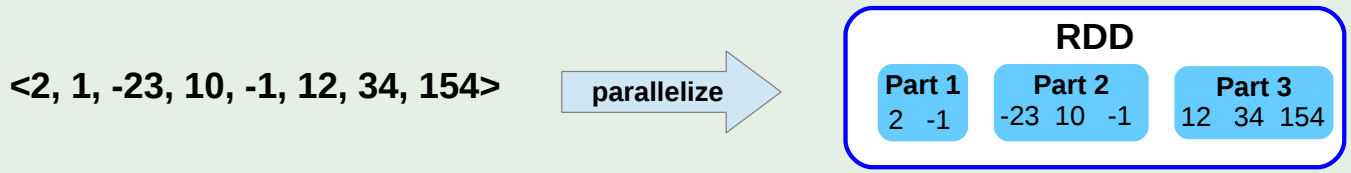
```
object MonProgSpark extends App {

 val conf = new SparkConf().setAppName("Mon Job")
 val sc = new SparkContext(conf)
 //code du programme Spark
}
```

Étape 2 du code : Création de RDD avec le SparkContext

Depuis une collection existante en mémoire du programme client

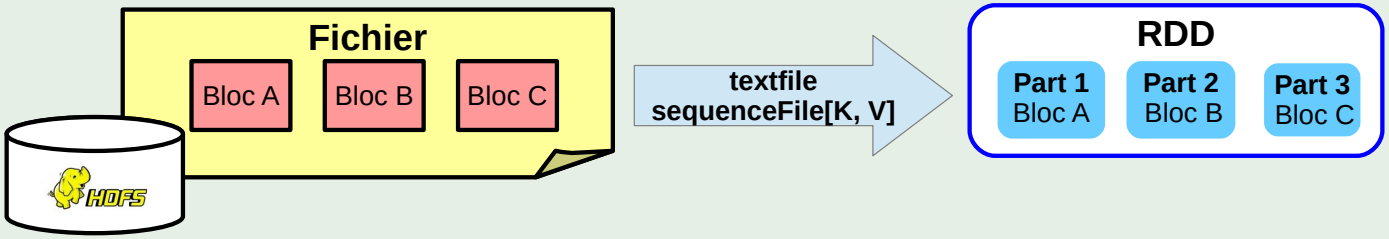
```
def parallelize[T](seq: Seq[T], numSlices: Int): RDD[T]
```



Depuis des données sur un stockage stable (ex : HDFS)

```
def sequenceFile[K, V](path: String, keyClass: Class[K],
    valueClass: Class[V], minPartitions: Int = 2): RDD[(K, V)]

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



Étape 3 du code : Transformations de RDDs

Caractéristique des transformations

- 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

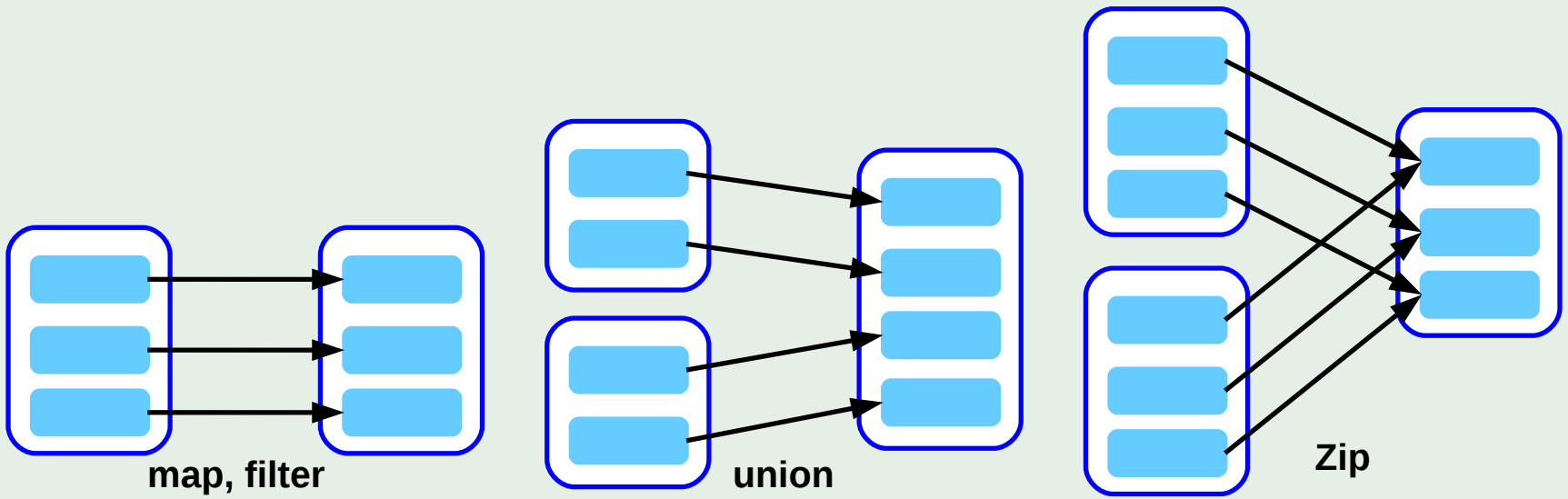
Dépendances étroites entre RDDs

Une relation 1 to 1

Chaque partition d'un parent RDD est utilisée par au plus une partition d'un RDD fils

$\Rightarrow$ Aucune synchronisation nécessaire pour calculer le RDD fils

Exemples :

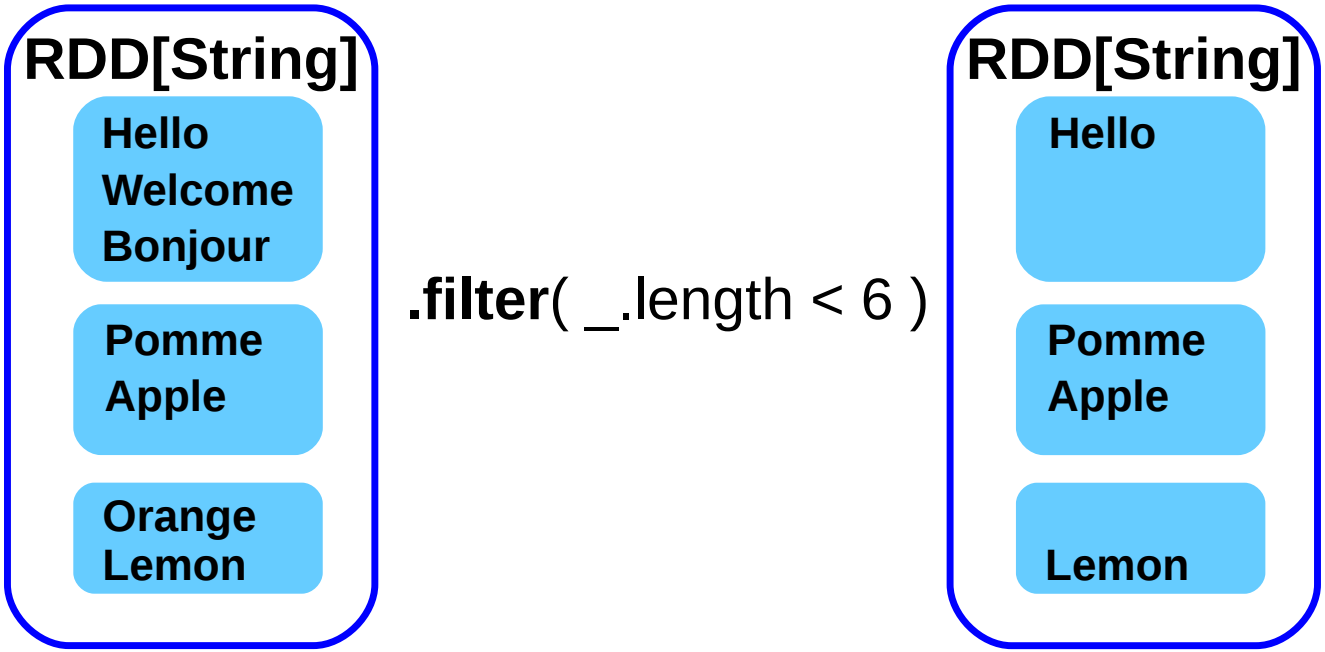


Transformation étroite `filter`

Spécification

Retourne un nouvel RDD contenant seulement les éléments qui satisfont un prédicat

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

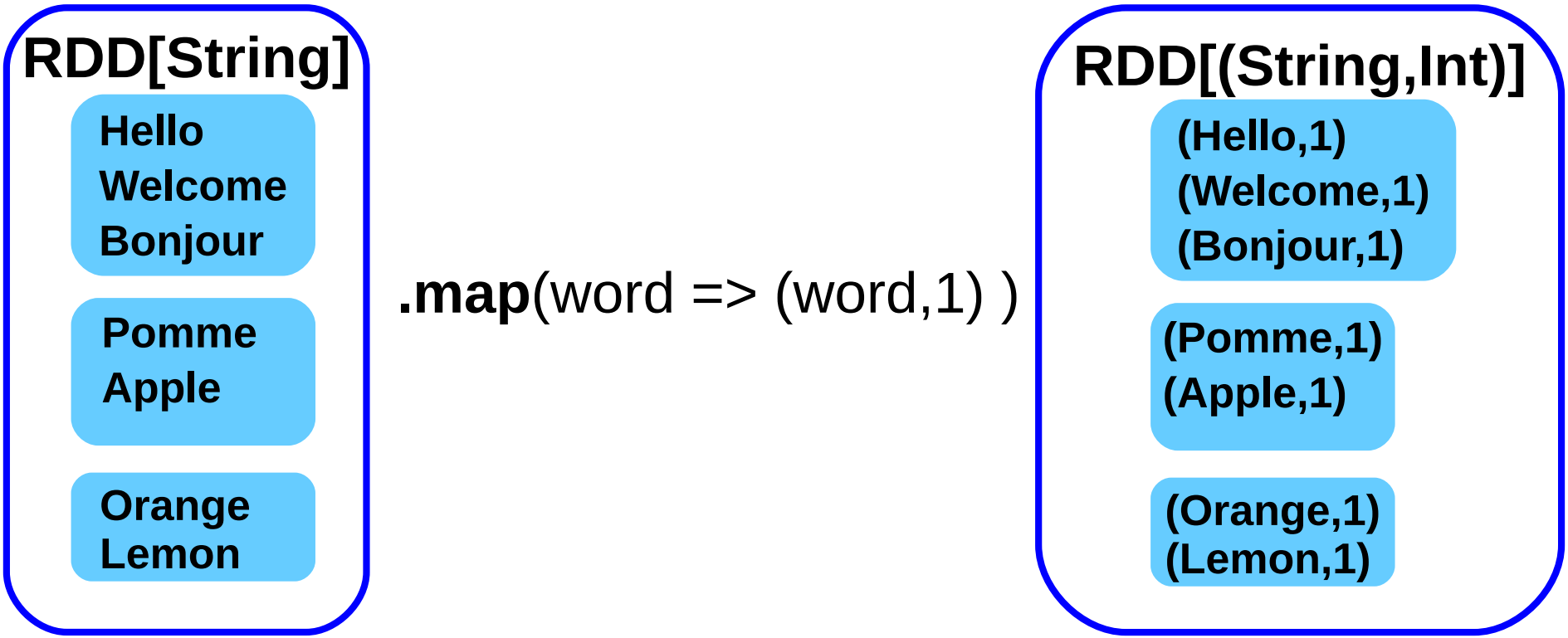


Transformation étroite map

Spécification

Retourne un nouvel RDD en appliquant une fonction à tous les éléments

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

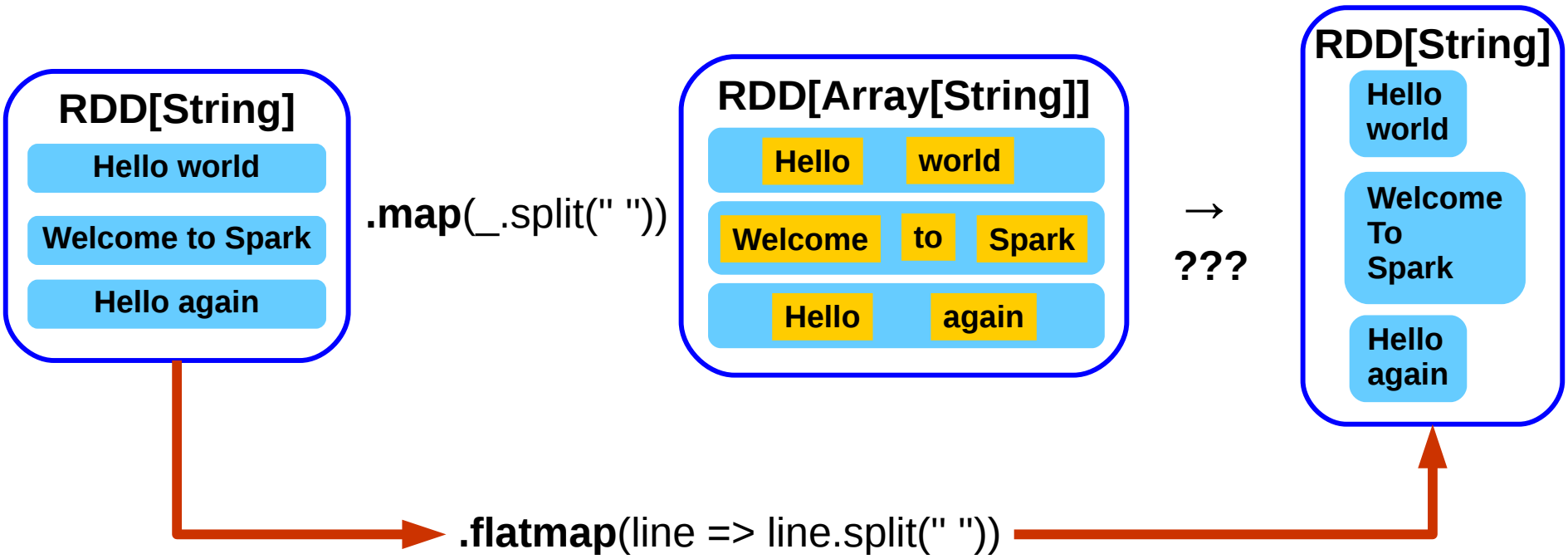


Transformation étroite `flatmap`

Spécification

Retourne un nouvel RDD en appliquant d'abord une fonction à tous les éléments puis unit les résultats

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

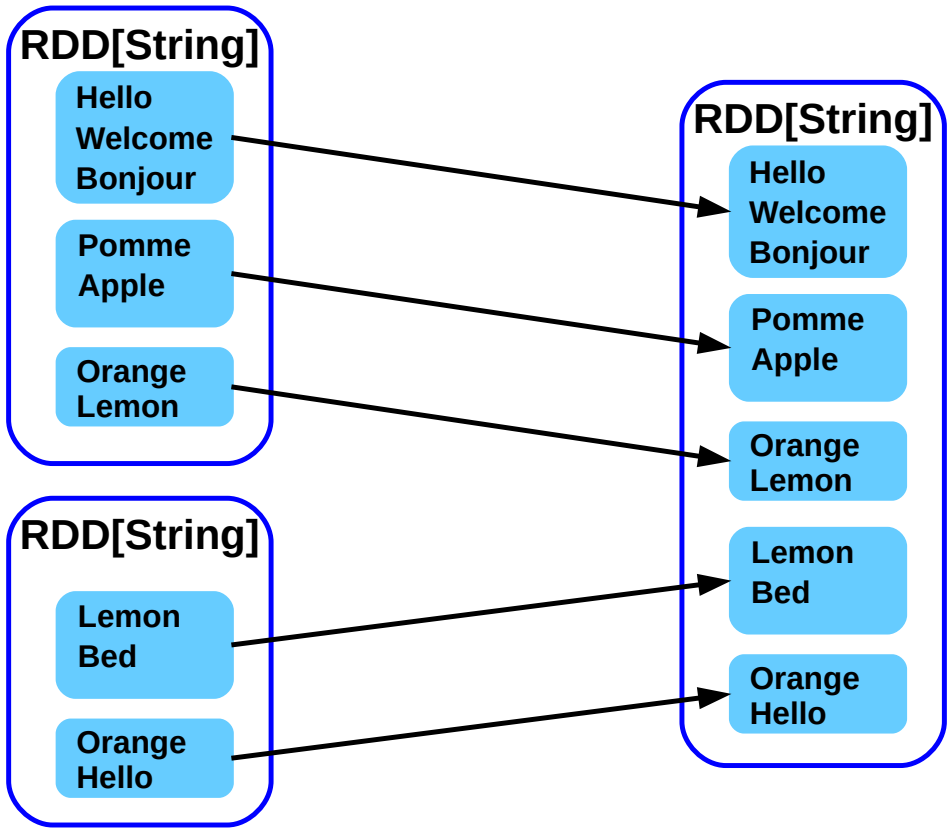


Transformation étroite Union

Spécification

Retourne l'union de deux RDD. Les doublons sont conservés.

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



Transformation étroite Zip

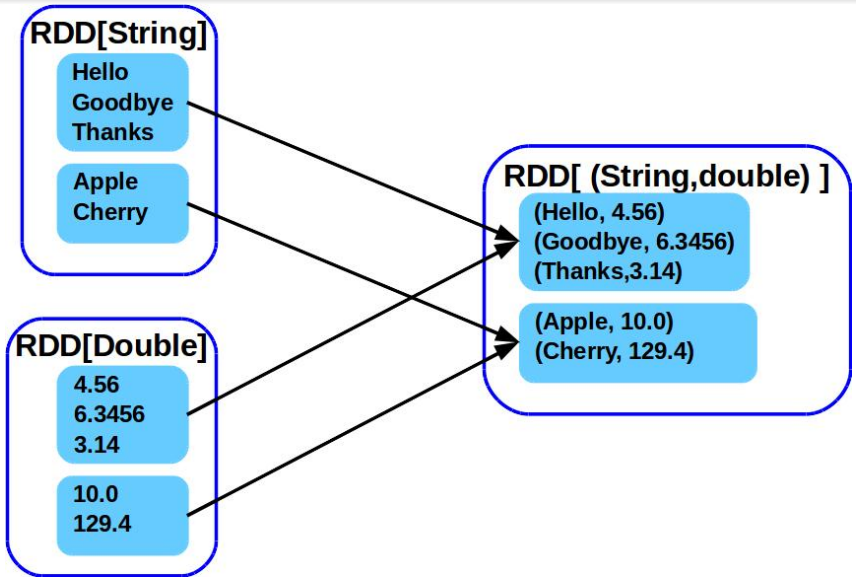
Spécification

Lie deux RDD en créant un RDD de couples clé-valeur, où le *n*ième couple est l'association des *n*ièmes éléments de chaque RDD.

```
def zip[U](other: RDD[U]): RDD[(T, U)]
```

Préconditions :

- Les deux RDD ont le même nombre de partitions
- Chaque partition correspondante ont le même nombre d'éléments



Variante utile

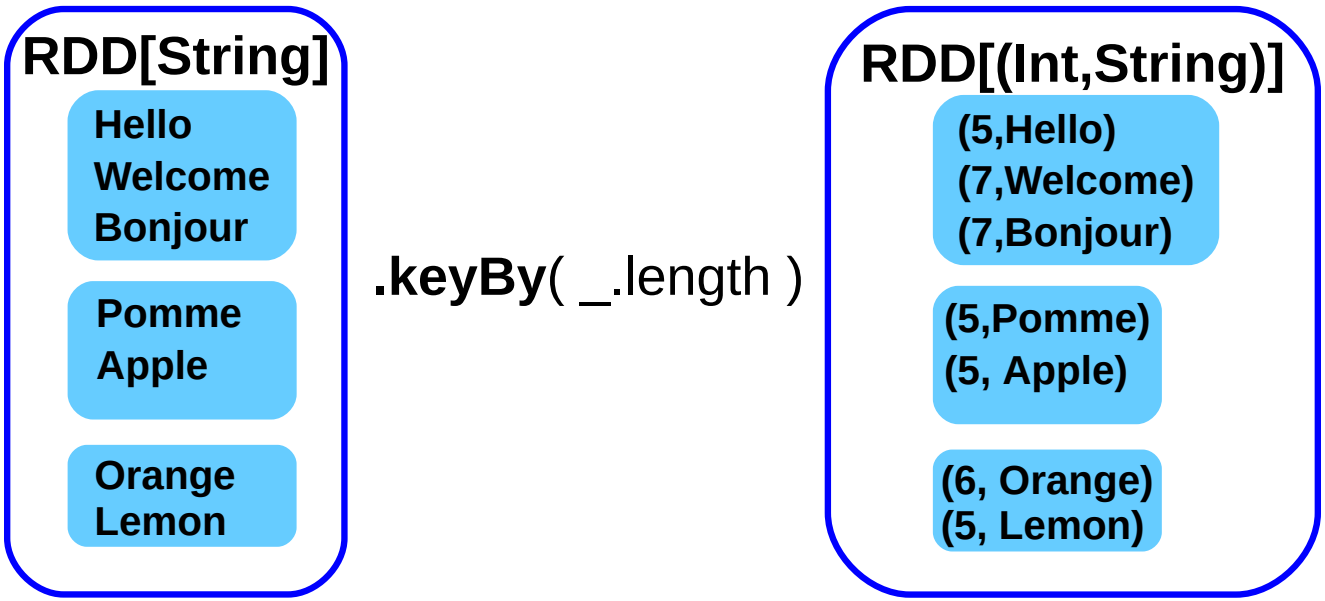
`def zipWithIndex(): RDD[(T, Long)]` : lier chaque élément avec son indice dans le RDD.

Transformation étroite keyBy

Spécification

Créé un RDD de tuple, en liant à chaque élément du RDD initial une clé

```
def keyBy[K](f: (T) =>K): RDD[(K, T)]
```



Dépendances larges entre RDDs

Une relation all to all (= Shuffle)

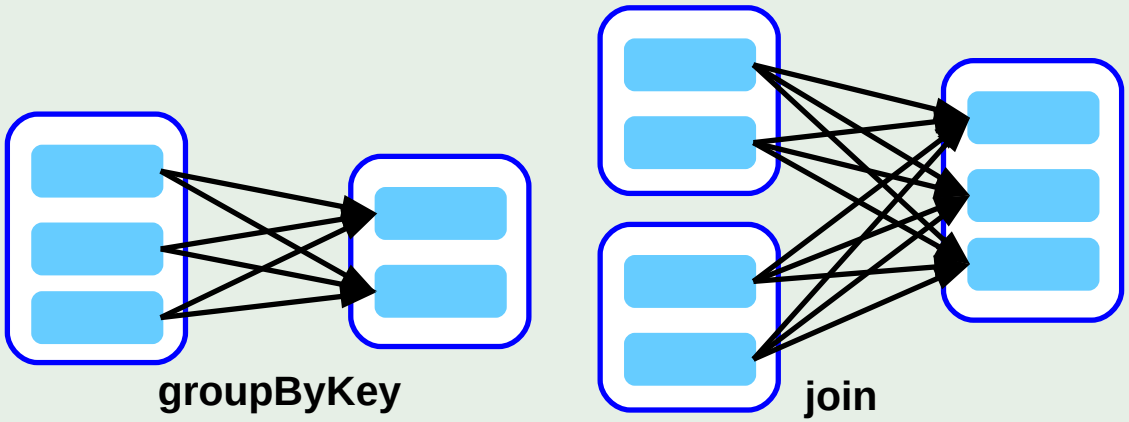
Les partitions filles dépendent de toute les partitions mères

⇒ I/O disque et réseau, opération de sérialisation

⇒ synchronisation nécessaire

Opérations coûteuses

Exemples :

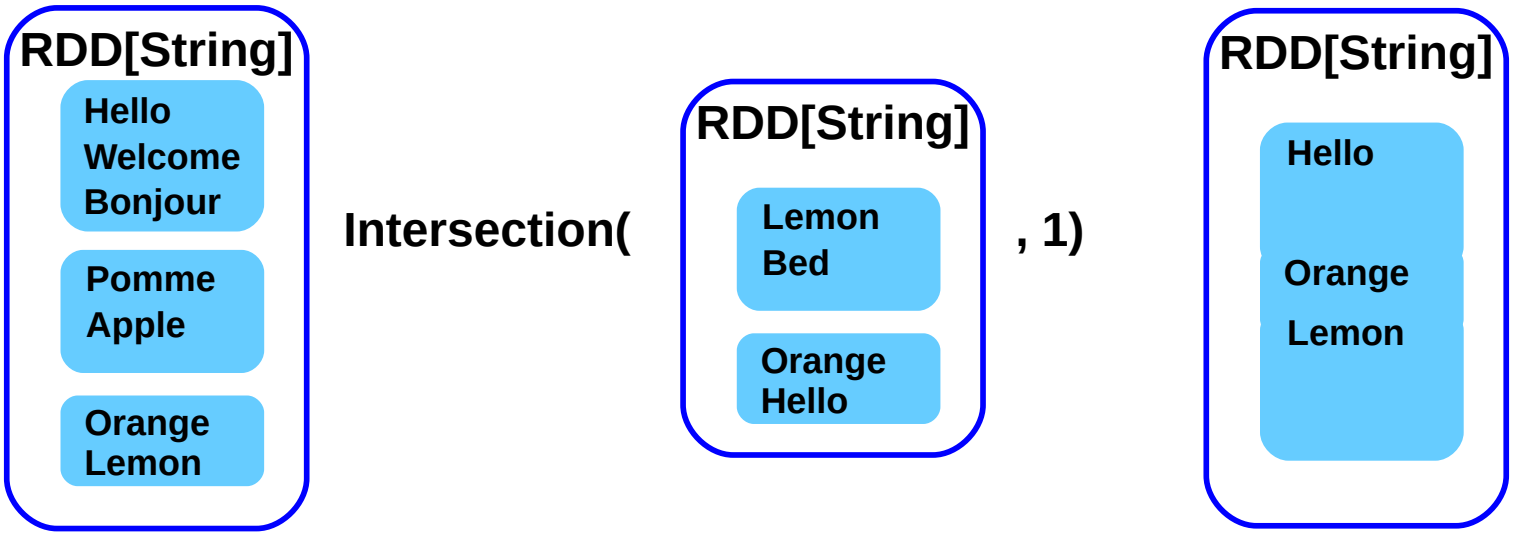


Transformation large `Intersection`

Spécification

Retourne l'intersection entre 2 RDD en supprimant les doublons.

```
def intersection(other: RDD[T]): RDD[T]
def intersection(other: RDD[T], numPartitions: Int): RDD[T]
def intersection(other: RDD[T], partitioner: Partitioner): RDD[T]
```

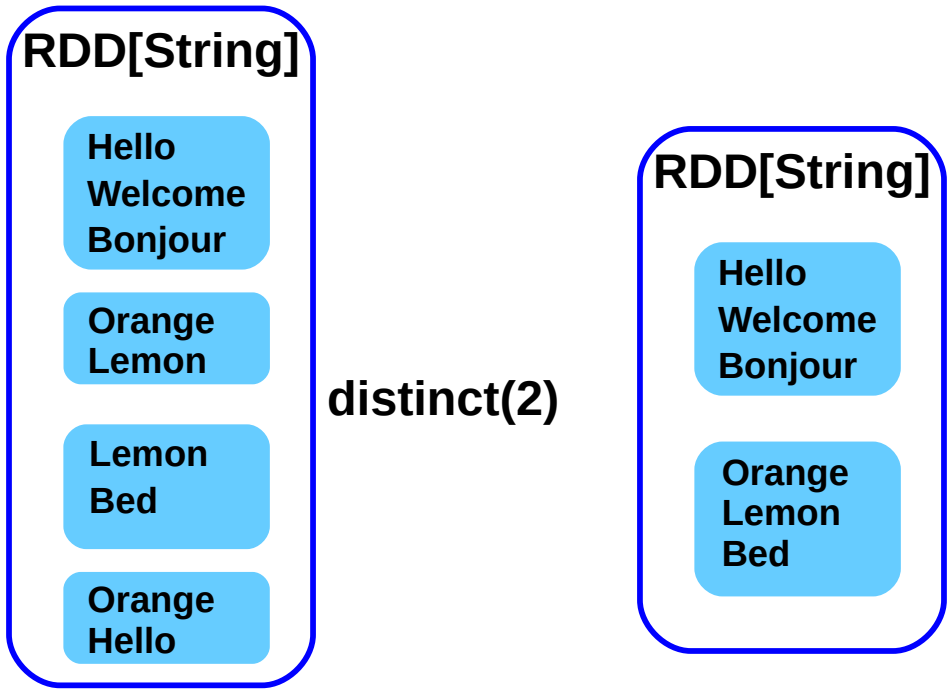


Transformation large `distinct`

Spécification

Retourne un nouvel RDD en supprimant les doublons du RDD appelant.

```
def distinct(): RDD[T]
def distinct(numPartitions: Int): RDD[T]
```

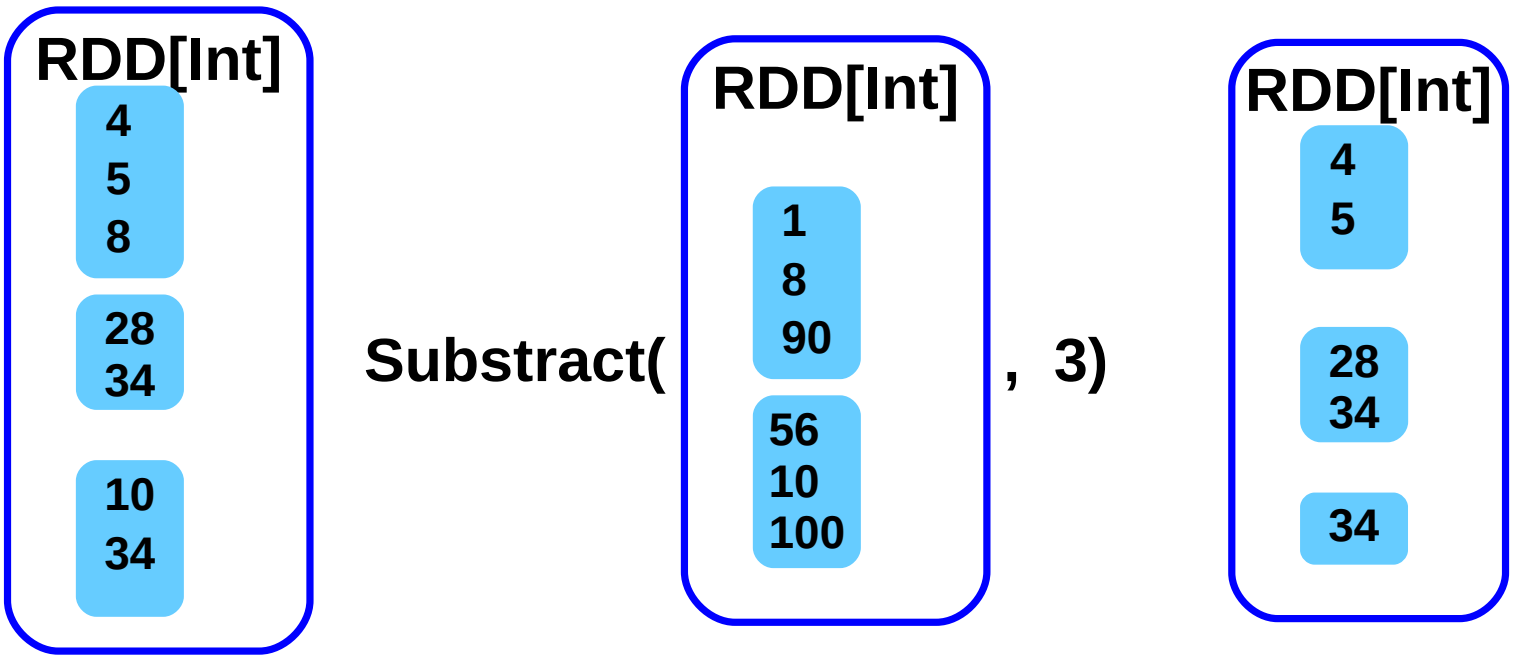


Transformation large subtract

Spécification

Retourne un nouvel RDD avec les éléments du RDD appelant qui ne sont pas dans un autre RDD

```
def subtract(other: RDD[T]): RDD[T]
def subtract(other: RDD[T], numPartitions: Int): RDD[T]
def subtract(other: RDD[T], p: Partitioner): RDD[T]
```

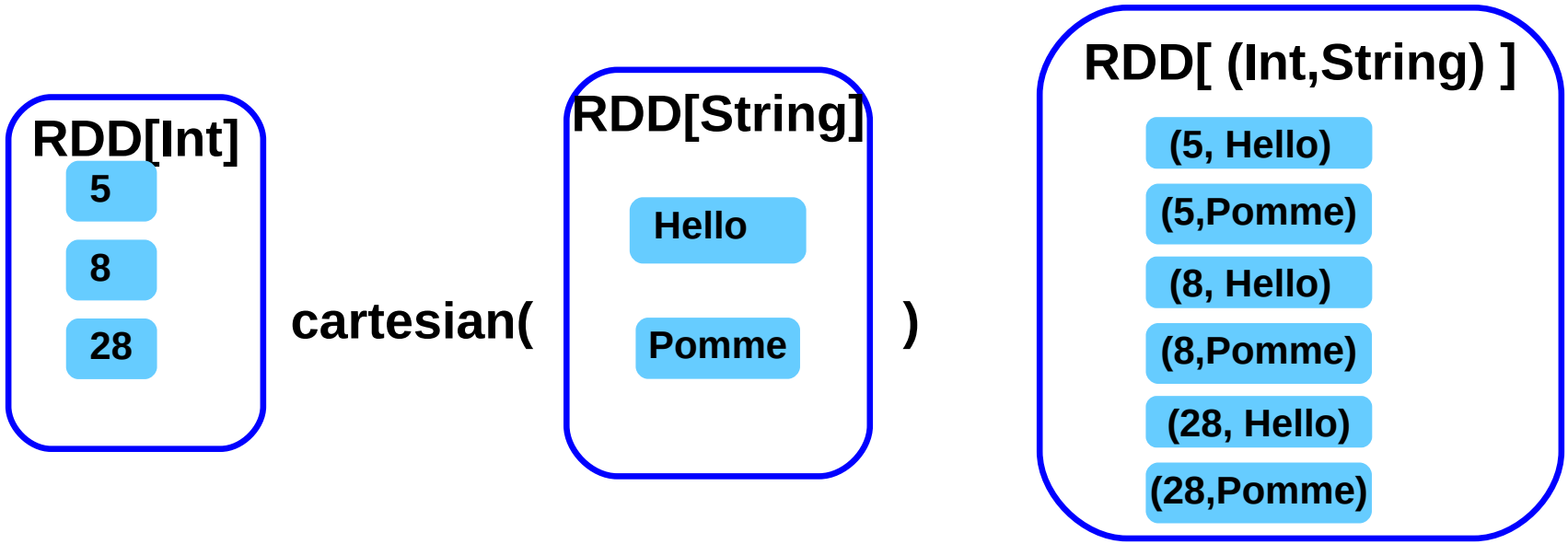


Transformation large `cartesian`

Spécification

Retourne un nouvel RDD égal au produit cartésien de deux RDDs : tous les couples d'élément (a, b) où a appartient au RDD appelant et b au RDD

```
def cartesian[U](other: RDD[U]): RDD[(T, U)]
```

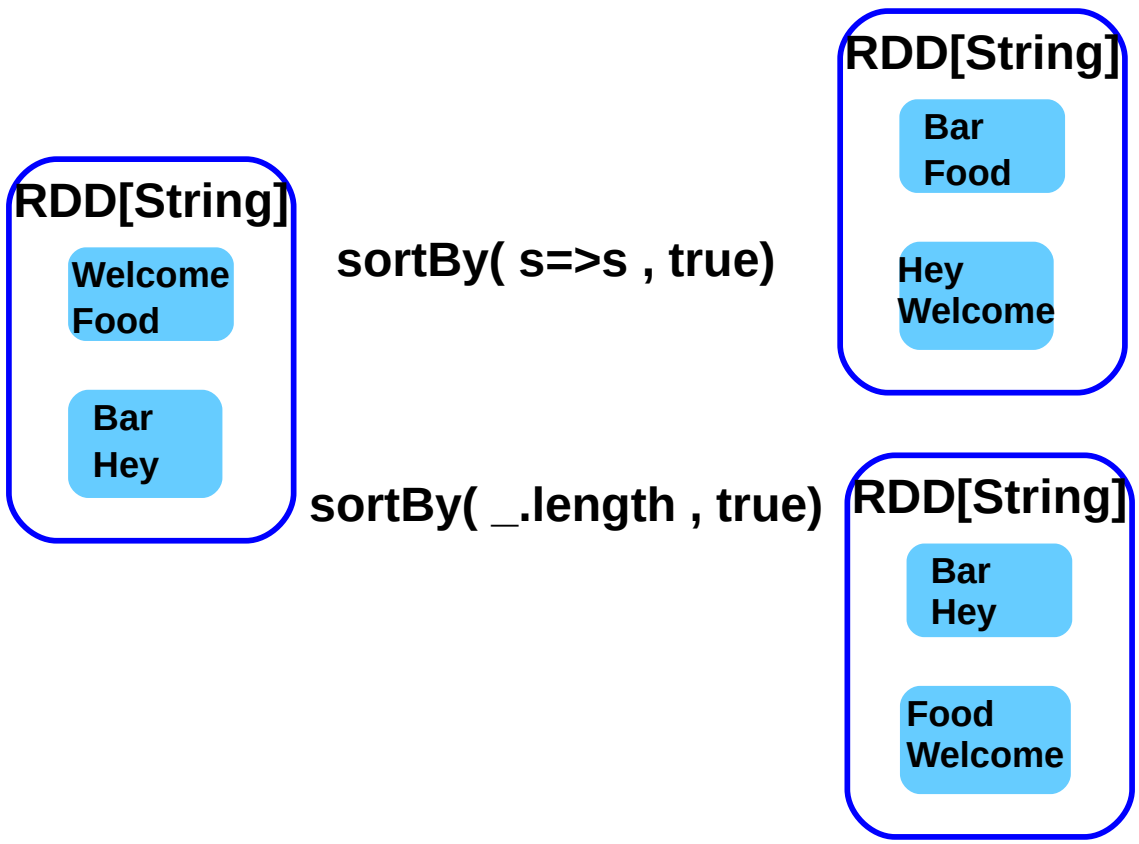


Transformation large sortBy

Spécification

Retourne un nouvel RDD trié en selon une fonction de tri.

```
def sortBy[K](f: (T) =>K, ascending: Boolean = true
    , numPartitions: Int = this.partitions.length): RDD[T]
```

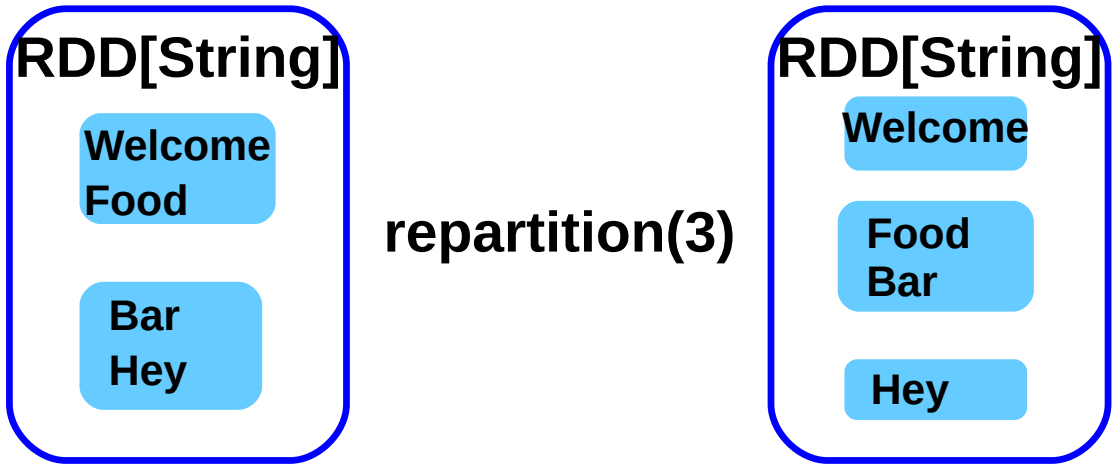


Transformation large `repartition`

Spécification

Retourne un nouvel RDD qui a exactement un nombre de partitions passé en paramètre

```
def repartition(numPartitions: Int): RDD[T]
```



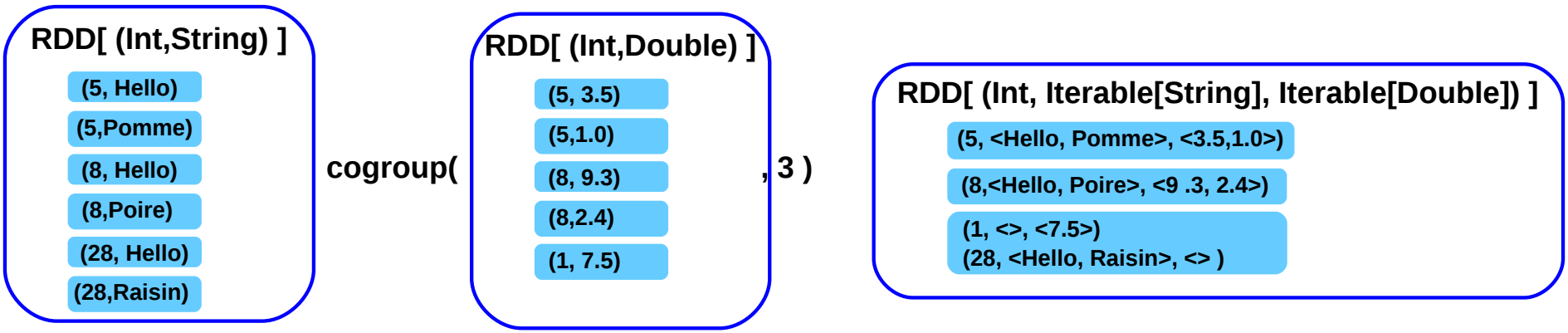
Transformation large cogroup

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

Spécification

Pour chaque clé `k` d'un RDD A ou B, retourne un RDD qui contient un tuple avec la liste des valeurs présentent dans A et la liste des valeurs présentent dans B.

```
def cogroup[W](o: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]
def cogroup[W](o: RDD[(K, W)], numPart: Int): RDD[(K, (Iterable[V], Iterabl
def cogroup[W](o: RDD[(K, W)], p: Partitioner): RDD[(K, (Iterable[V], Itera
```



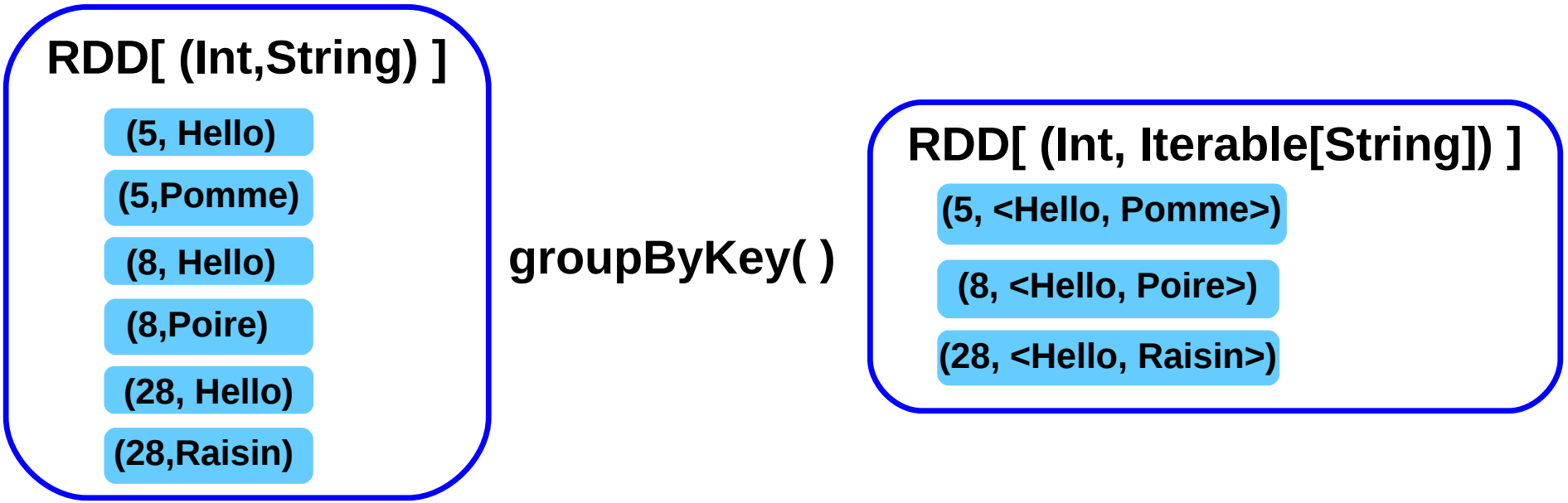
Transformation large groupByKey

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

Spécification

Groupe les valeurs de chaque clé. L'ordre des valeurs dans un groupe n'est pas déterministe.

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



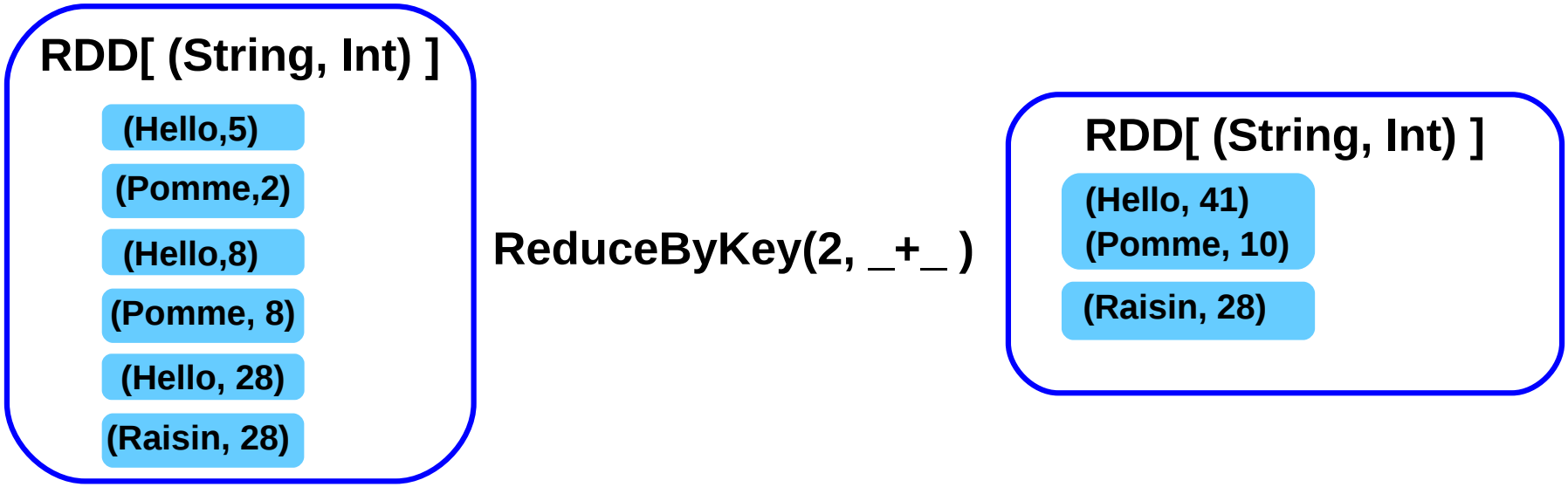
Transformation large `reduceByKey`

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

Spécification

Groupe les valeurs de chaque clé et applique pour chaque groupe une fonction associative et commutative de réduction. L'ordre des valeurs dans un groupe n'est pas déterministe.

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



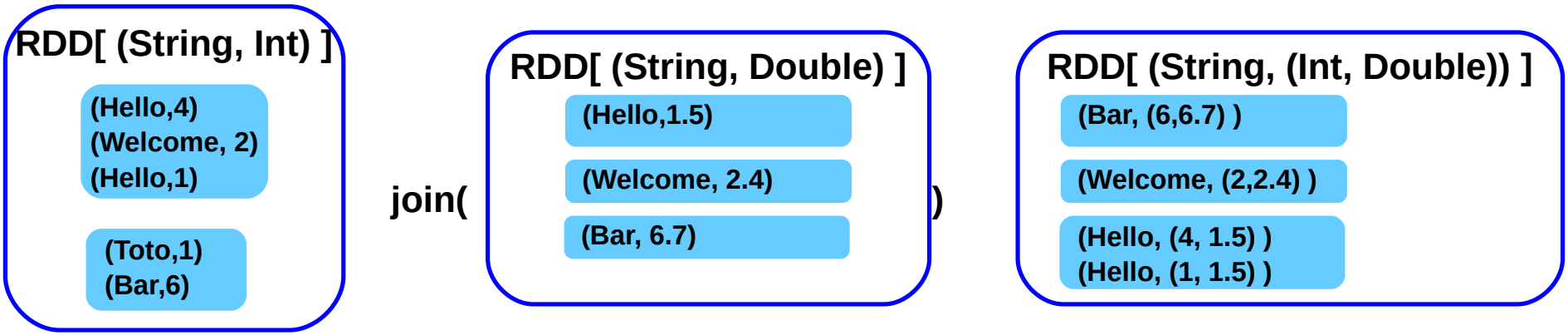
Transformation large `join`

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

Spécification

Retourne un RDD contenant tout paire d'éléments qui correspondent à la même clé dans un RDD A et B

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



Variantes : `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`

Étape 4 du code : Les actions

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** :

  $\Rightarrow$ 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 courantes sur les RDDs

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 courantes sur les RDDs

Actions de sauvegarde

- `def saveAsObjectFile(url_file: String): Unit` :
  Sauvegarde en tant qu'objets sérialisés dans un fichier référencé par une URL

- `def saveAsTextFile(url_file: 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]](url_file: String): Unit
```

Sauvegarde au format binaire Hadoop sur un format clé-valeur.

Un programme Spark

```
object Programme {
  def main(args: Array[String]):Unit= {
    val conf = new SparkConf().setAppName("Mon␣Programme");
    val spark = new SparkContext(conf)
    val textFile = spark.textFile("hdfs://...")
    val res = textFile.flatMap(line => line.split("␣"))
                      .map(x => (x, 1))
                      .reduceByKey(_ + _)
    res.saveAsTextFile("hdfs://...")
   }
}
```

Que fait ce programme ? Combien de RDD sont créés dans ce programme ?

Réponses

- WordCount
- 4 RDDs produits :

  `textFile` (`RDD[String]`), `textFile.flatMap` (`RDD[String]`), `textFile.flatMap.map` (`RDD[(String,Int)]`) et `res` (`RDD[(String,Int)]`)

La persistance des RDD

Caractéristiques

- Sauvegarder les partitions d'un RDD sur les nœuds qui l'héberge
- La sauvegarde se fait selon un niveau de stockage (en cache ou disque)

$\Rightarrow$ tolérance aux pannes

$\Rightarrow$ réutilisation possible sans recalculer le RDD.

Méthodes

- `def cache: RDD.this.type`
  $\Rightarrow$ Persiste en mémoire vive
- `def persist(newLevel: StorageLevel): RDD.this.type`
  $\Rightarrow$ Persiste en spécifiant un niveau de stockage
- `def unpersist(blocking: Boolean = true): RDD.this.type`
  $\Rightarrow$ Annule la persistance en mode bloquant ou non.

La persistance des RDD

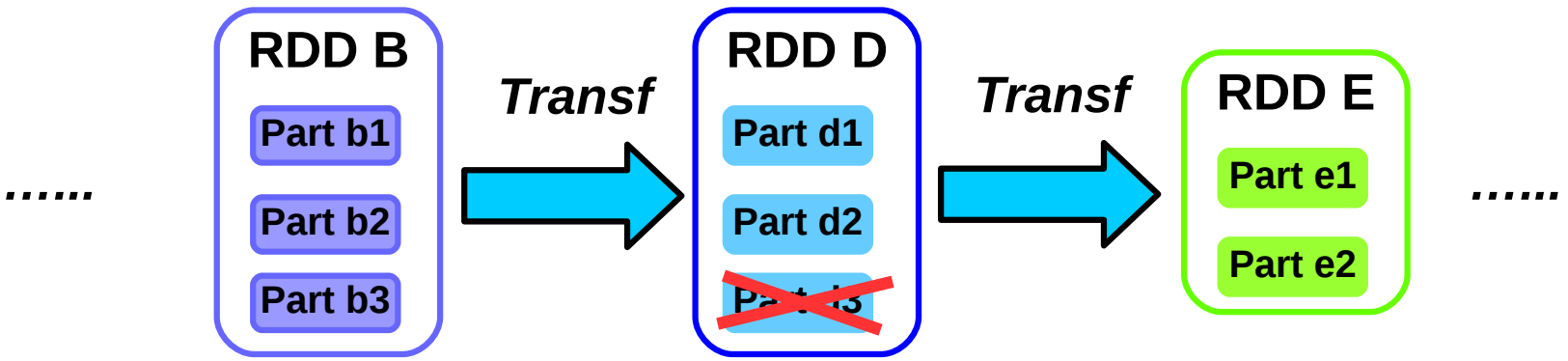
Les niveaux de stockage

- `MEMORY_ONLY` et `MEMORY_ONLY_SER` : met en cache le RDD
  - ⇒ rapide
  - ⇒ plus économe en mémoire si sérialisé
  - ⇒ risque de perte de partition si le RDD ne tient pas en mémoire
- `MEMORY_AND_DISK` et `MEMORY_AND_DISK_SER` : met en cache le RDD et utilise le disque local mémoire insuffisante
  - ⇒ moins rapide
  - ⇒ pas perte de données si RDD volumineux
- `DISK_ONLY` : stocke le RDD entièrement sur disque
  - ⇒ forte latence
  - ⇒ sauvegarde possible de RDD très volumineux

Réplications des partitions sur 2 nœuds

`MEMORY_ONLY_2`, `MEMORY_ONLY_SER_2` `MEMORY_AND_DISK_2`,
`MEMORY_AND_DISK_SER_2`,`DISK_ONLY_2`

Tolérance aux pannes



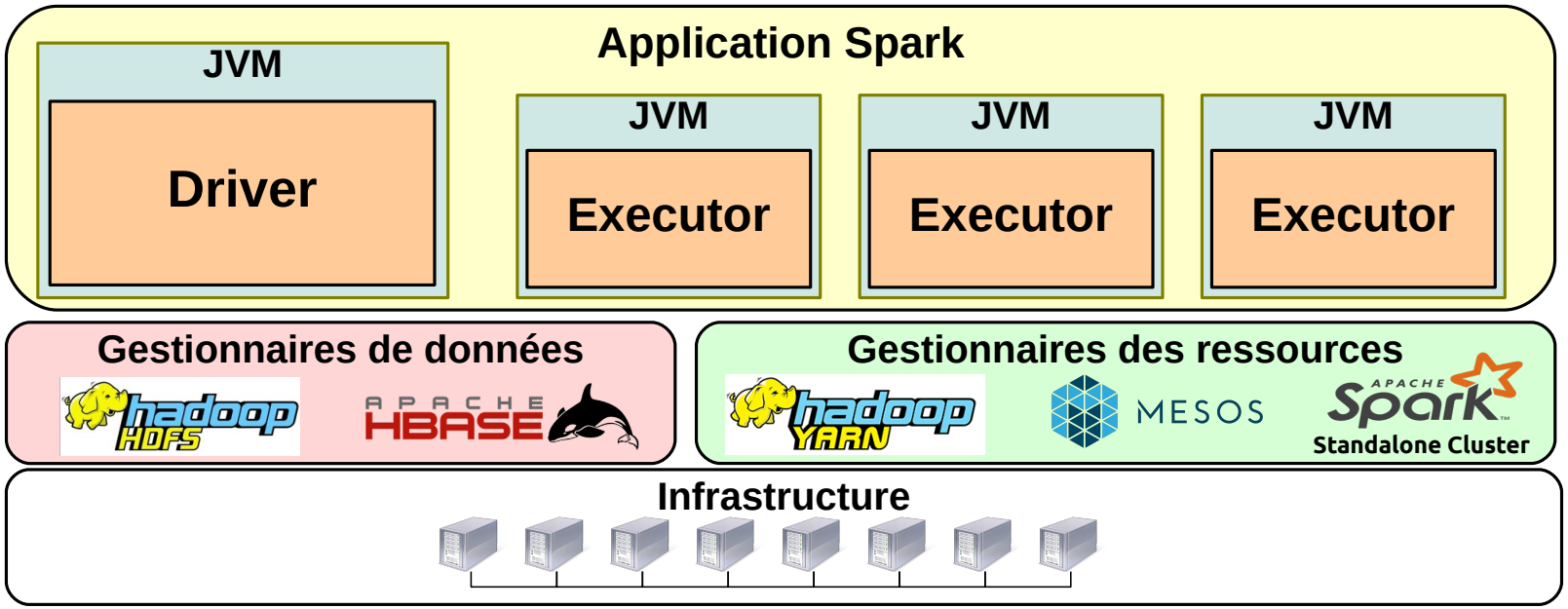
Comment retrouver la partition d3 perdue à la suite d'une panne ?

- Sans persistance : Retour à la case départ
- Avec persistance : reprendre à partir de l'ancêtre persistant le plus proche
- Avec persistance et réplication : faute transparente

Application spark

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

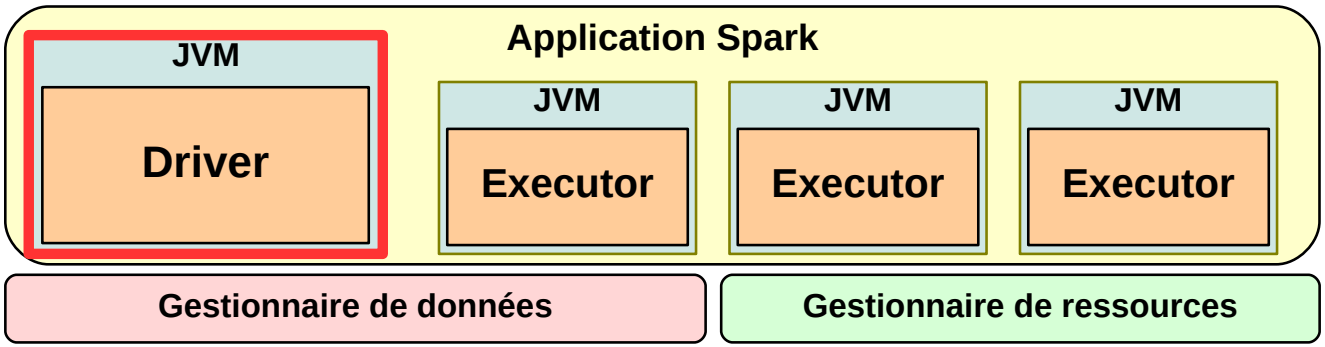


Le driver

Objectifs

JVM maître exécutant le main de l'application

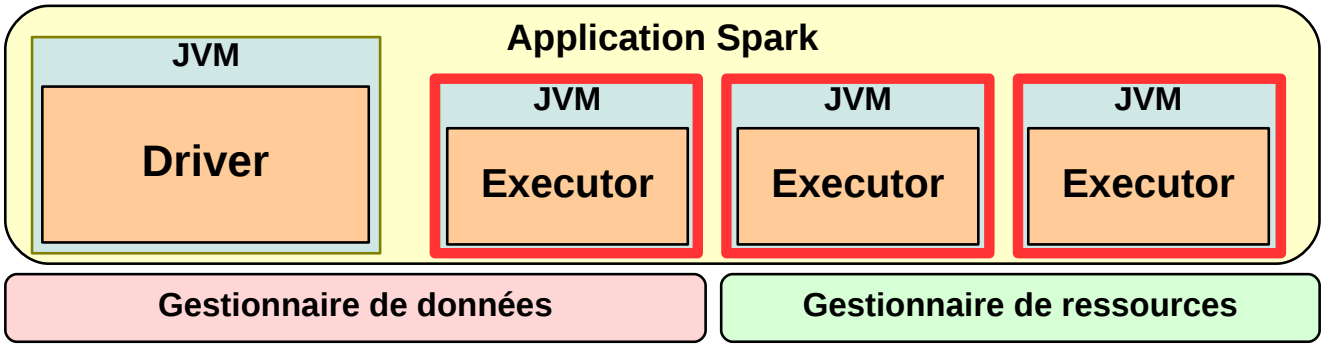
- Interaction avec les gestionnaires de ressources et données
- Définition des tâches :
  - code
  - placement
  - dépendances (transfert de données)
- Orchestration de l'exécution des tâches :
  - Affectation sur les executors
  - Surveillance des tâches terminées ou défaillantes



Les executors

Caractéristiques

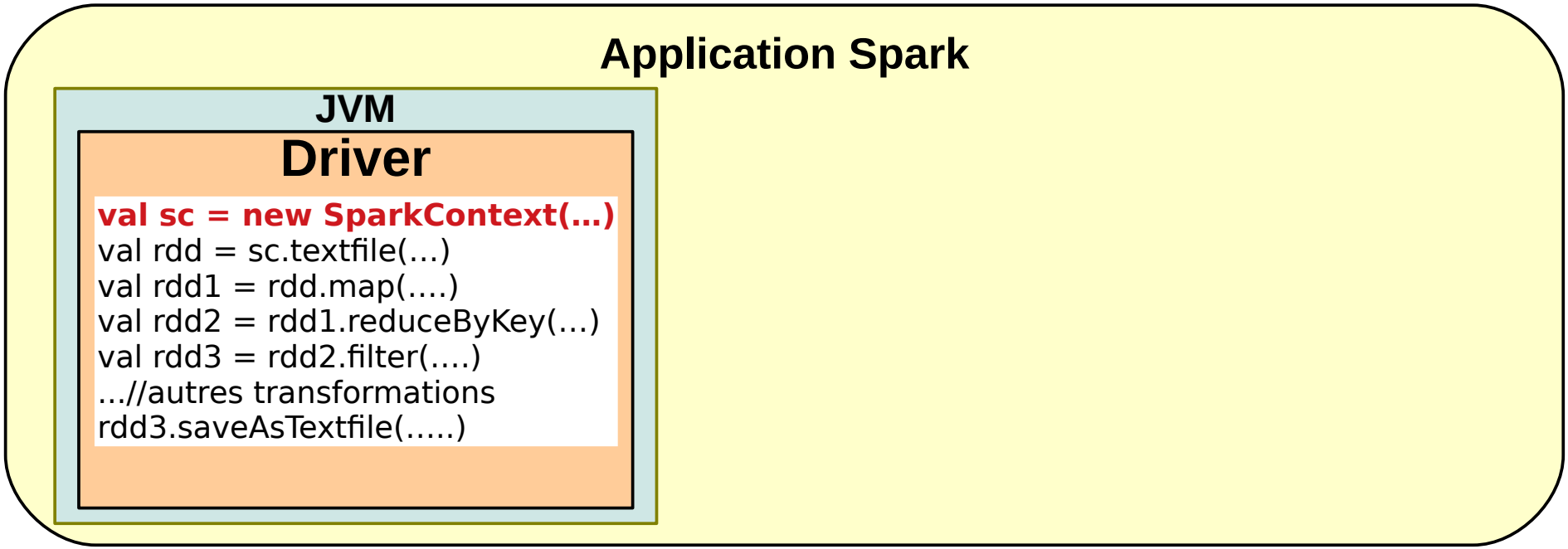
- JVM esclave exécutant les tâches de l'application
- Communication Driver → Executor :
  - affectation de nouvelles tâches
  - annulation de tâches
- Communication Executor → Driver :
  - notification de l'avancement des tâches
- Communication Executor-Executor
  - échange de données entre tâches dépendantes



Exécution d'une appli Spark : étape 1 (sparkcontext)

Initialisation du sparkcontext

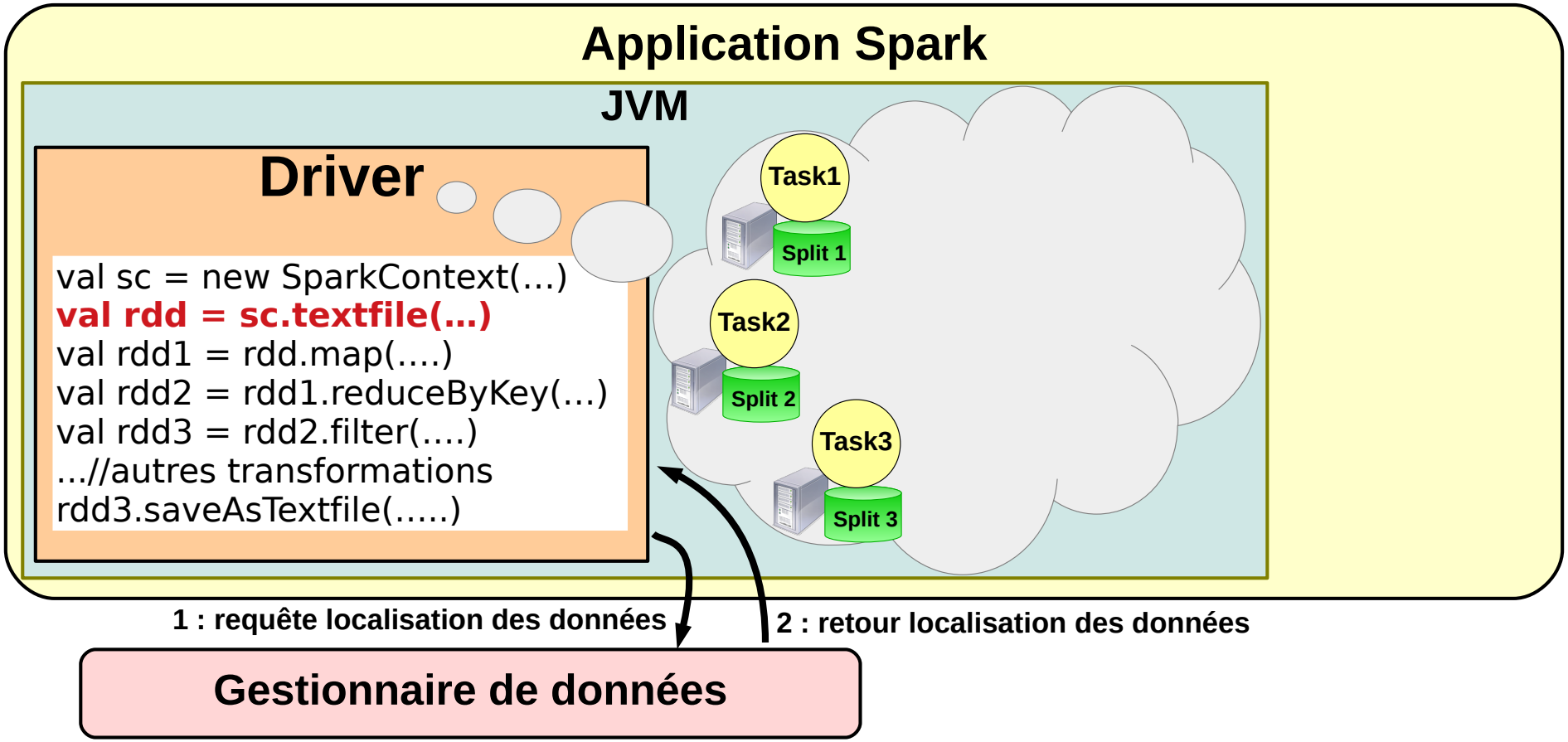
- Prise en compte de la configuration
- Construction des méta-données de l'application



Exécution d'une appli Spark : étape 2 (créations)

Création des premiers RDD

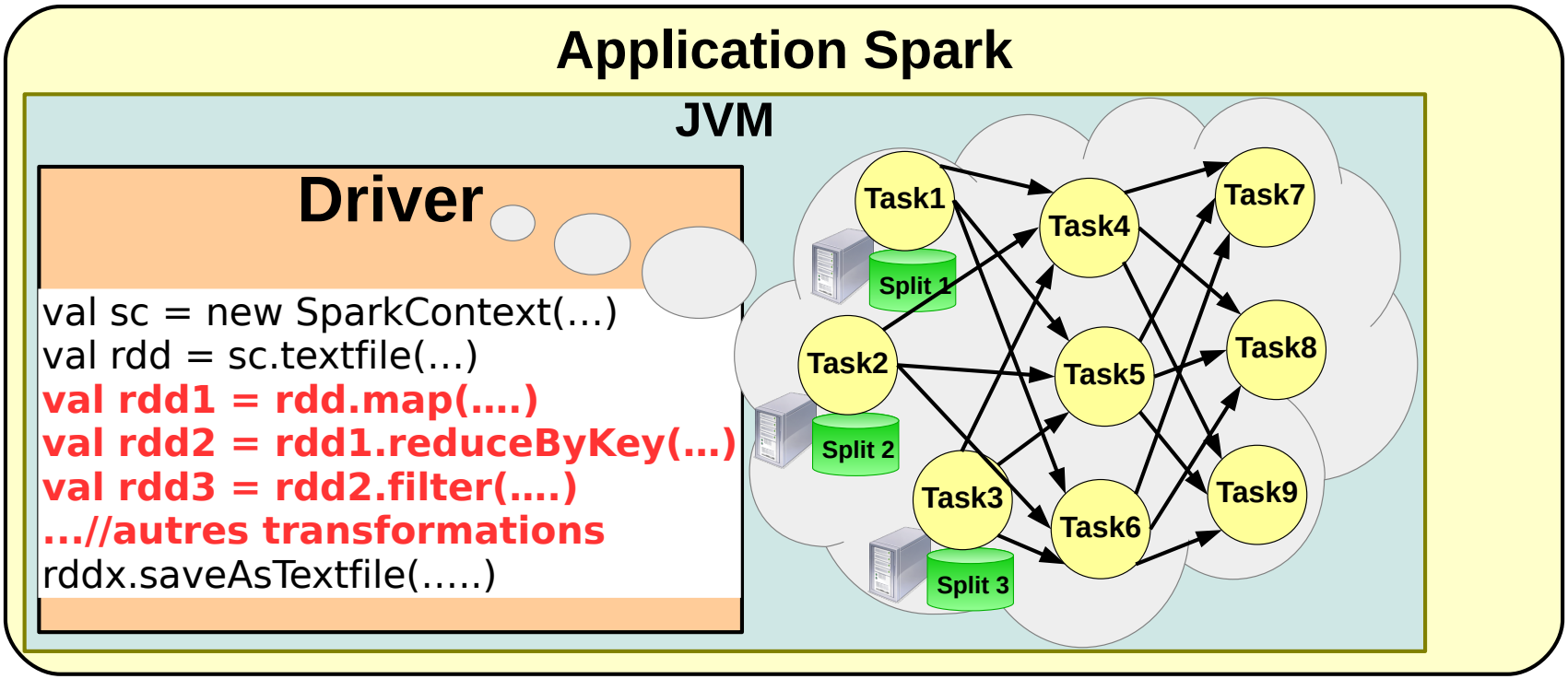
- Détermination des emplacements des tâches racines
- Prise en compte de la localisation des splits (1 tâche par split)



Exécution d'une appli Spark : étape 3 (transformations)

Transformations de RDDs

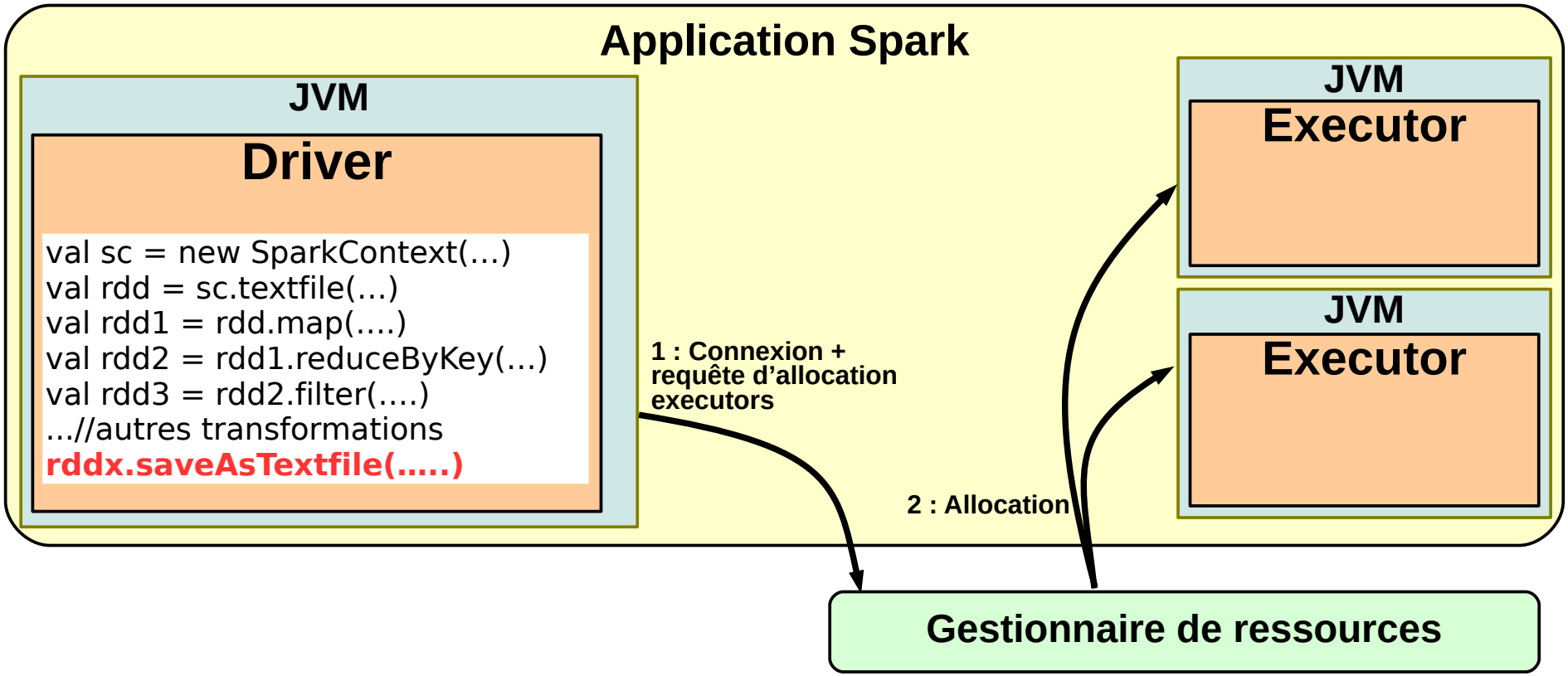
- Traduction des transformations en graphe dirigé acyclique de tâches
- Optimisation des communications inter-tâches



Exécution d'une appli Spark : étape 4 (action)

Exécution d'un Job Spark (1ère partie)

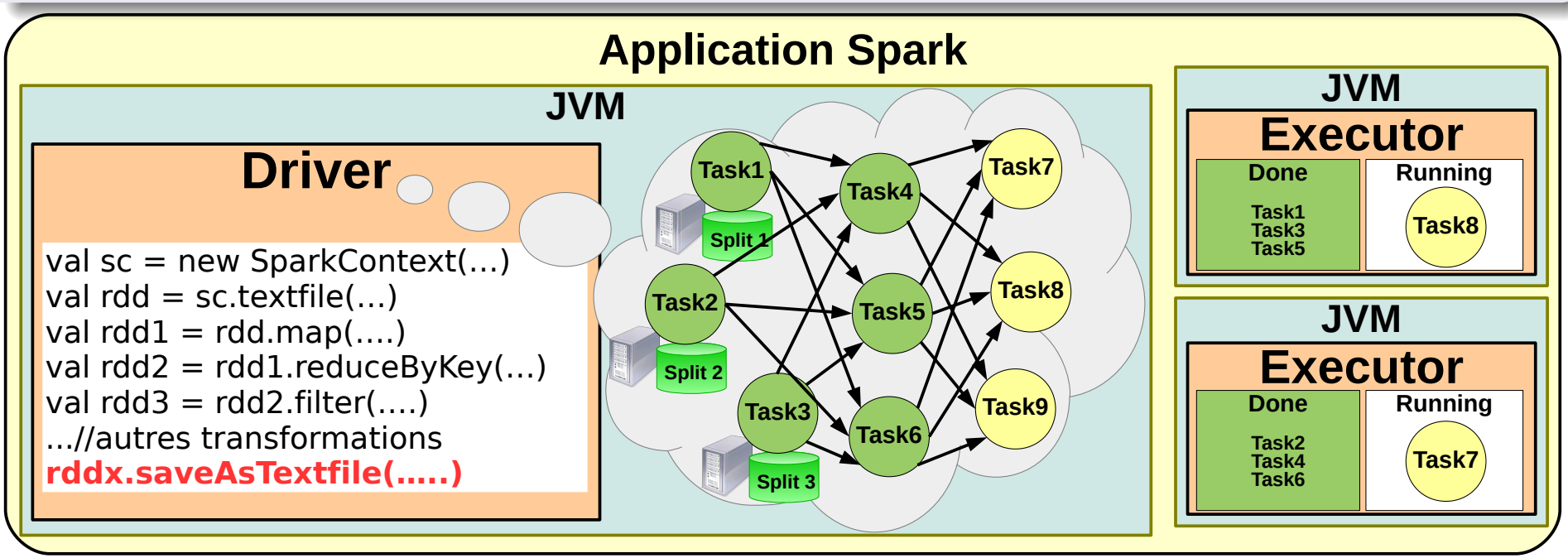
- Connexion au gestionnaire de ressources
- Requête d'allocation des executors sur l'infrastructure



Exécution d'une appli Spark : étape 4 (action)

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 l'hôte hébergeant l'executor (persistance des RDD)



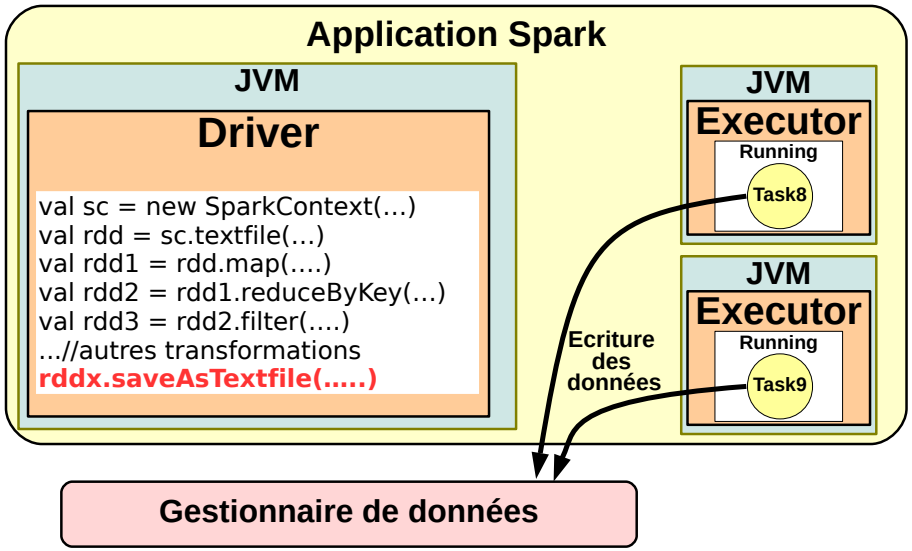
Exécution d'une appli Spark : étape 4 (action)

Exécution d'un Job Spark (3ème partie)

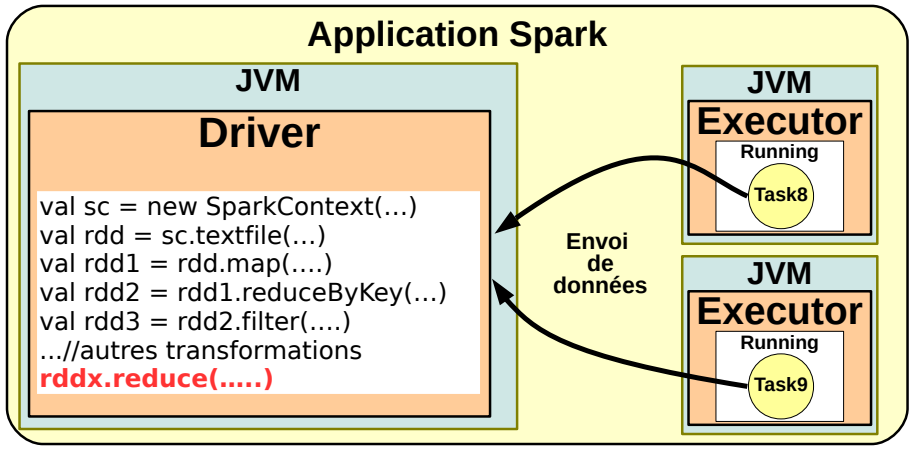
Les données finales du flux sont envoyées :

- 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



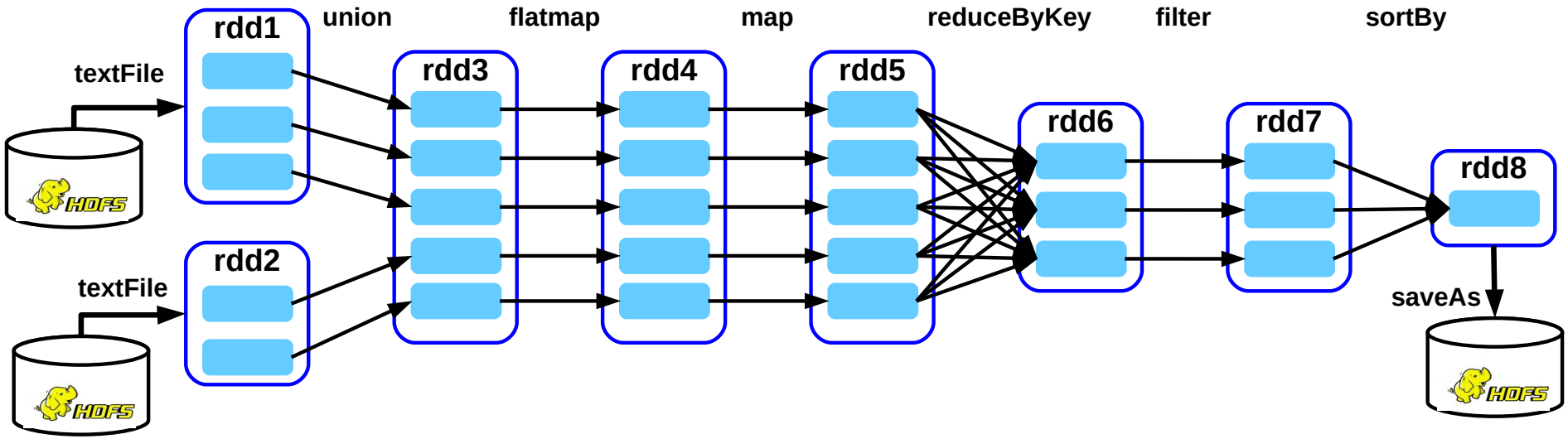
Construction du DAG de tâches

```
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 du DAG de tâches

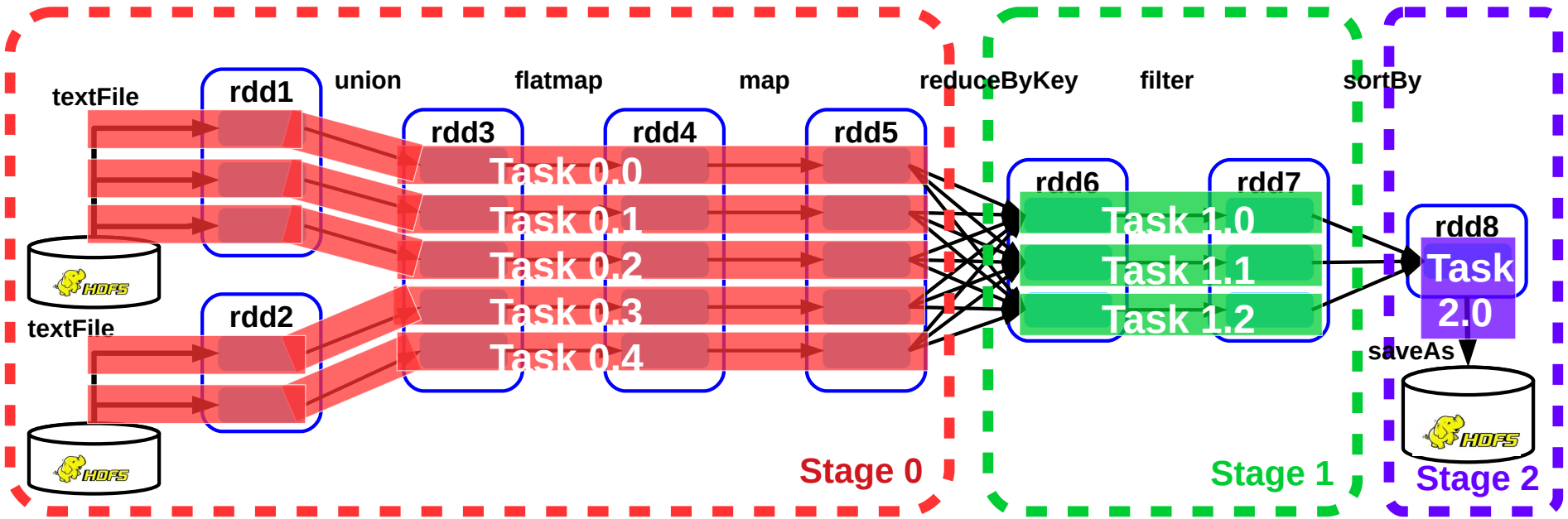
```
val rdd1= sc.textFile("hdfs://namenode/f1")
val rdd2 = sc.textFile("hdfs://namenode/f2")
val rdd3 = rdd1.union(rdd2);
val rdd4 = rdd3.flatMap(_.split(" "))
val rdd5 = rdd4.map((_,1))
val rdd6 = rdd5.reduceByKey(_+_, 3)
val rdd7 = rdd6.filter(_._2 > 1)
val rdd8 = rdd7.sortBy(_._2, true, 1)
rdd8.saveAsTextFile("hdfs://namenode/out")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



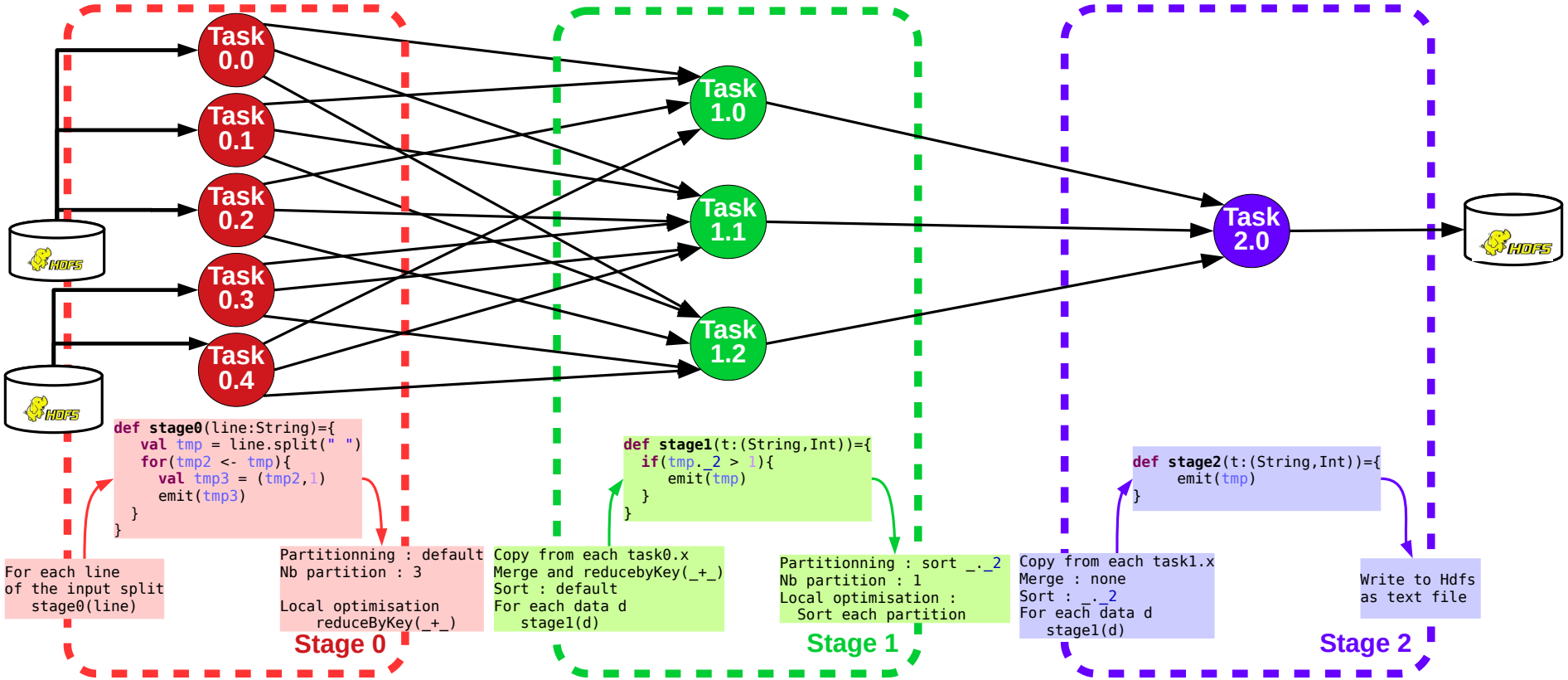
Construction du DAG de tâches : les stages

Définition

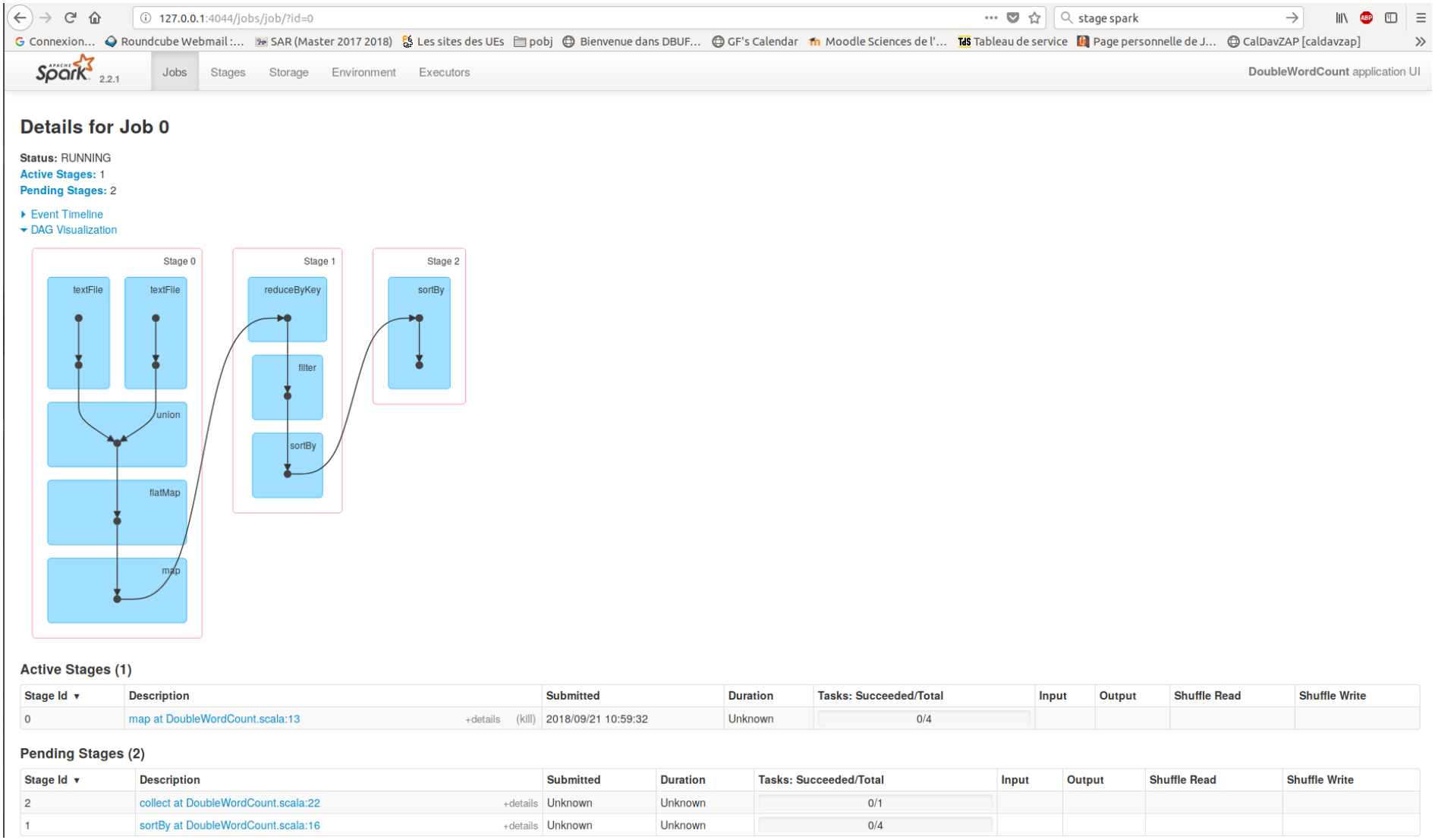
- Enchaînement continu de transformations simples de RDD compris entre
  - soit deux transformations shuffle
  - soit une transformation de shuffle et une action
- Définit un ensemble de tâches indépendantes et parallèle exécutant le même code mais sur des partitions différentes



Construction du DAG de tâches : exemple



Suivi d'exécution d'un job via l'interface Web



Qu'affiche ce programme ?

```
val conf = new SparkConf().setAppName("Essai")
val sc = new SparkContext(conf)
val data = Array(1, 2, 3, 4, 5)
var counter = 0
var rdd = sc.parallelize(data)
rdd.foreach(x => counter += x)
sc.stop()
println("counter = " + counter)
```

Réponse

```
counter = 0
```

Les types d'objet partagés de Spark

Broadcast variable

- Permet de copier une variable immutable sur chaque machine en utilisant un algorithme de diffusion
- La diffusion se fait au moment d'exécuter le stage qui l'utilise

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))

 broadcastVar.value // permet d'accéder à la variable
```

Accumulateur

- Variable qui peut uniquement s'incrémenter
- Créée à partir d'un valeur initiale et s'incrémente avec la méthode `add`
- Seul le driver peut y accéder en lecture

```
val accum = sc.longAccumulator("My␣Accu")
sc.parallelize(Array(1, 2, 3, 4)).foreach(accum.add(_))
accum.value // permet d'accéder à la valeur de l'accumulateur
```

Déploiement d'une application Spark

Déploiement local

- Exécution locale sur un seul processus que l'on peut multi-threader
- Tests et debugages de programmes
- Gestionnaires de données = système de fichiers local (URL = `file://...`)

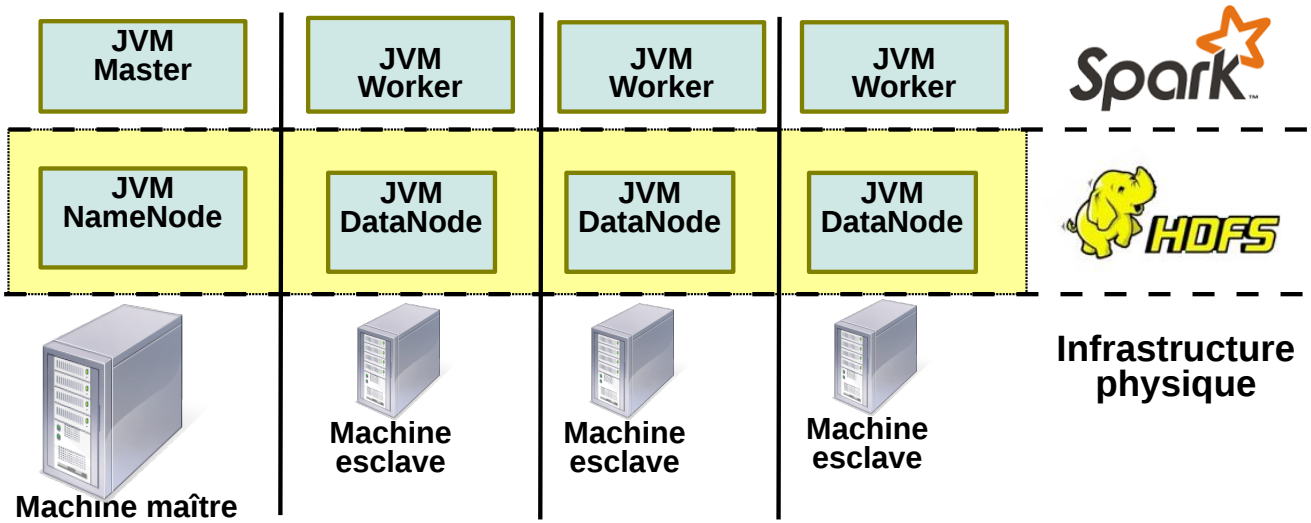
Déploiement distribué

- Gestionnaires de ressources de calcul :
  - **Spark Standalone** : le gestionnaire de Spark
  - **Mesos** : gestionnaire distribué de conteneur (Apache)
  - **Yarn** : gestionnaire de ressources de Hadoop
  - **Amazon EC2** : cloud IaaS d'Amazon
  - **Kubernetes** : gestionnaire distribué de conteneur (Google)
- Gestionnaires de données :
  - HDFS (URL = `hdfs://namenode:port/...`)
  - Amazon S3
  - SGBD

Gestionnaire Spark standalone

Une architecture maître esclave

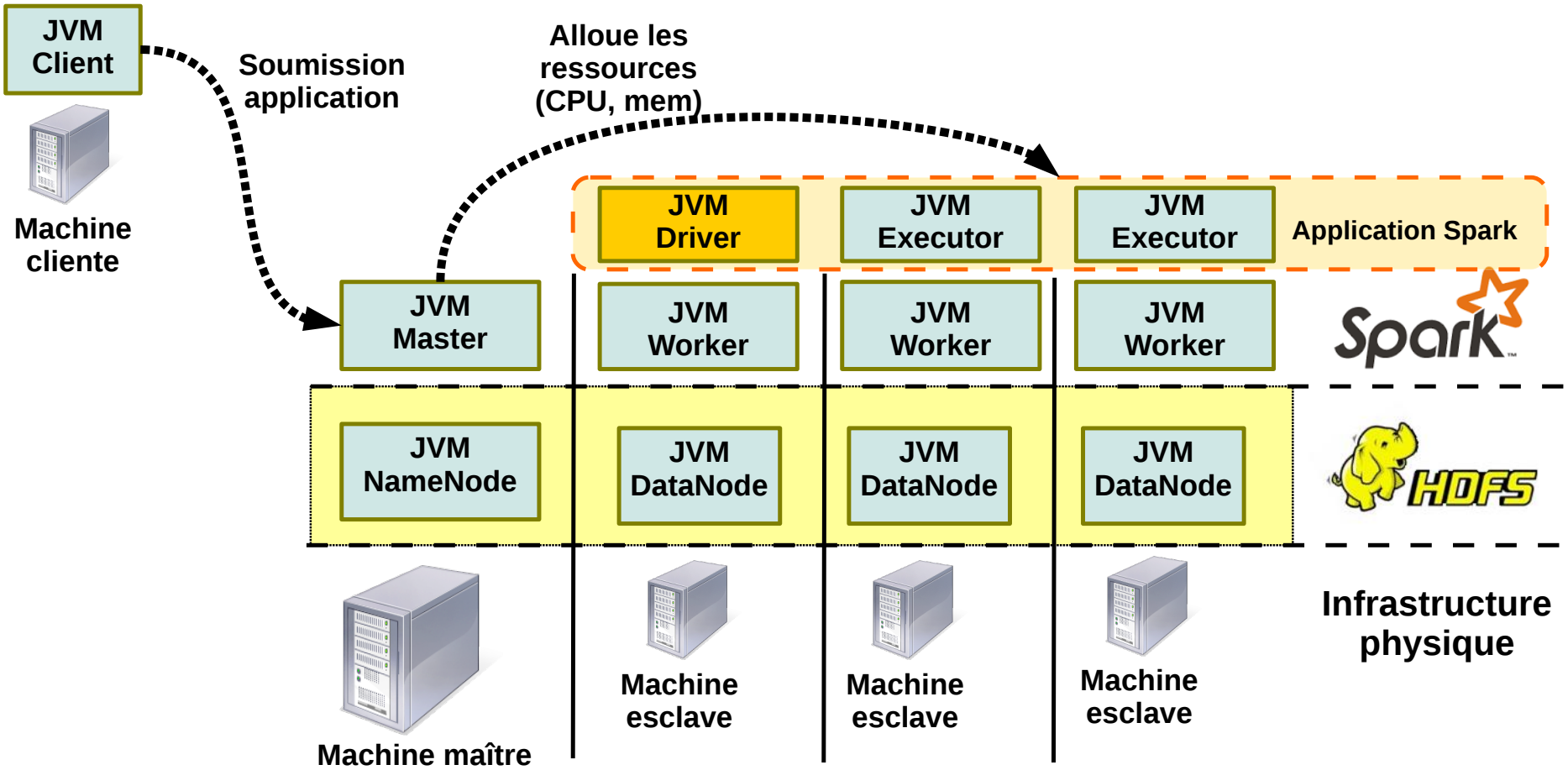
- **Spark Master** :
  - Reçoit les soumissions d'applications
  - Surveille les workers
  - Affecte les executors
- **Spark Worker** :
  - Déploie les tâches localement et surveille les executors



Gestionnaire Spark standalone : mode cluster

Principes

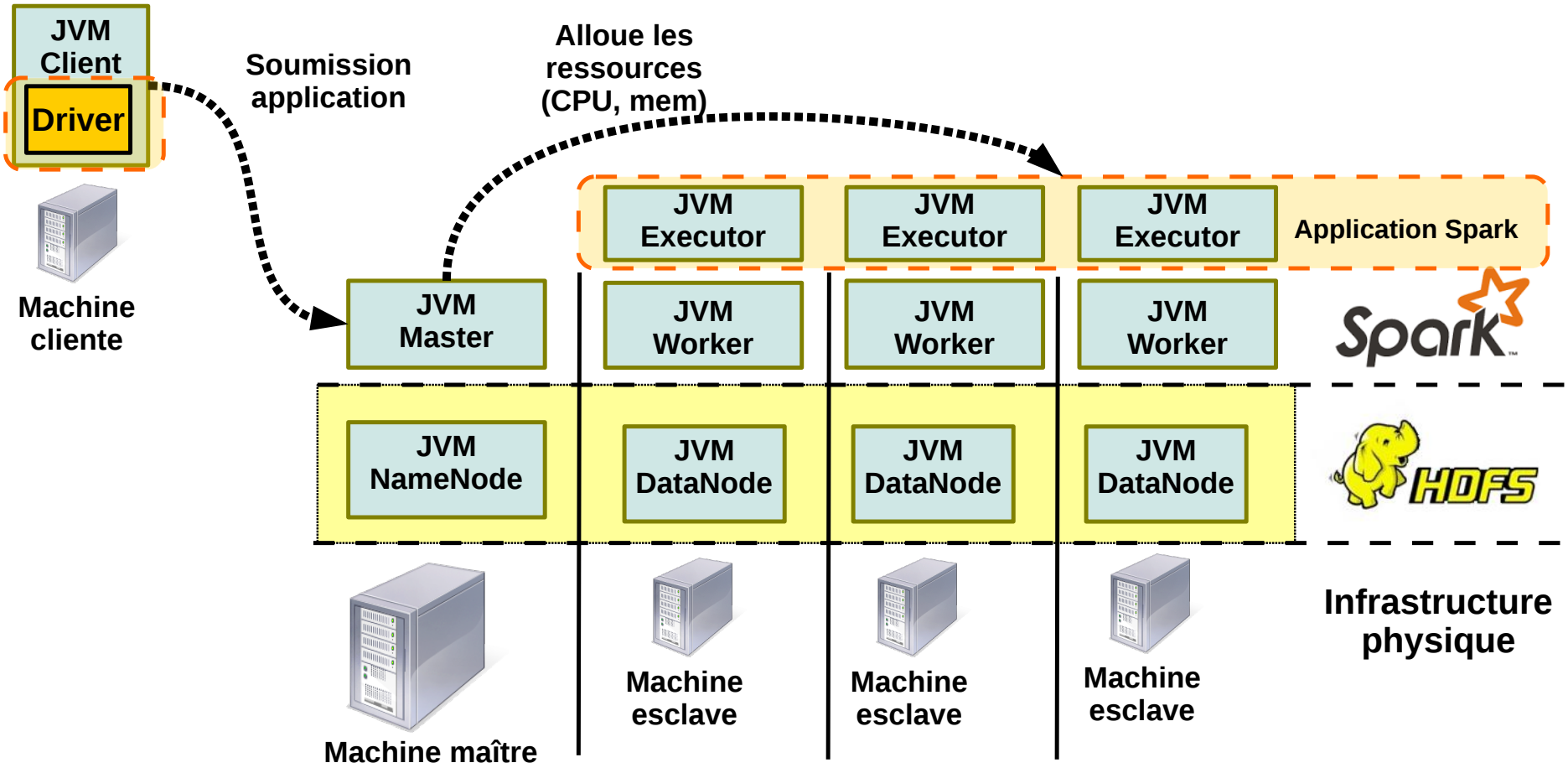
- Le driver s'exécute sur une machine esclave du cluster
- L'application est soumise de manière asynchrone



Gestionnaire Spark standalone : mode client

Principes

- Le driver s'exécute sur la machine physique du client
- Le client attend la fin de l'application et interagit avec les executor

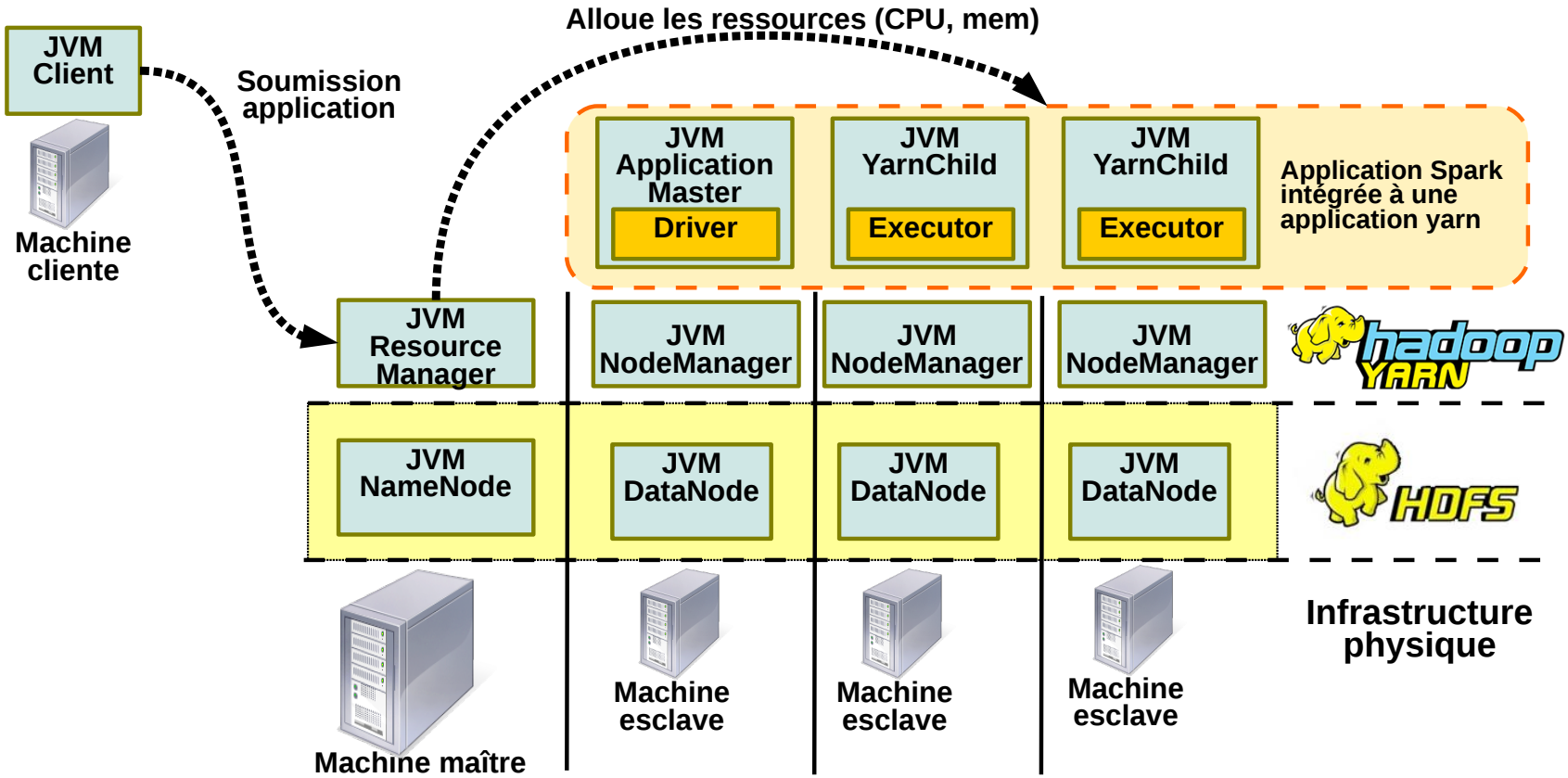


Gestionnaire Yarn

Principes

- Le conteneur Application Master héberge le driver
- Les conteneurs YarnChild hébergent les executors

⇒ Spark et Hadoop MR peuvent cohabiter sur le même gestionnaire



Spark vs. Hadoop Map-Reduce

		
Langage natif	Java (120 000 lignes)	scala (30 000 lignes)
Langages supportés	Tout langage via stdin/stdout	Scala, python, Java, R
Analyse temps réel Streaming	non	oui
Vitesse de traitement	lent dû aux nombreuses I/O	tout se fait en mémoire $\Rightarrow$ 10 à 100 fois + rapide
Programmation	lourde et verbeuse	simple, API haut niveau
Abstraction de mémoire	aucune	RDD
Tolérant au panne	oui	oui
Expressivité de flux complexes	nécessite un scheduler externe de job (Oozie)	Tout s'exprime dans la même application
Dépendances avec d'autres système	Fortes : écosystème Hadoop riche et complexe	Faibles : Spark est un tout en un (sauf stockage)
Coût de fonctionnement	faible (mémoire disque)	plus onéreux (mémoire RAM)
Communauté	de moins en moins nombreuse	forte, projet top-level