Apache Spark : une plate-forme de traitement de données à large échelle

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

CODEL – Master 2 SAR 2017/2018

Encore un nouvel outil ?

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

Pas d'abstraction d'une mémoire partagée

- réutilisation difficile des données intermédiaires
  $\Rightarrow$ inadapté pour des algos de machine learning ou de graphe
- Ecriture des données intermédiaires sur un disque
  $\Rightarrow$ Beaucoup d'entrée/sortie inutiles

Hadoop Map-Reduce

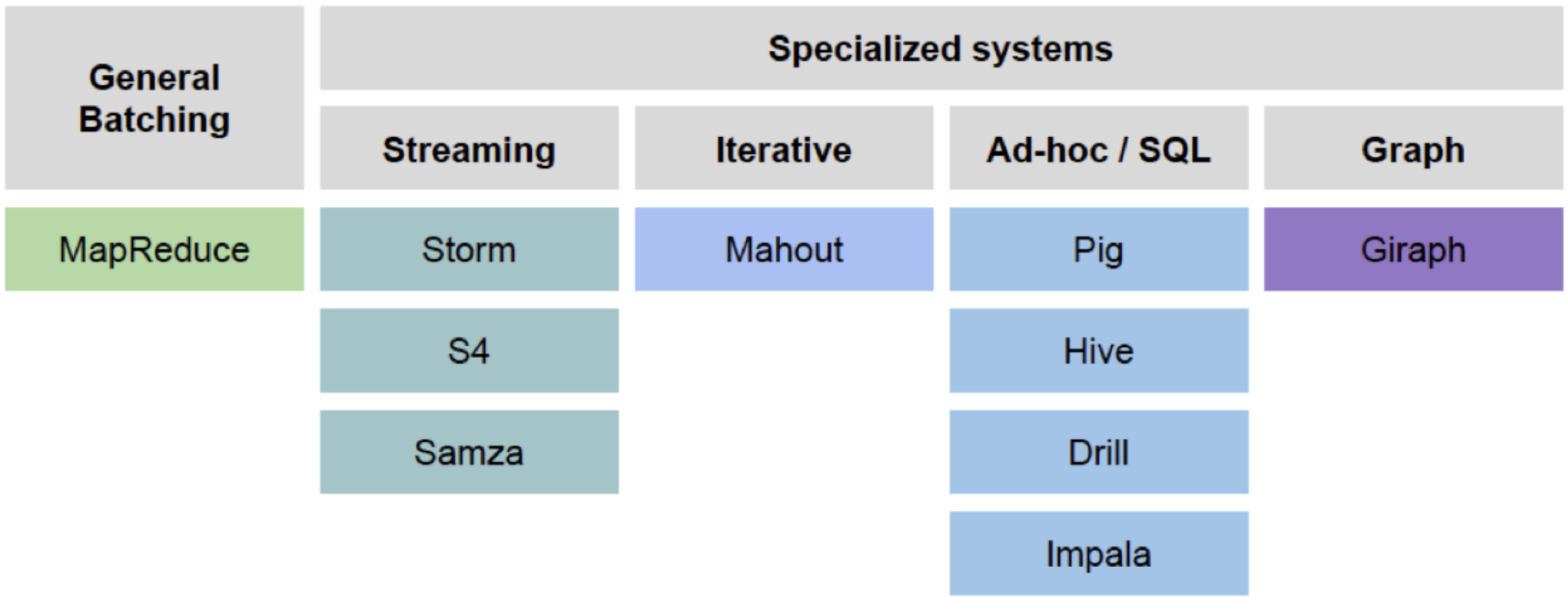
Langage Java

Peu adapté à la programmation fonctionnelle :

$\Rightarrow$ Expression de traitement générique difficile et verbeux

Un large écosystème diversifié

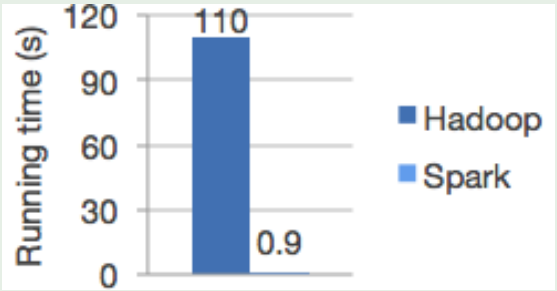
Vision peu unifiée, interactions coûteuses entre logiciels



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

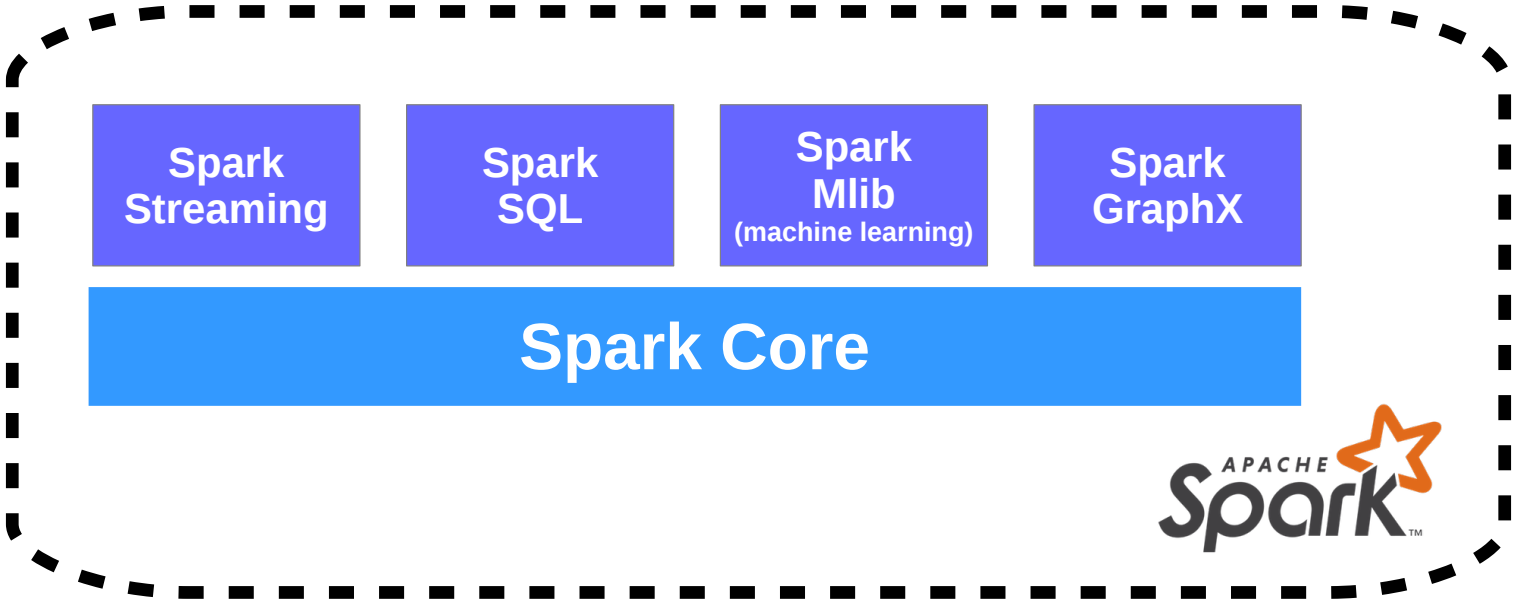


- Une généralisation du Map-Reduce avec une approche fonctionnelle
- Une abstraction des données : **Resilient Distributed Dataset (RDD)**
- Une API pour Scala, Java, Python et R
- Un déploiement sur cluster dédié ou sur cluster Hadoop Yarn
- Pas de système de stockage propre
  mais très interfaçable avec HDFS, NFS, S3, etc..

Spark : qu'est ce que c'est ?

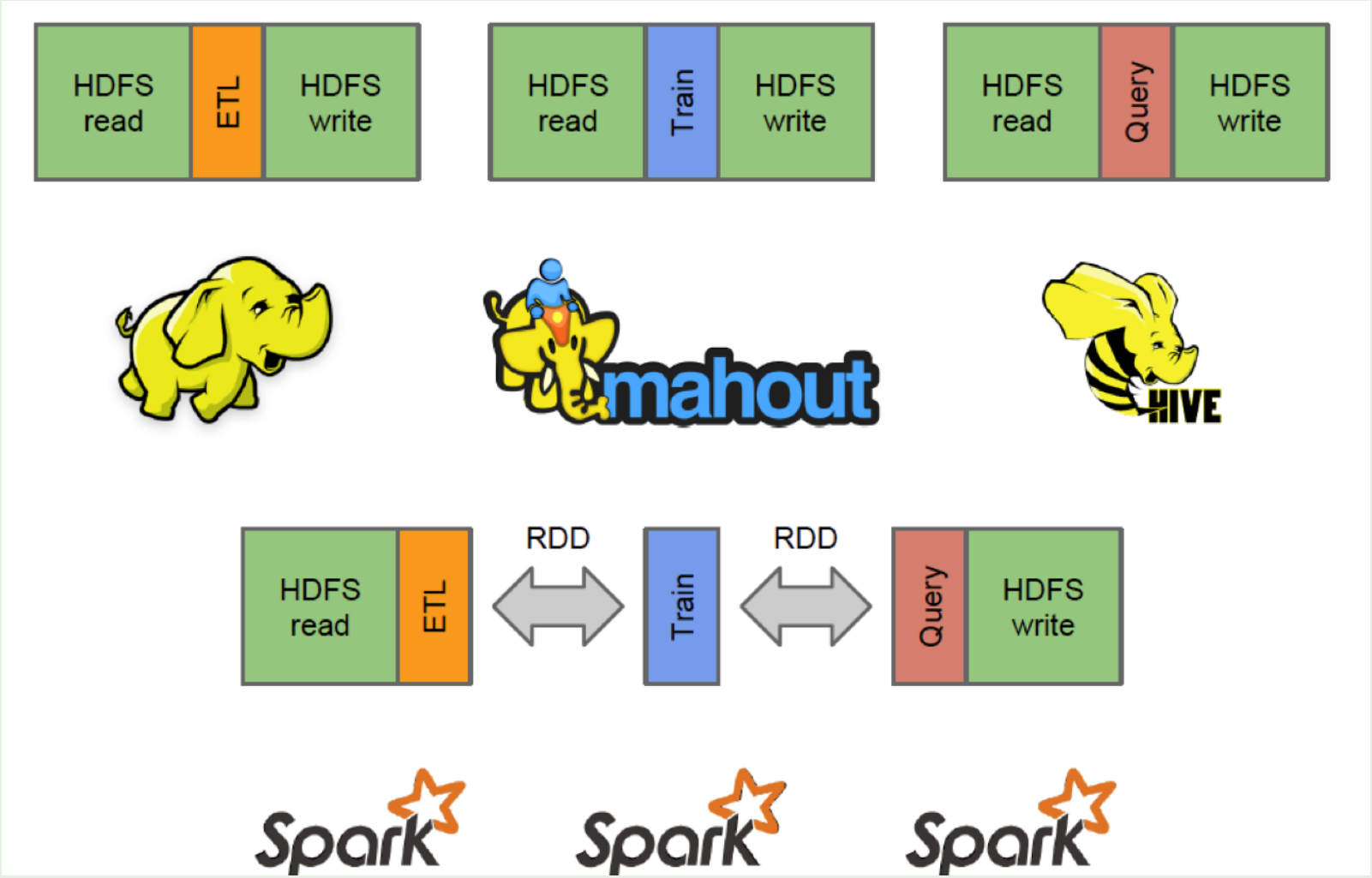
Une plate-forme unifiant plusieurs librairies

- **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



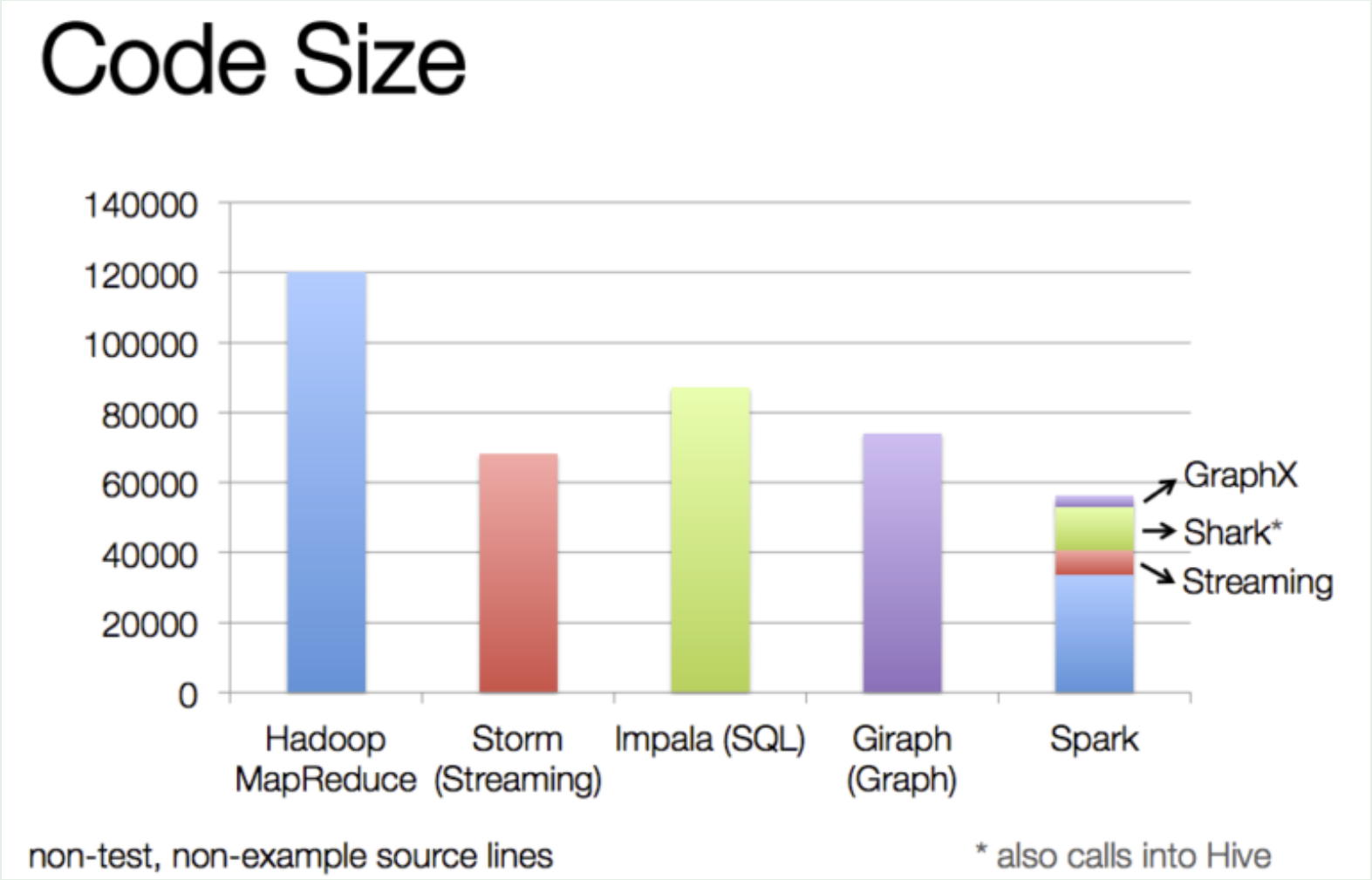
Spark : qu'est ce que c'est ?

Un flux de données simplifié



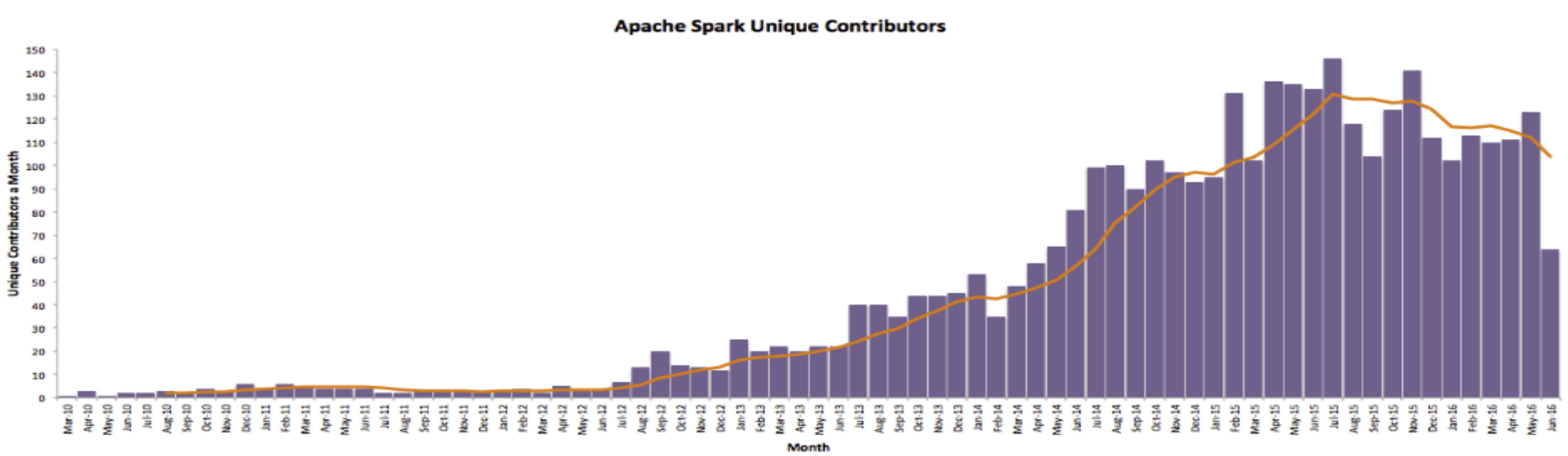
Spark : qu'est ce que c'est ?

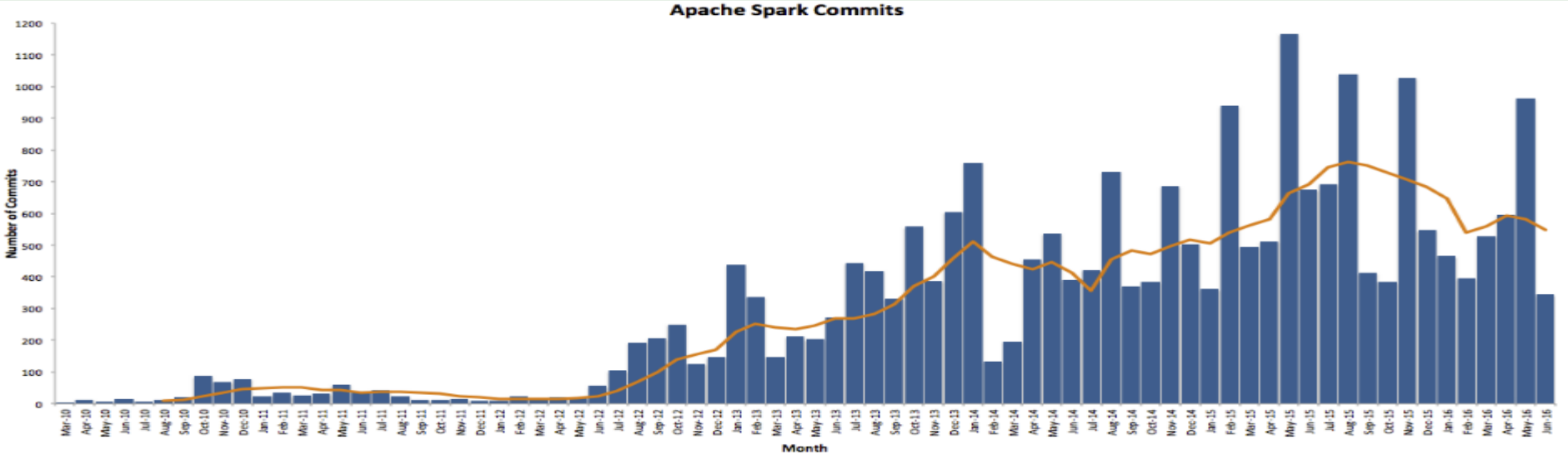
Une plate-forme écrite en Scala



Spark : qu'est ce que c'est ?

Une plate-forme très à la mode





Spark : qu'est ce que c'est ?

Un bref historique

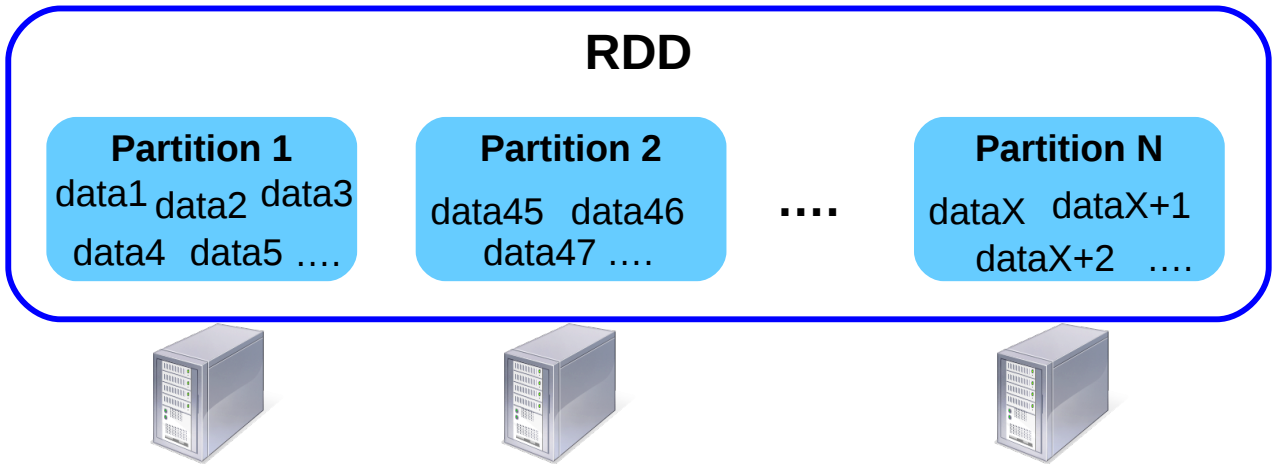
- 2009 : conception initiale par Matei Zaharia en doctorat à Berkeley University.
- 2013 : reprise par la fondation Apache, devient l'un des projets les plus actifs
- 2014 : Détrône Hadoop Map-Reduce en battant le record tu tri le plus rapide de 100 To
  - Hadoop Map-Reduce : 72 minutes avec 2100 machines
  - Spark : 23 minutes avec 206 machines
- 2015 : plus de 1000 contributeurs venants de 200 entreprises
- Décembre 2017 : version 2.2.1

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)
- créée que par des opérations déterministes.
- avec un niveau de persistance



RDD vs. Distributed Shared Memory

Définition **Distributed Shared Memory**

Un espace d'adressage global, où les applications lisent et écrivent de manière aléatoire

Points forts du RDD sur la DSM

- RDD ⇒ immutabilité :
  - ⇒ on ne peut que créer, pas modifier
  - ⇒ réplication de tâches et de données facilitées
- RDD ⇒ Écriture macros
  - ⇒Adapté pour l'écriture en bloc très utiles en Big Data

Points forts de la DSM sur le RDD

- DSM ⇒ mutabilité :
  - ⇒Adapté aux applications qui doivent mettre à jour un état partagé

L'API des RDD

Déclaration

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

API des Méta-données internes

- 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

L'objet SparkContext

Définition

Point d'entrée principal pour les fonctionnalités de Spark :

- Connexions avec le cluster Spark
- Stockage des méta-données d'un job Spark (ex : Configuration)
- Création de RDD
- Création d'accumulateurs ou variables de diffusion

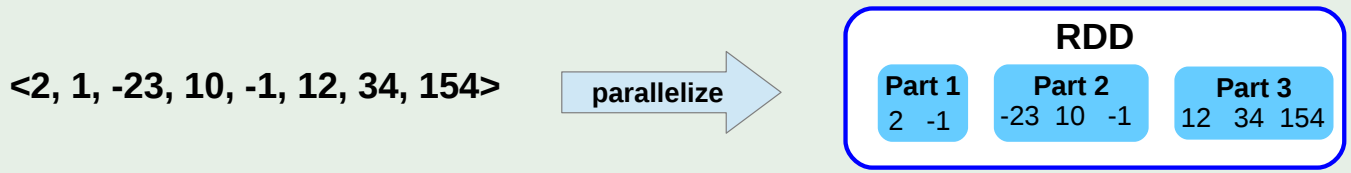
```
object MonProgSpark extends App {

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

Création de RDD à partir d'un SparkContext

Depuis une collection existante en mémoire du programme client

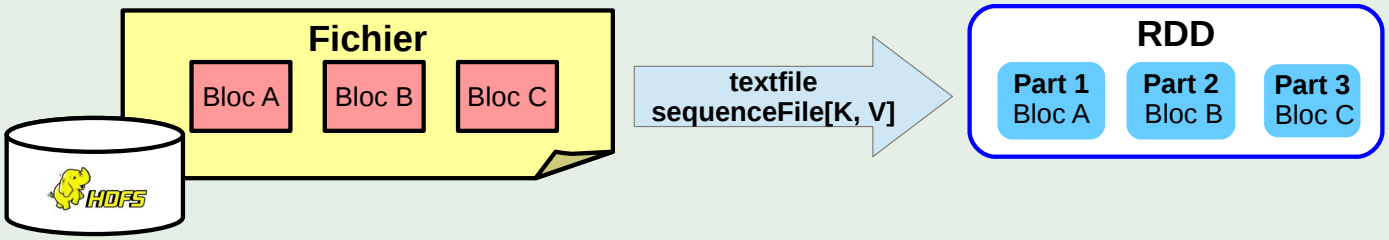
```
def parallelize[T](seq: Seq[T], numSlices: Int): RDD[T]
```



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]
```



Transformations de RDDs

Caractéristiques

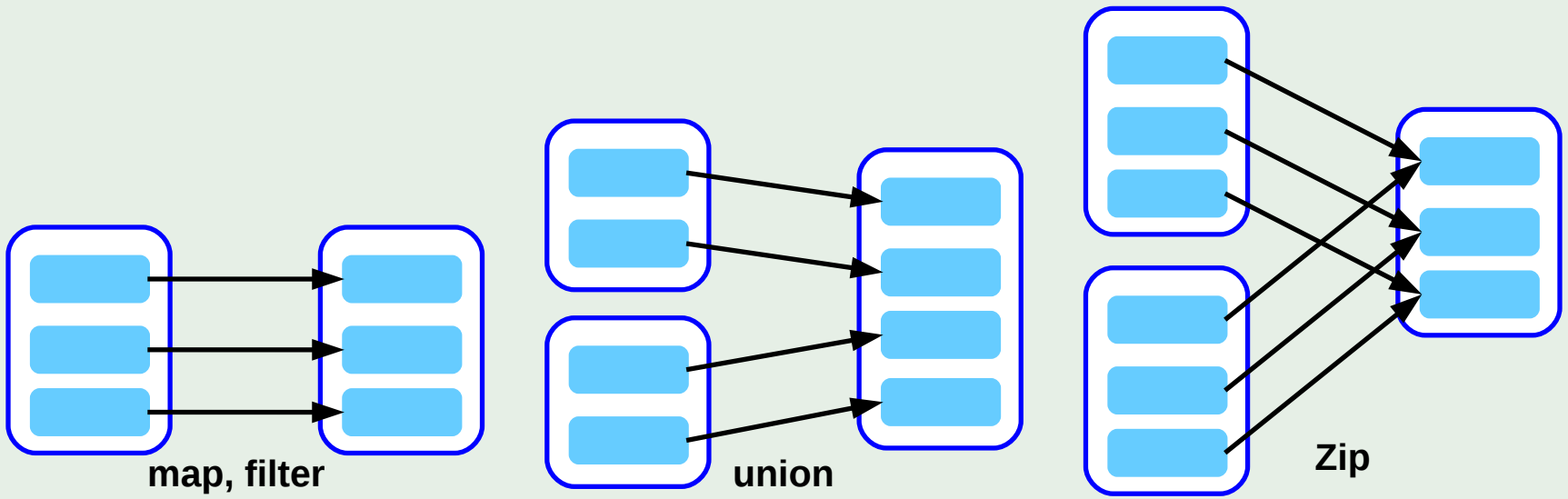
- Permet de décrire une fonction de transition entre un RDD parent et un RDD fils.
- Étape de transition décrivant un flux de données
- Exécution paresseuse : permet des optimisations avant l'exécution

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
- pas besoin de synchronisation pour passer du RDD parent au RDD fils

Exemples :

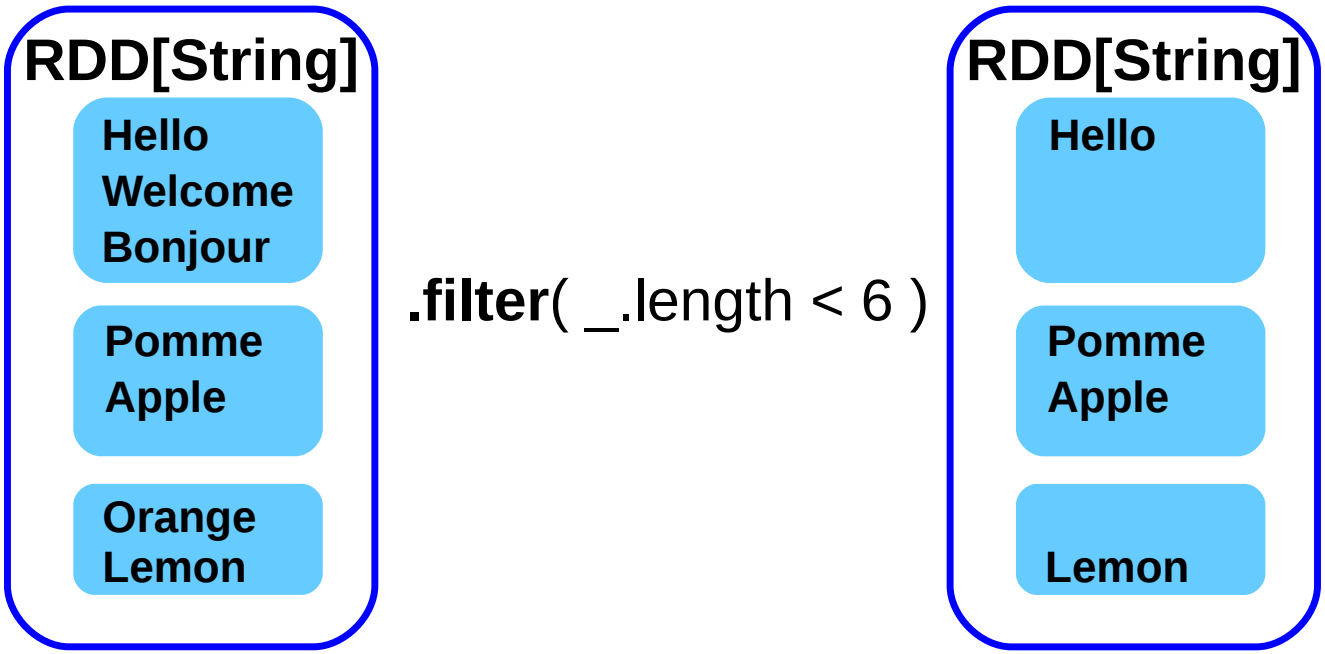


Transformation étroite `filter`

Spécification

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

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

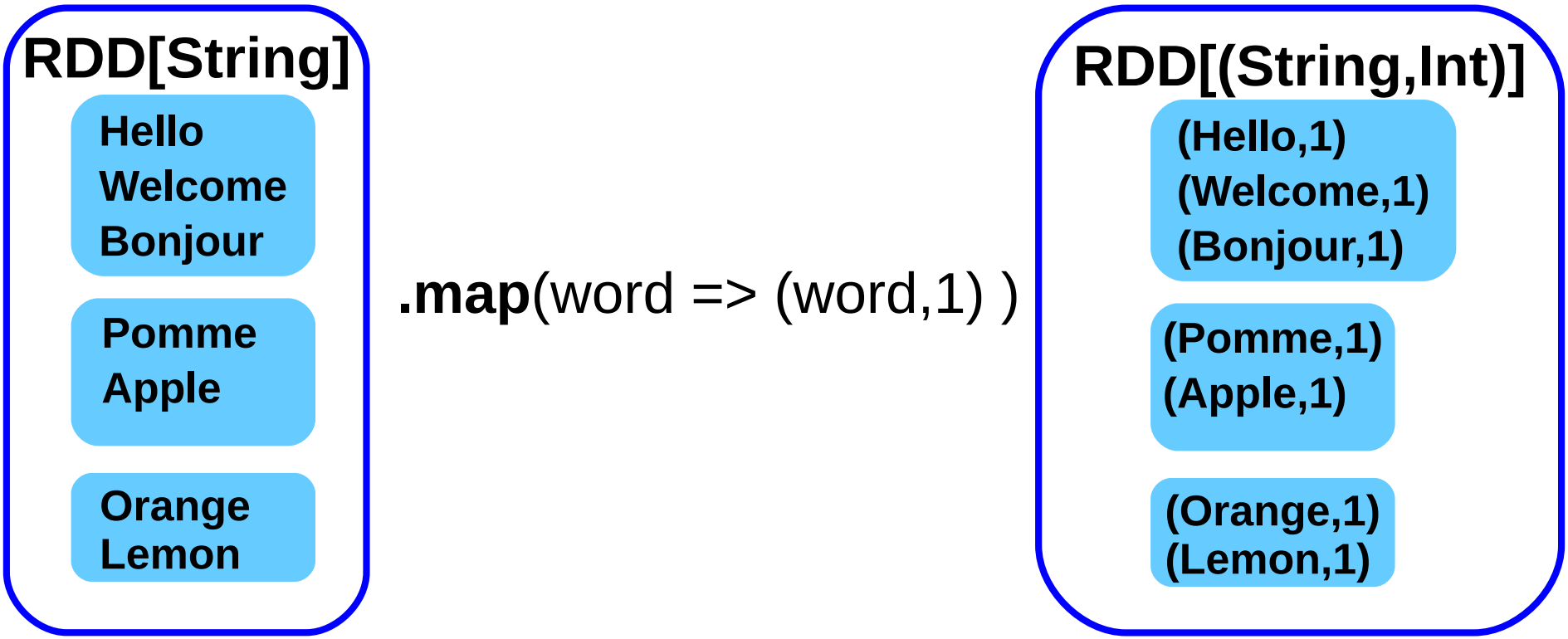


Transformation étroite map

Spécification

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

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

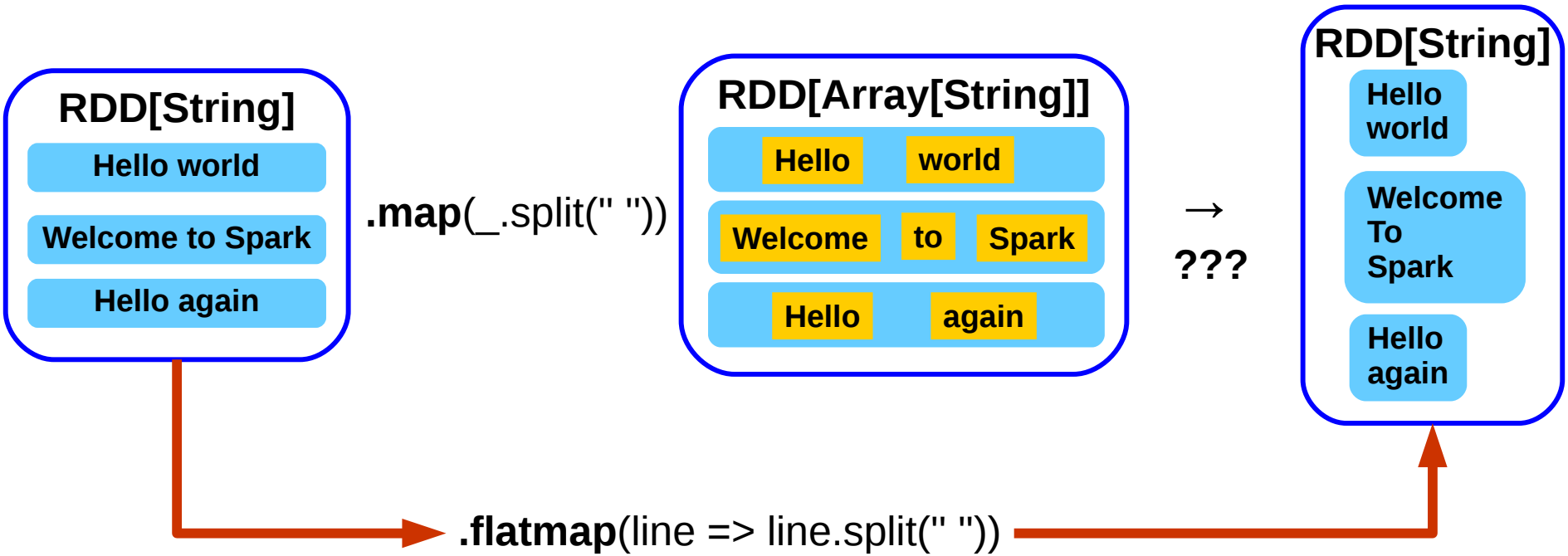


Transformation étroite `flatmap`

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]
```

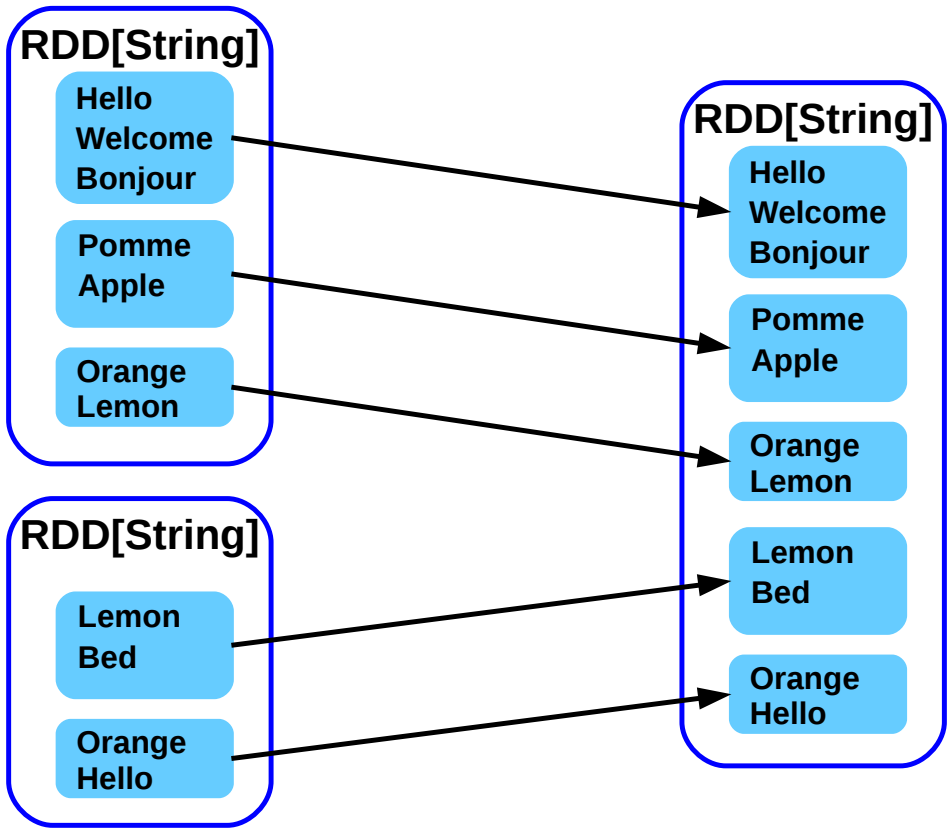


Transformation étroite Union

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

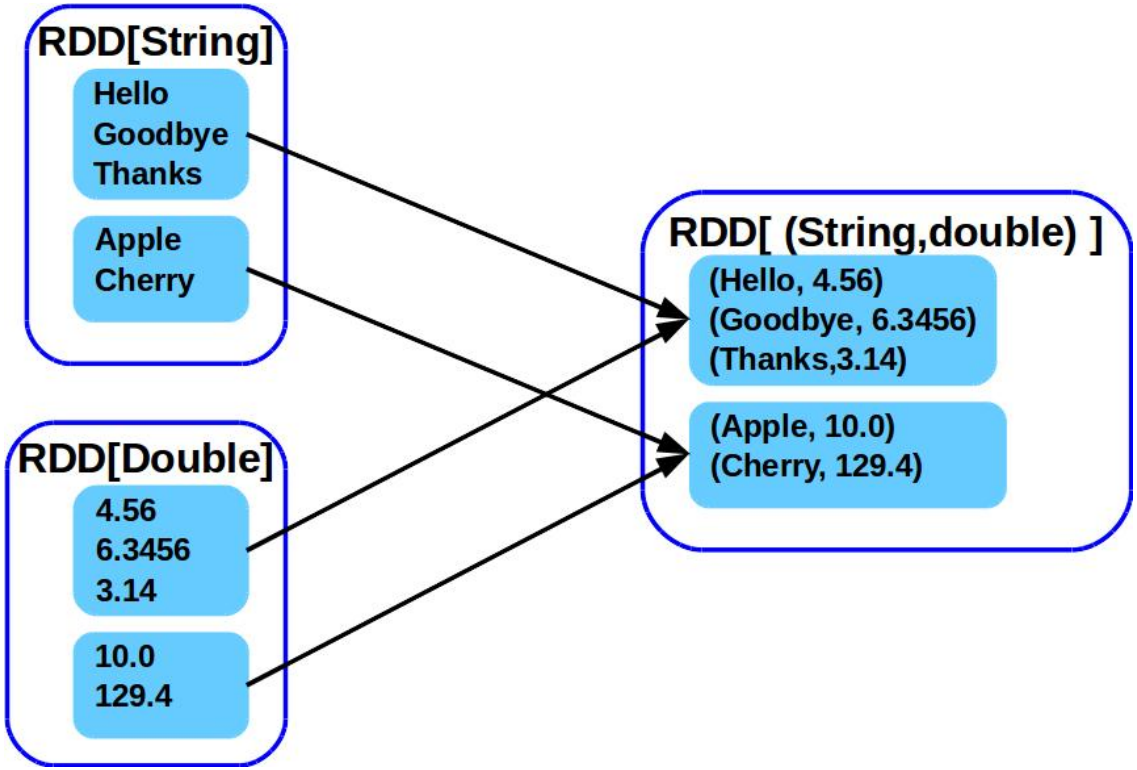
Lie deux RDD en créant un RDD de couples clé-valeur, où le *n*ième couple est l'association des*n*ième éléments de chaque RDD.

```
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

Transformation étroite Zip



Variantes

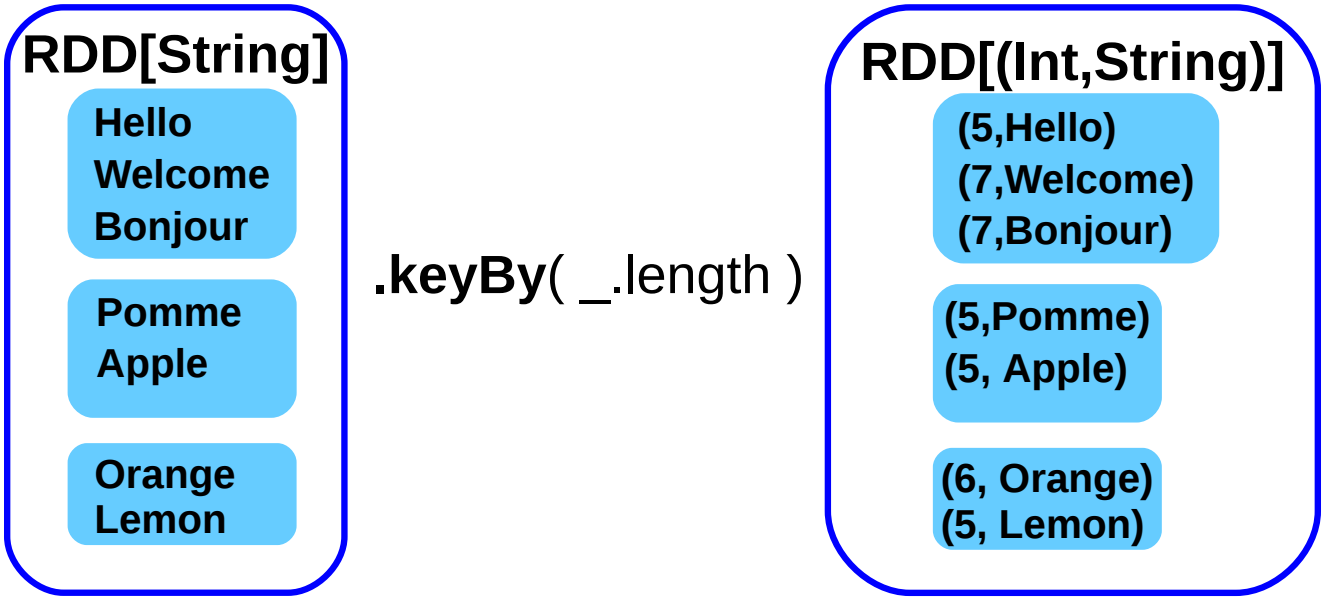
- `def zipWithIndex(): RDD[(T, Long)]` : relie chaque élément avec son indice dans le RDD.
- `def zipWithUniqueId(): RDD[(T, Long)]` : relie chaque élément avec un identifiant

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)]
```

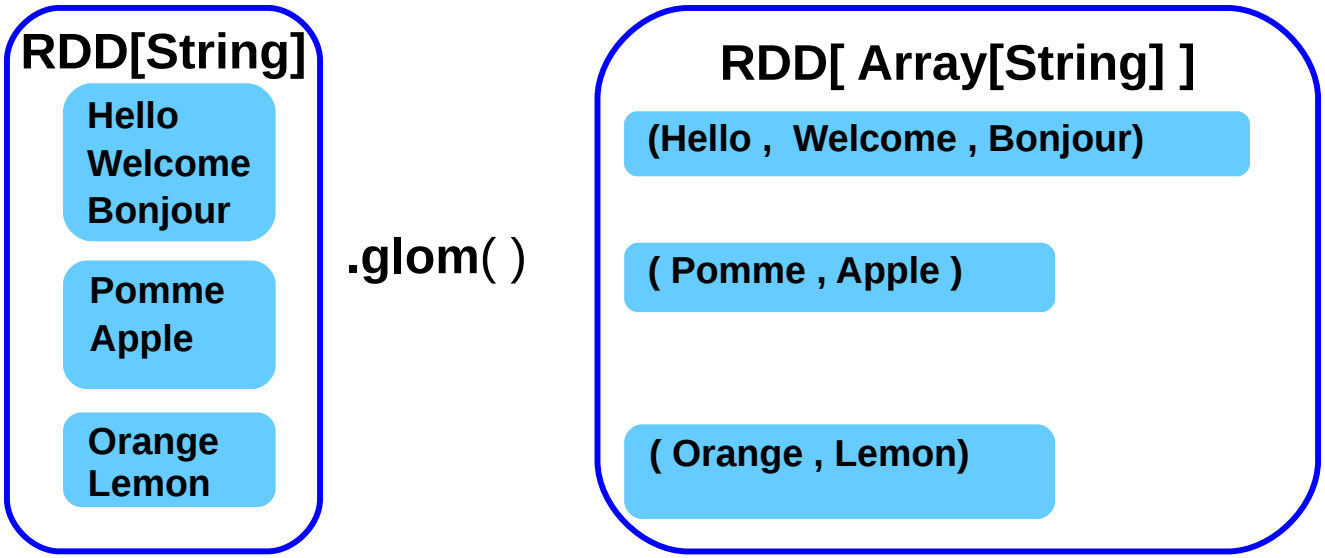


Transformation étroite `glom`

Spécification

Retourne un nouvel RDD créé par l'agrégation de tous les éléments d'une partition en un tableau

```
def glom(): RDD[Array[T]]
```



Dépendances larges entre RDDs

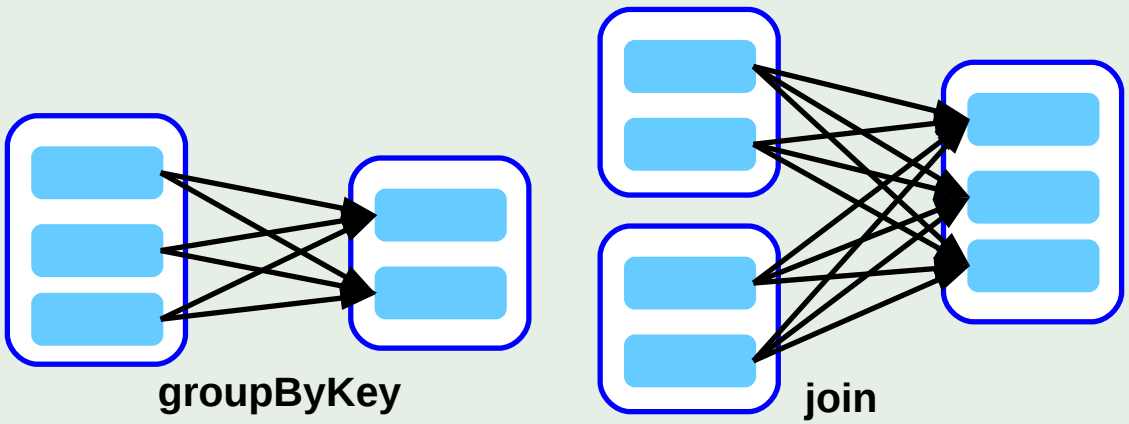
Une relation all to all (= Shuffle)

- 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 :

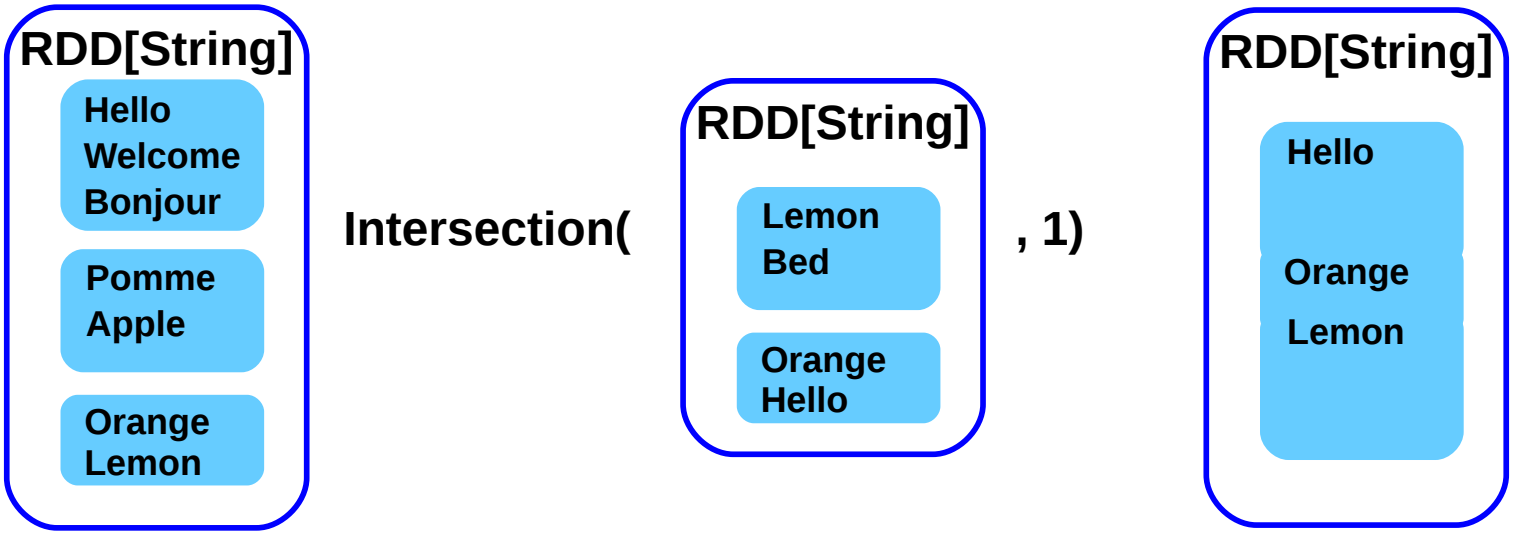


Transformation large `Intersection`

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]
```

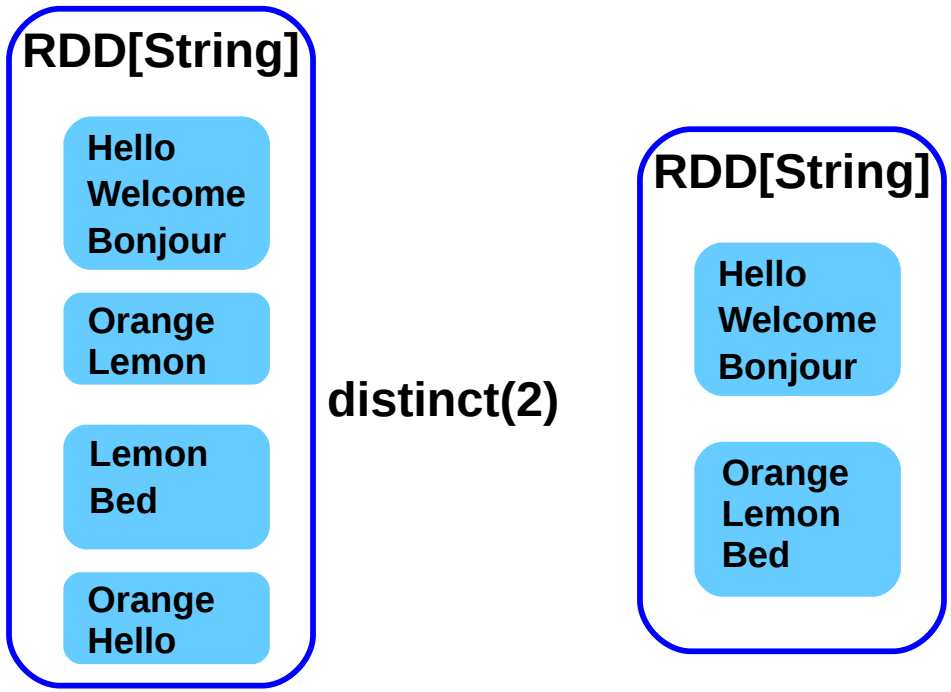


Transformation large `distinct`

Spécification

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

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

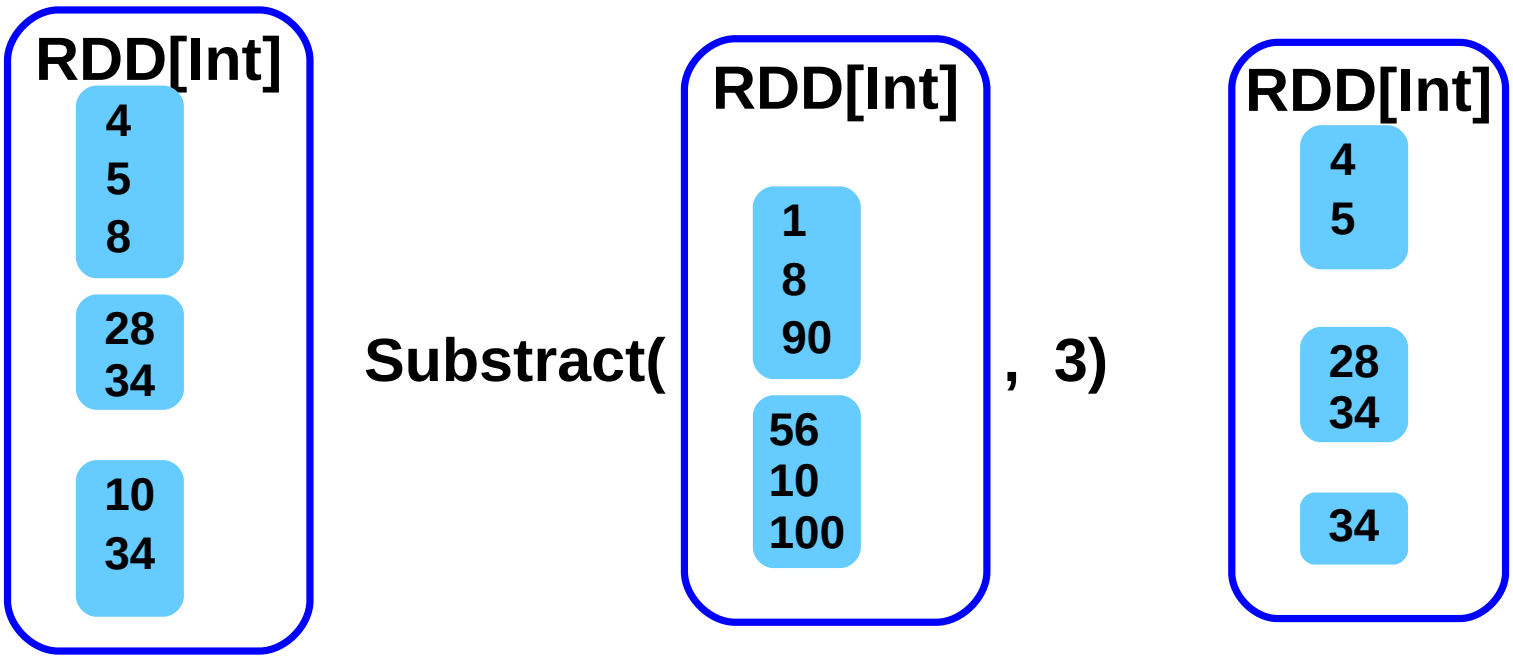


Transformation large `subtract`

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]
```

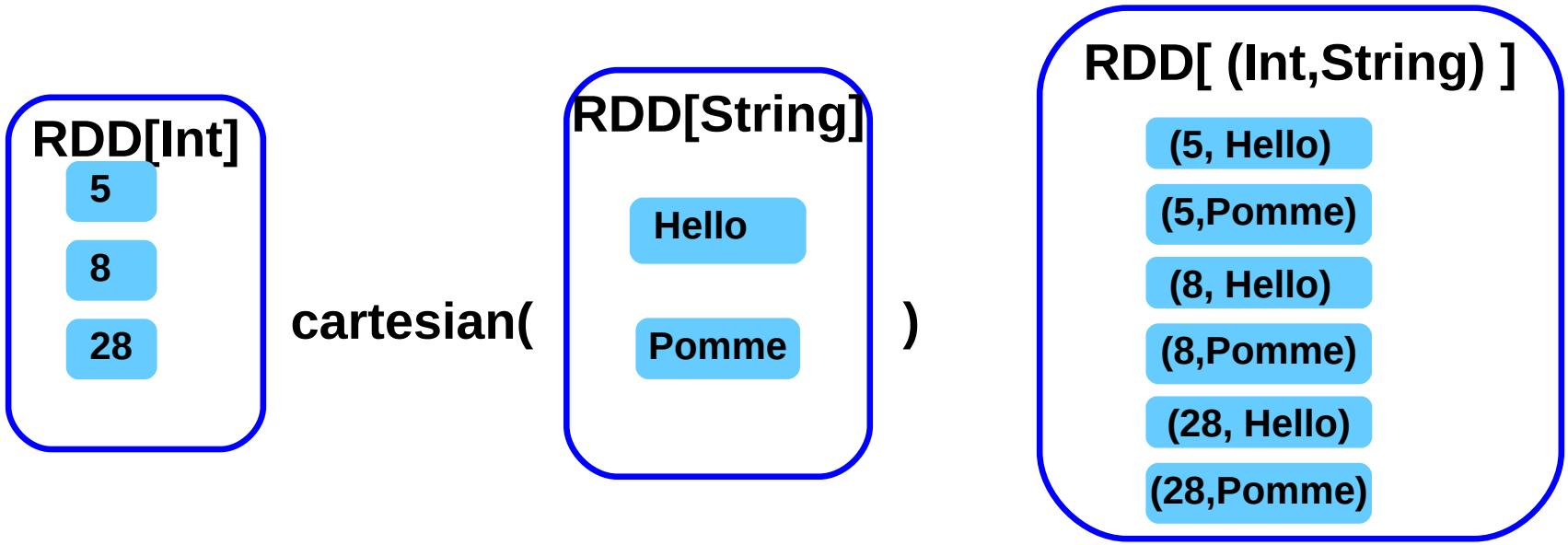


Transformation large `cartesian`

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)]
```

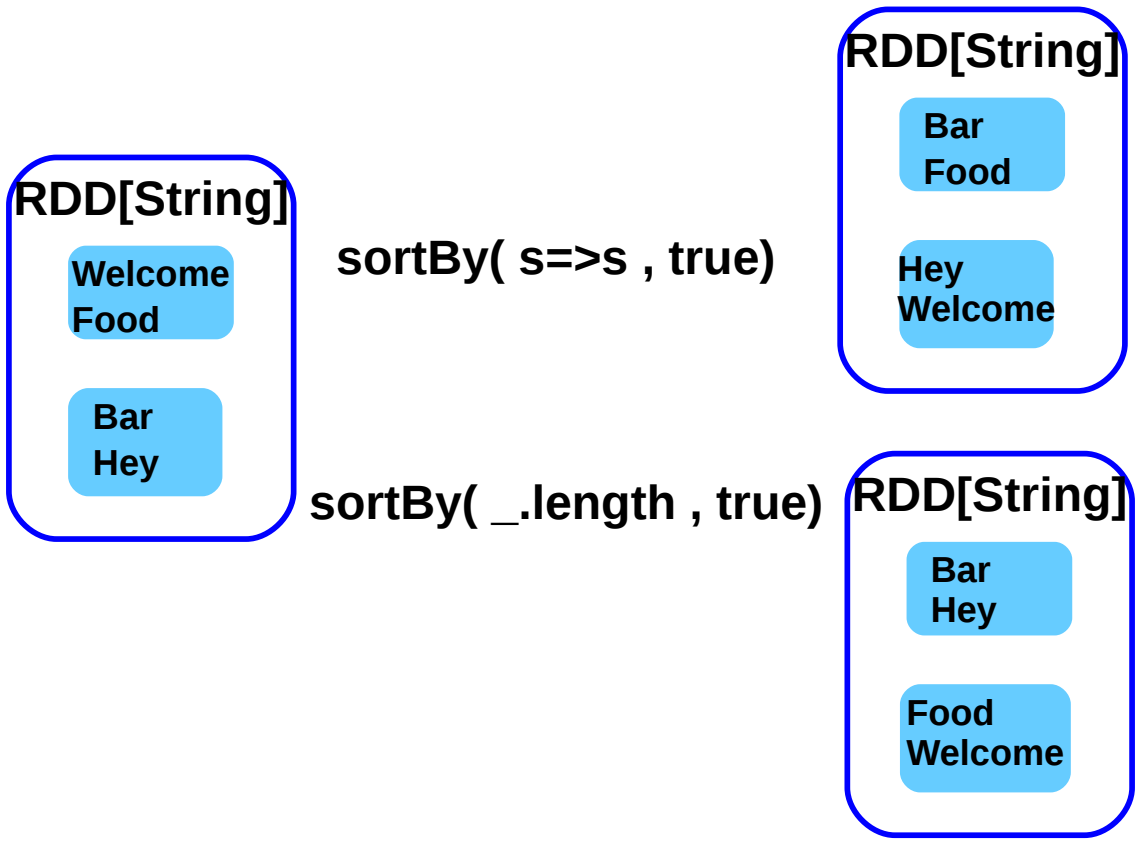


Transformation large sortBy

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]
```

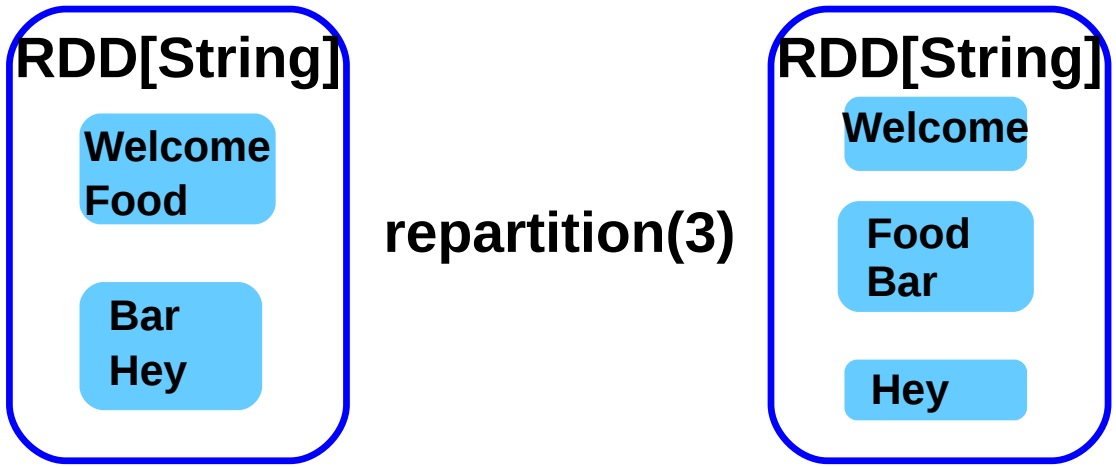


Transformation large `repartition`

Spécification

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

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



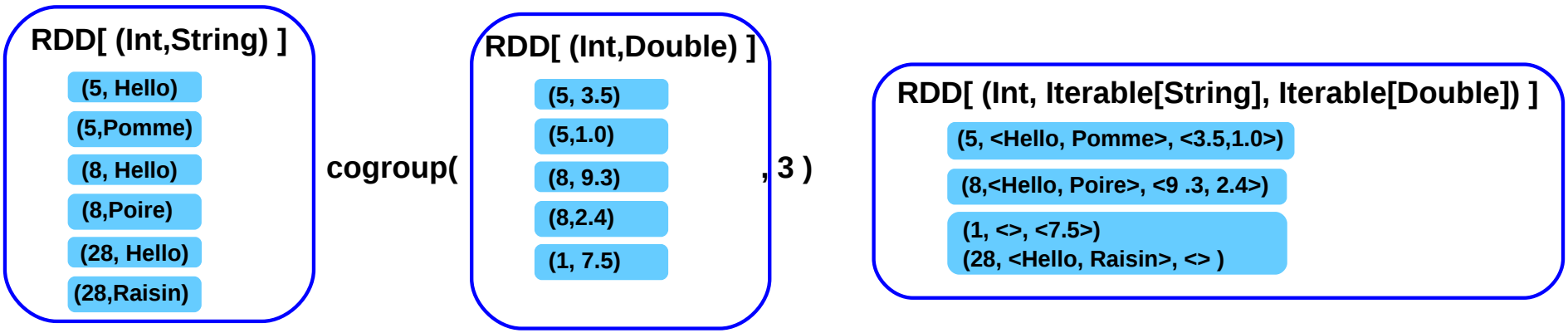
Transformation large cogroup

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
```



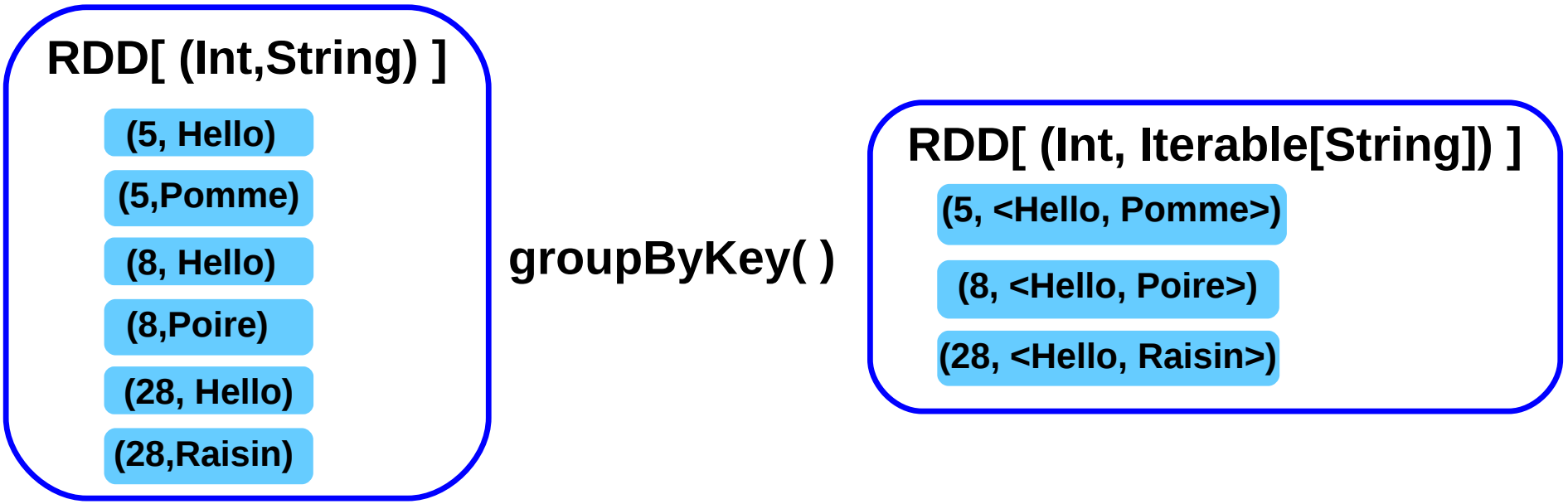
Transformation large groupByKey

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])]
```



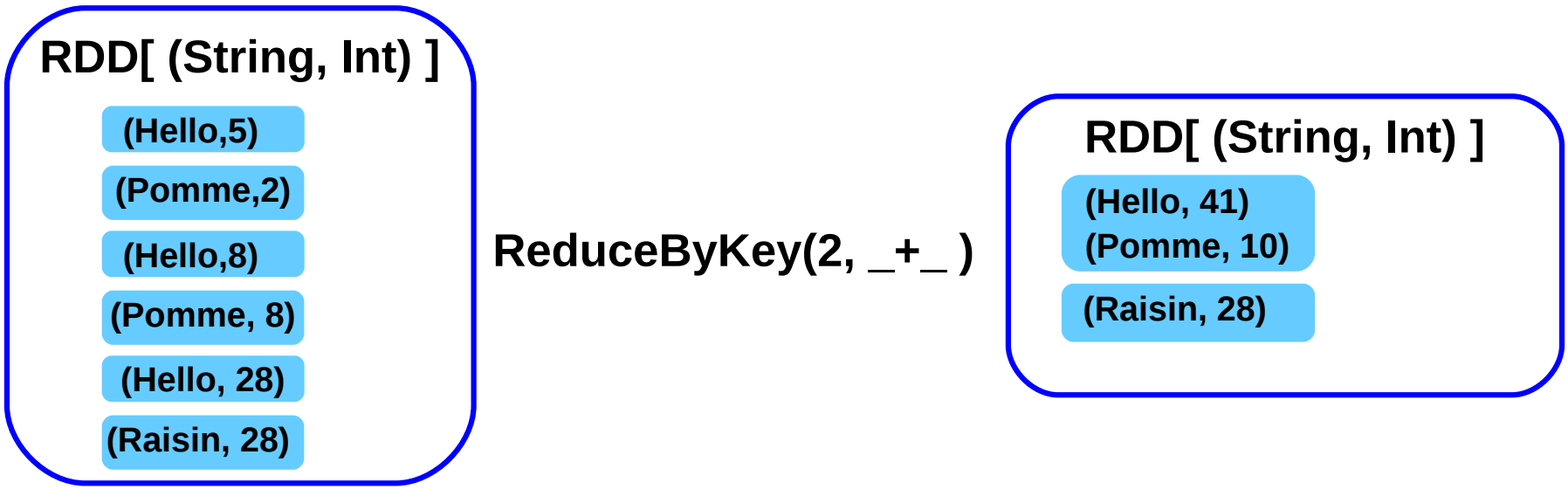
Transformation large `reduceByKey`

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)]
```



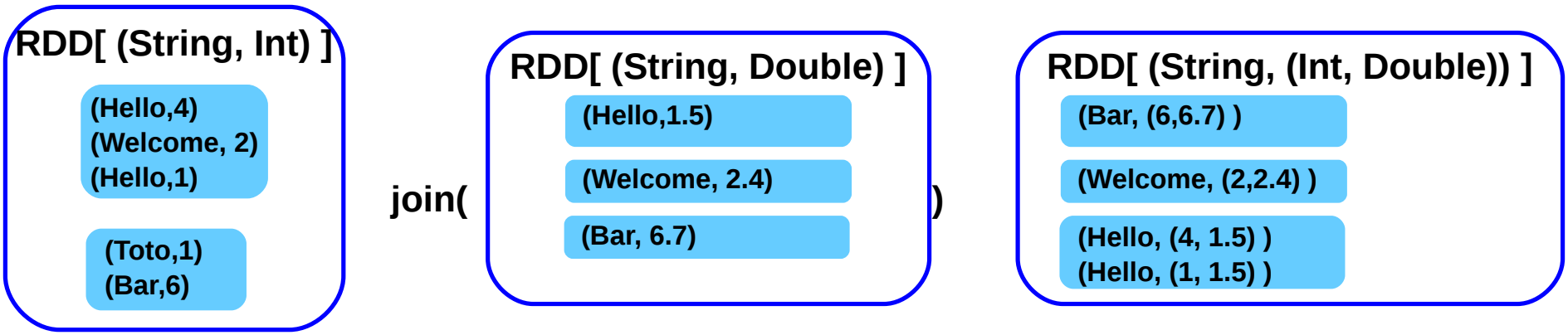
Transformation large `join`

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))]
```



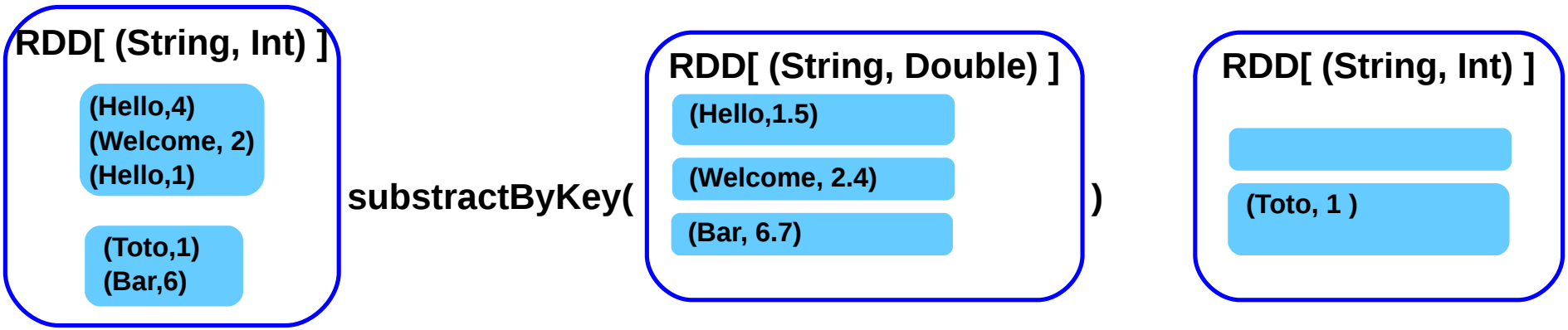
Transformation large `subtractByKey`

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

Spécification

Retourne un nouvel RDD où les clés du RDD appelant ne sont pas dans un autre RDD passé en paramètre

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



Les actions

Définition

- Marque la fin du flux de donnée :
  - en retournant une valeur résultat à l'application
  - et/ou en exportant les données sur un stockage stable
- déclenche la soumission d'un job Spark

  $\Rightarrow$exécution de toutes les transformations du flux

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

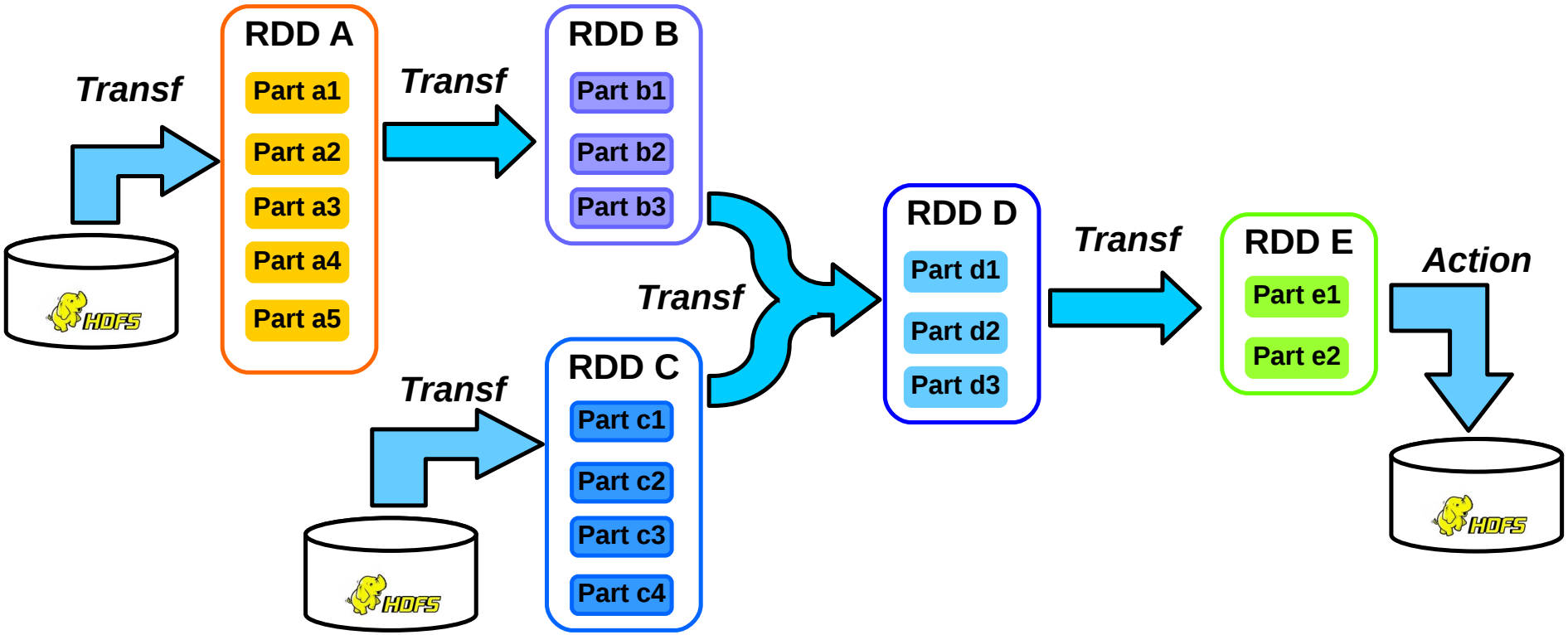
- `def saveAsObjectFile(path: String): Unit` :
  Sauvegarde en tant qu'objets sérialisés dans le fichier `path`.

- `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`.

Vision globale d'un flux de données



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

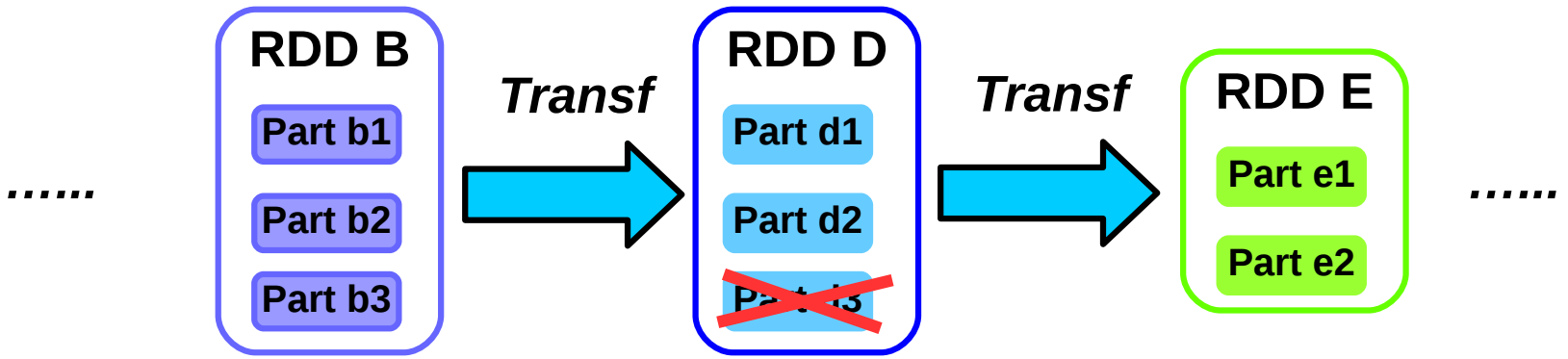
- `def cache: RDD.this.type`
  $\Rightarrow$ Persiste en mémoire vive
- `def persist(newLevel: StorageLevel): RDD.this.type`
  $\Rightarrow$ Persiste en spécifiant un niveau de stockage
- `def unpersist(blocking: Boolean = true): RDD.this.type`
  $\Rightarrow$ Annule la persistance en mode bloquant ou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

Les niveaux de stockage

- `MEMORY_ONLY` et `MEMORY_ONLY_SER` : stocke le RDD de manière désérialisée en mémoire vive
  - ⇒ 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` : stocke le RDD de manière désérialisée en mémoire vive et sérialise sur le disque local si la mémoire est insuffisante
  - ⇒ moins rapide
  - ⇒ pas besoin de recalculer les données si perte
- `DISK_ONLY` : stocke le RDD entièrement sur disque
  - ⇒ récupération la moins performante
  - ⇒ stockage de RDD volumineux possible

Possibilité de répliquer les partitions sur `N` nœuds avec `MEMORY_ONLY_<N>`, `MEMORY_AND_DISK_<N>`, `DISK_ONLY_<N>`, etc.

Tolérance aux pannes



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

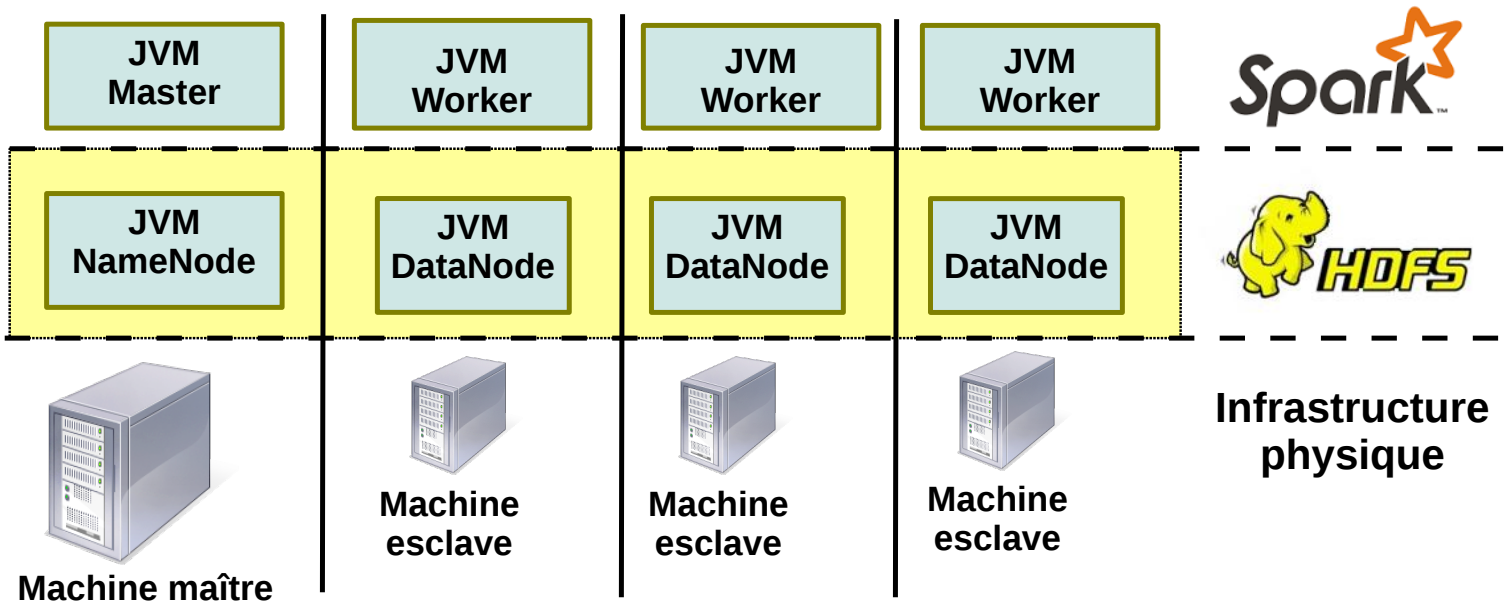
- Sans persistance : Retour à la case départ
- Avec persistance : reprendre à partir de l'ancêtre persistant le plus proche
- Avec persistance et réplication :faute transparente

Un programme Spark

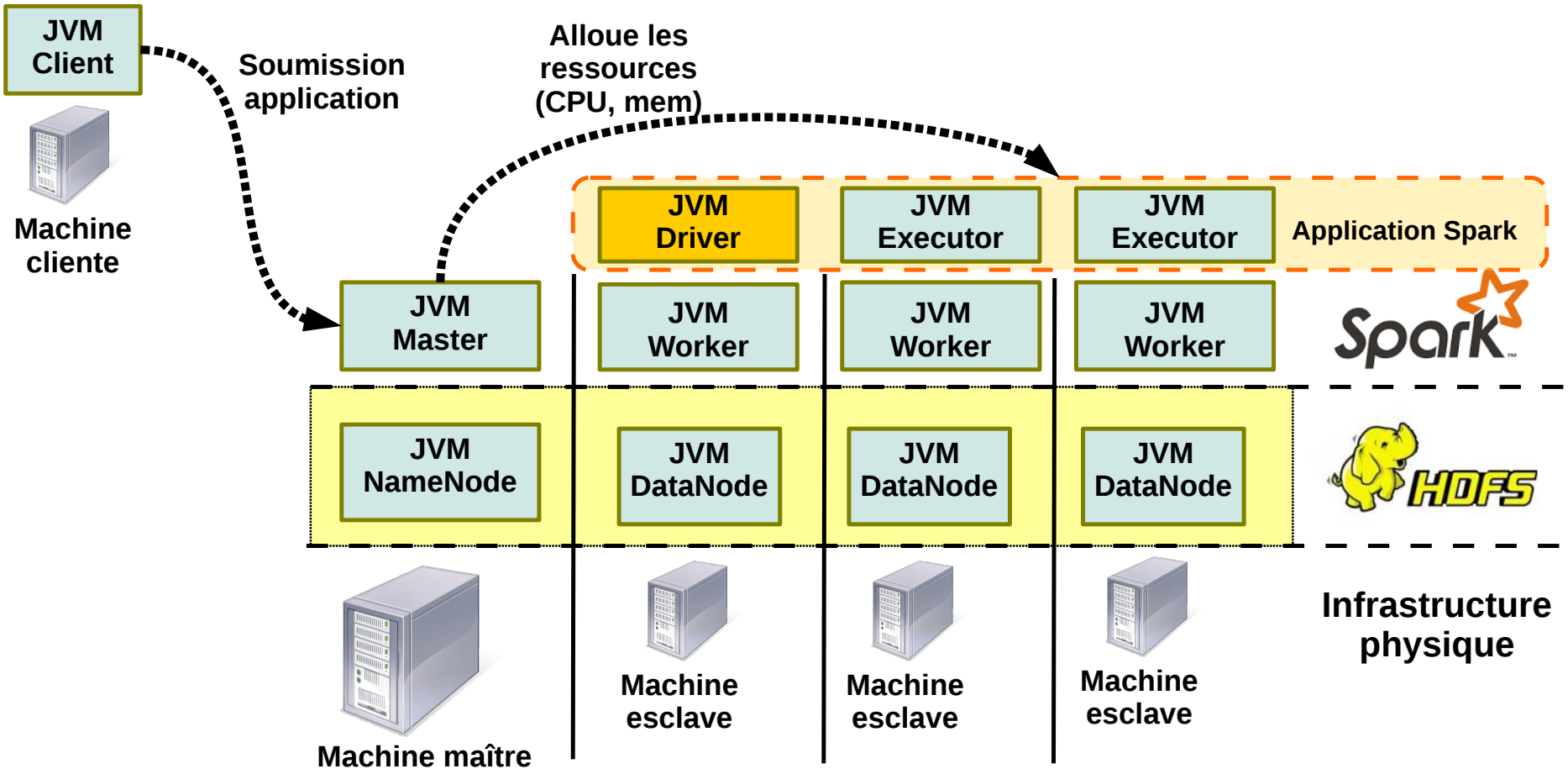
```
object Programme {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Mon Programme");
    val spark = new SparkContext(conf)
    val textFile = spark.textFile("hdfs://...")
    val rdd = textFile.flatMap(line => line.split(" "))
                      .map(x => (x, 1))
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

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

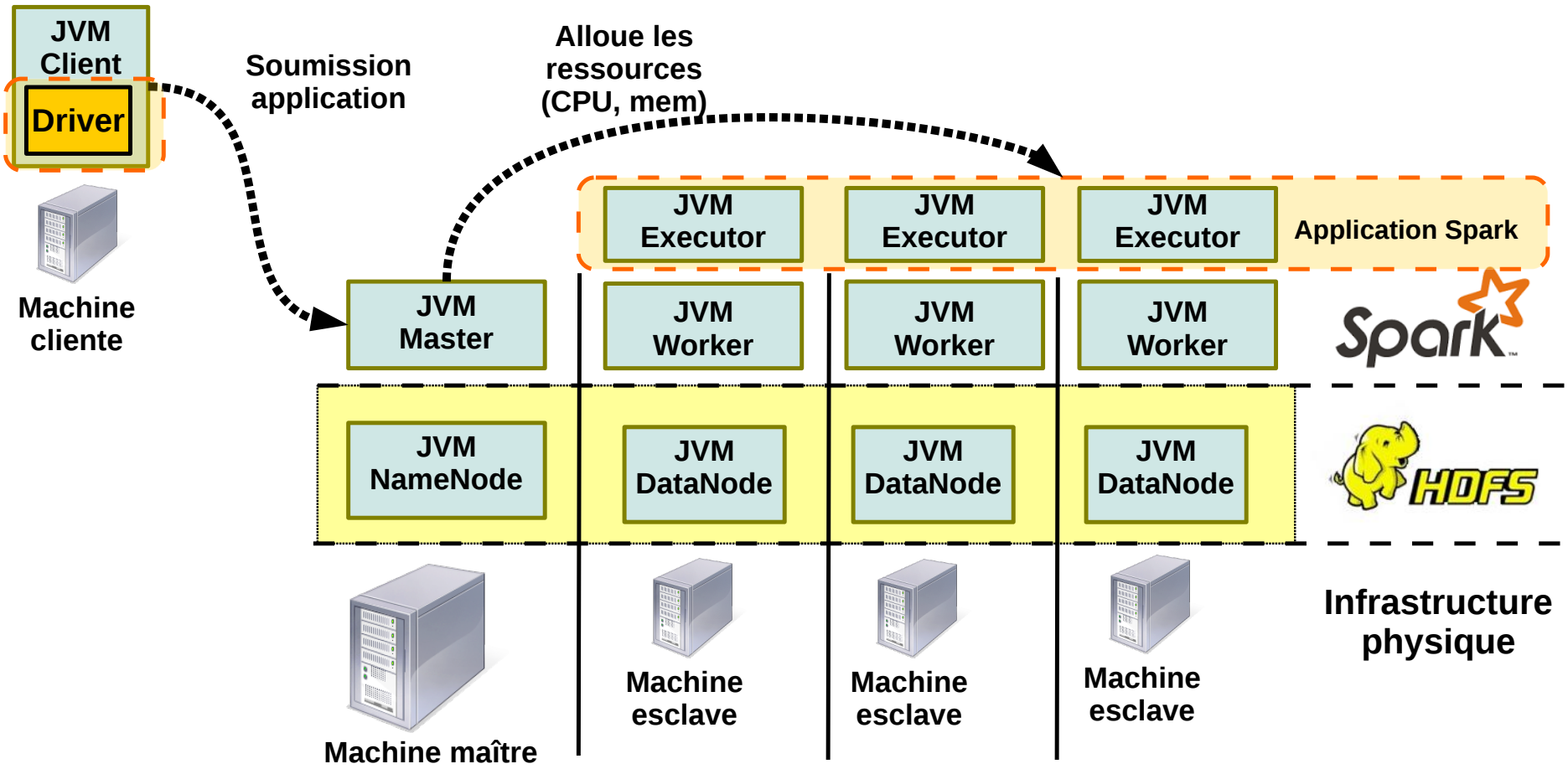
Architecture de Spark



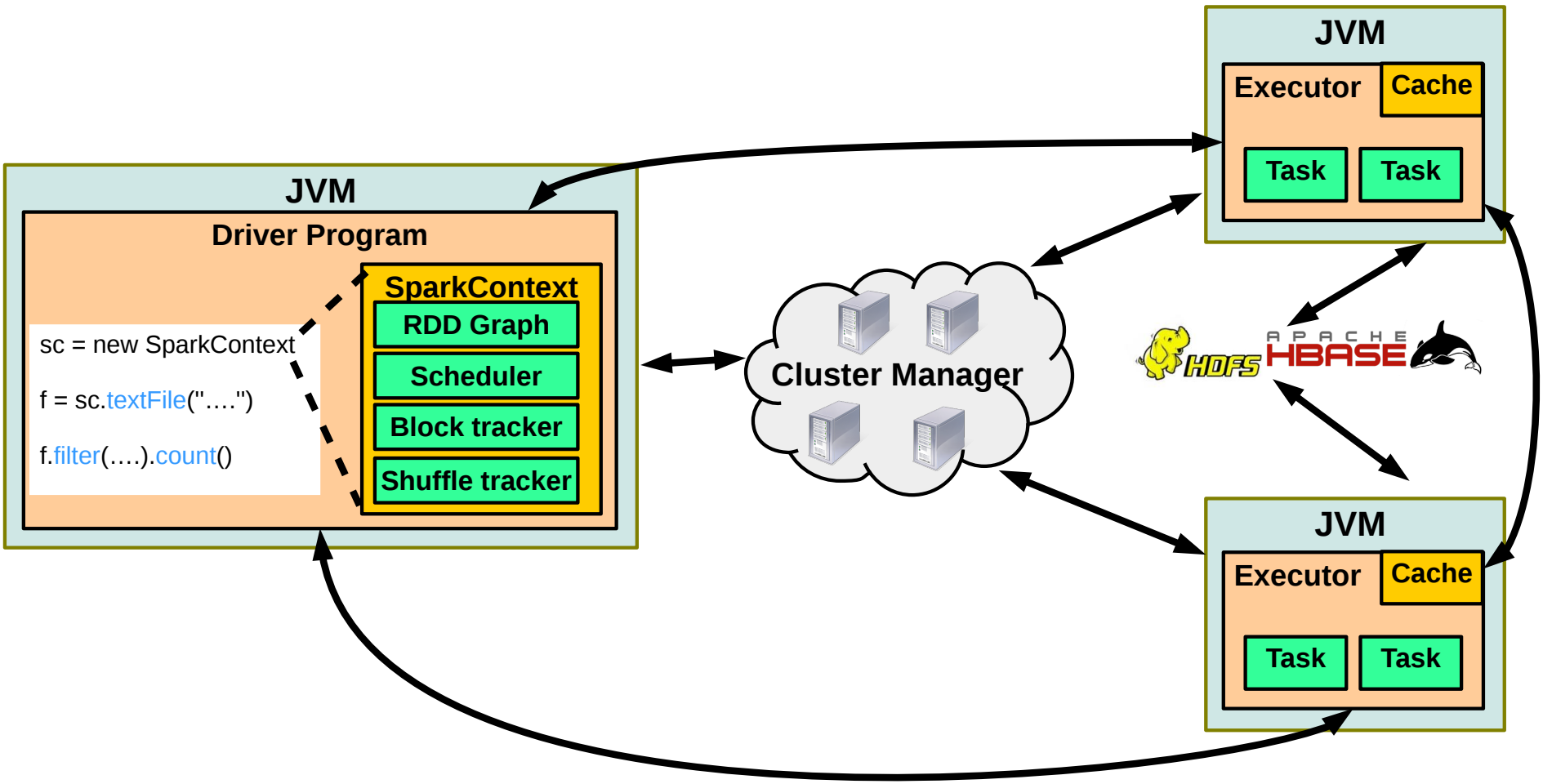
Soumission d'une application : mode cluster



Soumission d'une application : mode client



Application Spark



Application Spark

Définitions

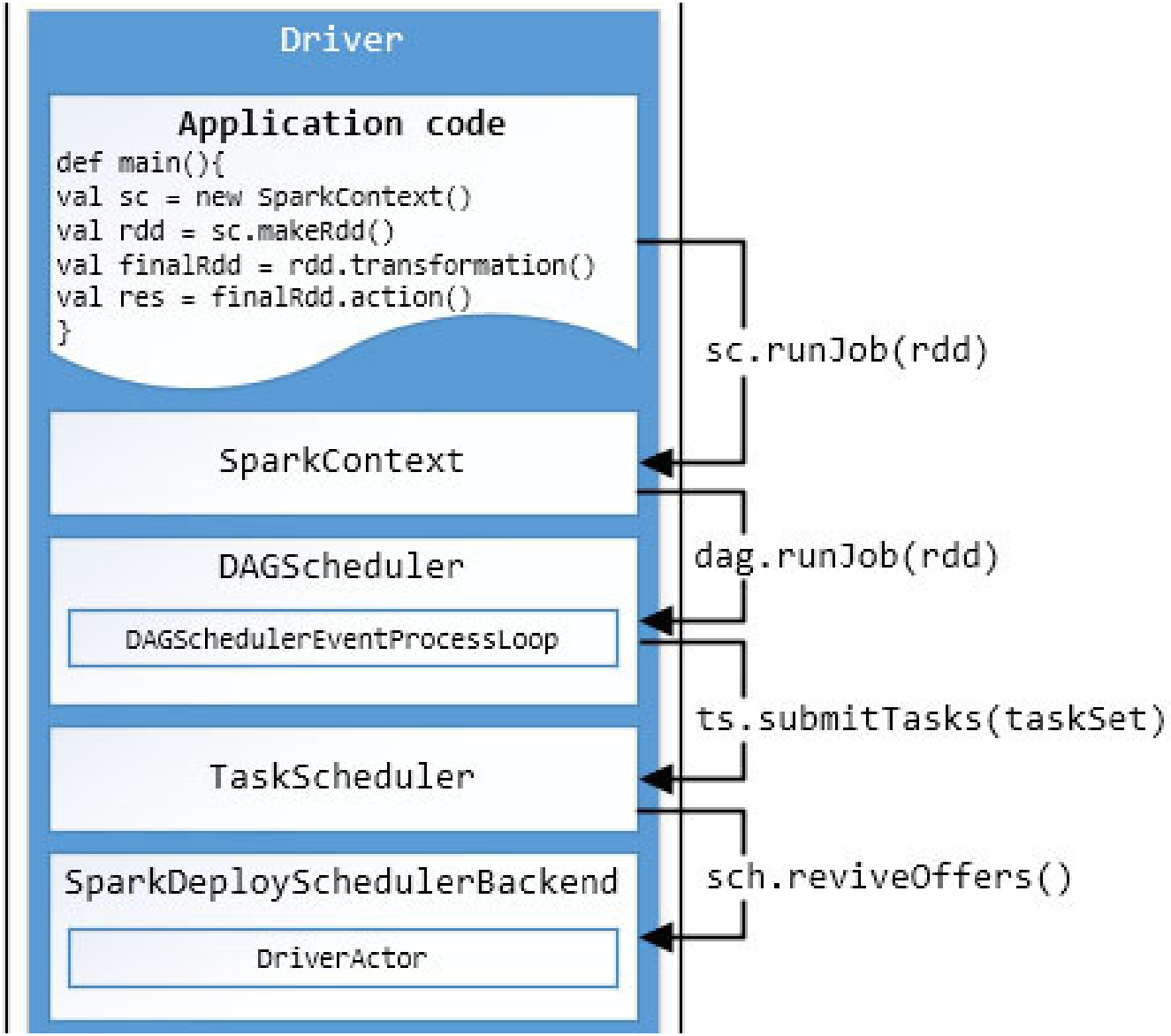
- **Driver :** Le processus exécutant la fonction main() de l'application
- **Cluster manager :** le service qui gère le cluster
- **Worker :** un nœud (physique) qui peut exécuter du code applicatif
- **Executor :** un processus exécutant les tâches, garde les données en mémoire ou sur disque. Affecté à une seule et unique application

Le driver

Orchestre les différents exécuteurs de l'application. Il est composé principalement de deux sous-tâches :

- **d'un DAG Scheduler :**
  - Connaissance des fonctions à exécuter sur chaque partition
  - Construits des groupes de tâches à exécuter (code + localisation)
  - À l'écoute des résultats de l'application
  - Soumet les tâches au Task Scheduler quand elles n'ont plus de dépendance
  - Soumet à nouveau les groupes de tâches défaillantes
- **d'un Task Scheduler**
  - Ordonnance les tâches sur les executors
  - relance les tâches défaillantes
  - Informer le DAG Scheduler

Soumission d'un job



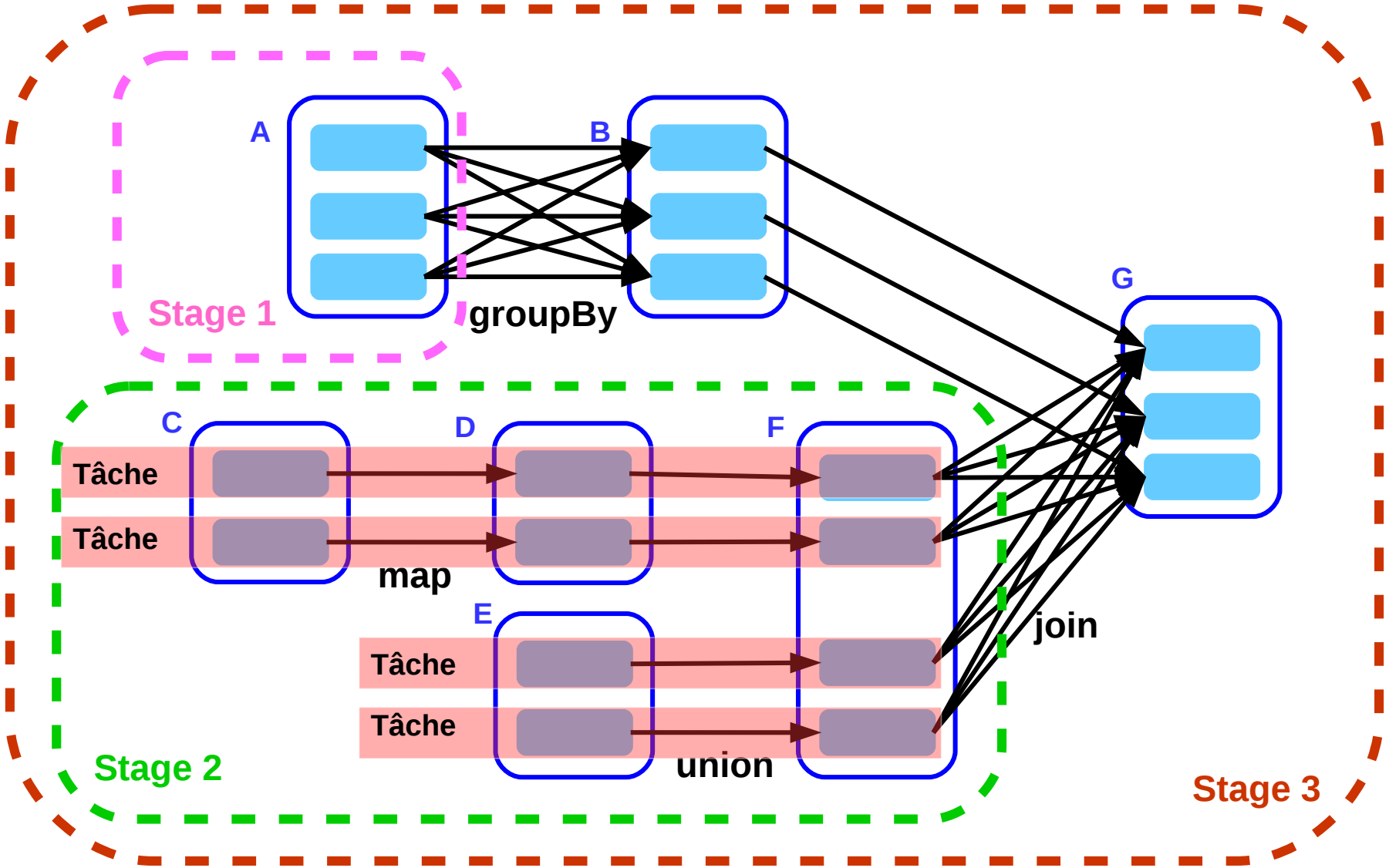
Les *stages*

Définition

- Ensemble de tâches indépendantes
- Toutes les tâches font le même traitement
- Toutes les tâches dans le stage ont le même type de dépendance
- 2 types de stage :
  - **Shuffle Map :** délimité par des opérations de type shuffle
    $\Rightarrow$ résultats des tâches = entrées d'un autre stage
  - **Result :** tâches qui calculent une action finale

Organisation des *stages*

Organisation sous la forme d'un DAG



Modes de déploiement de Spark

Déploiement local

- Exécution locale sur un seul processus que l'on peut multi-threader
- Tests et debugages de programmes

Déploiement distribué

Spark est compatible avec 3 gestionnaires de cluster :

- **mode Standalone** : le gestionnaire de Spark
- **Mesos** : gestionnaire distribué de conteneur
- **Yarn** : un container pour le driver (appmaster) et un container par executors

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 variables partagées

Définition

On parle de variable partagée lorsqu'une variable est accédée par différents processus ayant un espace d'adressage propre

Conséquence

Chaque processus n'a qu'une copie de la variable counter :

$\Rightarrow$ une mise à jour n'est pas répercutée sur les autres réplicas

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

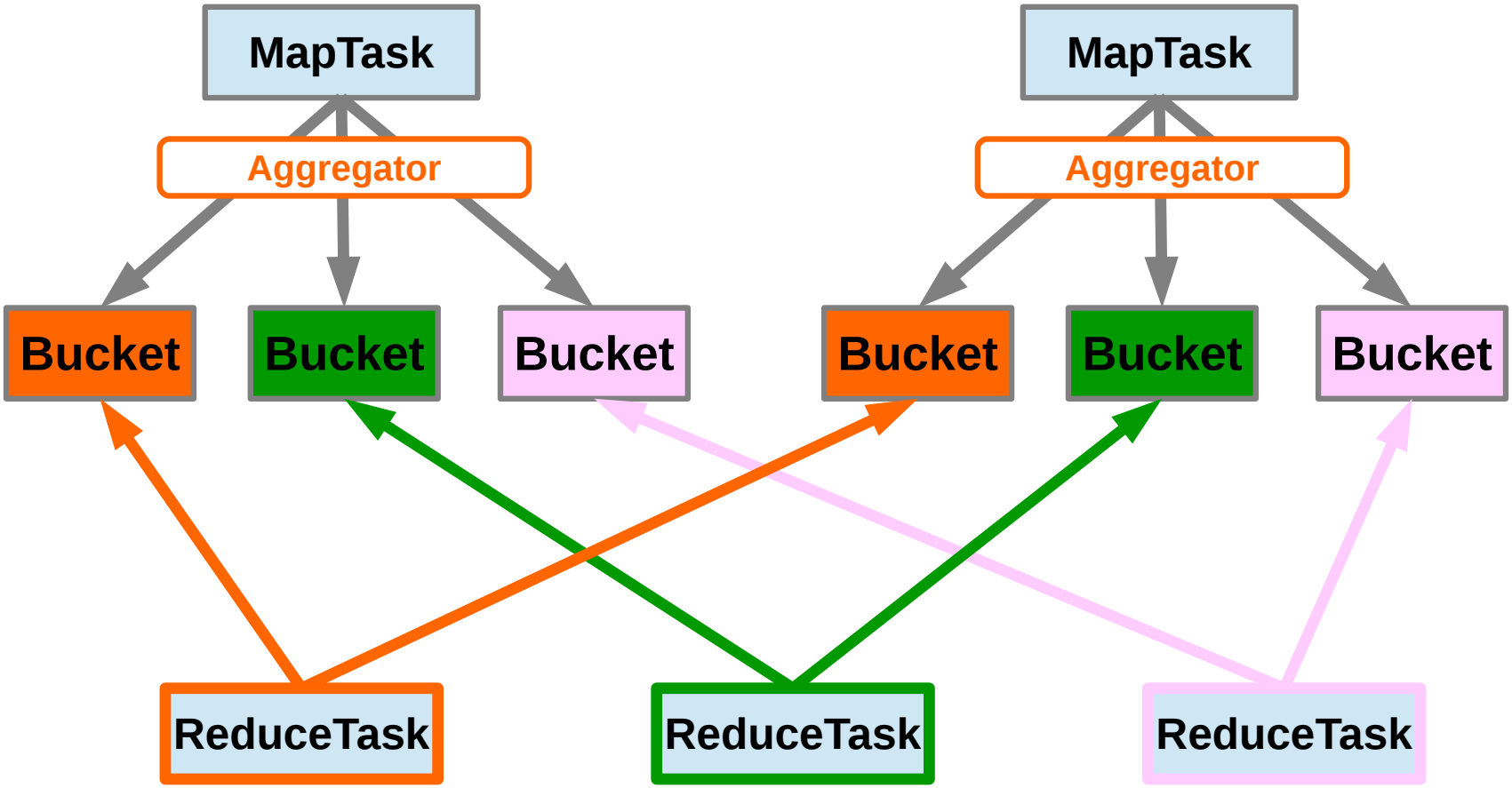
- variable qui peut uniquement s'incrémenter
- créée à partir d'un valeur initiale et s'utilise avec l'opérateur +=
- seul le driver peut y accéder en lecture

```
val accum = sc.accumulator(0, "My Accumulator")
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
accum.value // permet d'accéder à la valeur de l'accumulateur
```

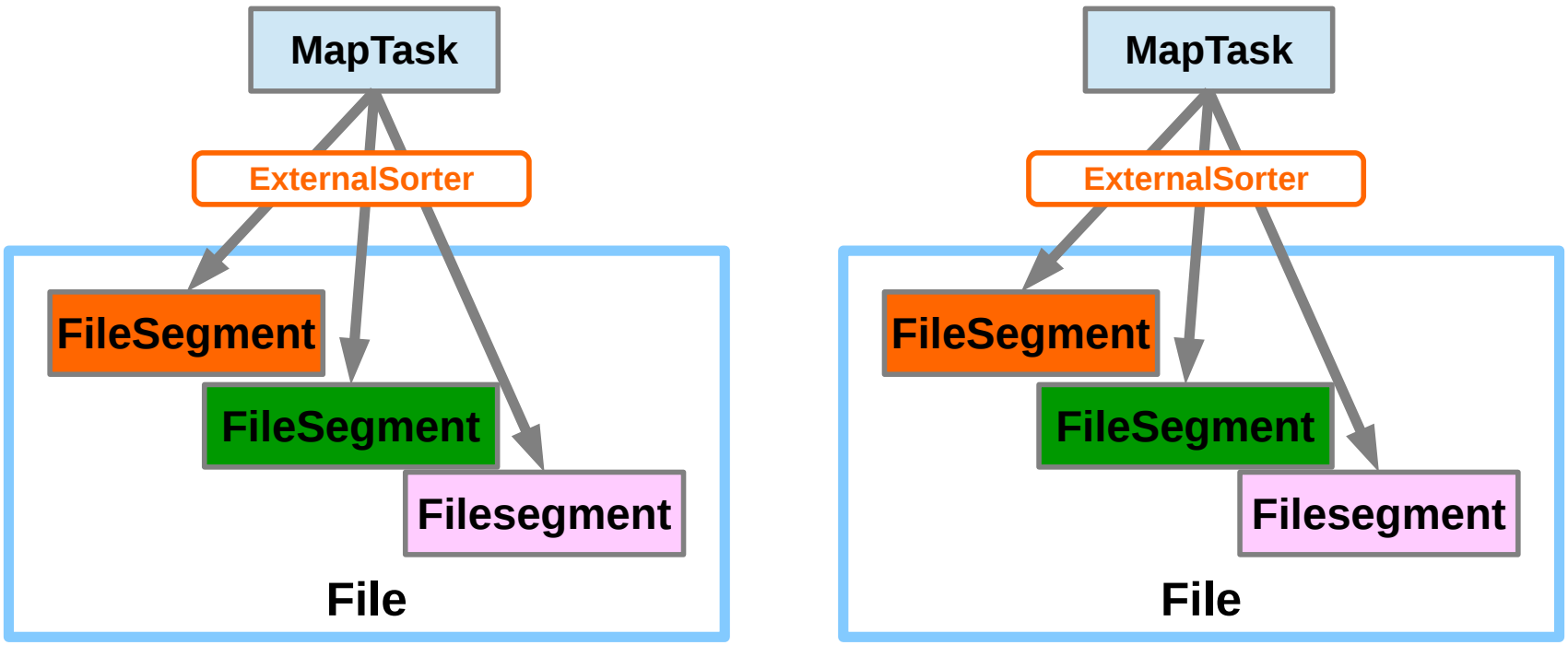
Implémentation des opérations de Shuffle

Même principe que pour Hadoop Map Reduce :

- Stockage des résultats intermédiaires sur le système de fichier local
- Partitionnement des données intermédiaires
- les données sont téléchargés par les tâches de la phase suivante



Implémentation des opérations de Shuffle



Regroupement trié par tâche :

- équivalent au mécanisme de shuffle de Hadoop
- un fichier trié par tâches
- une politique de tri modulaire
- NbFichiers = *nbMaps*

Shuffle : Hadoop vs. Spark

Hadoop Map Reduce	Spark
Shuffle coté map : • écritures des sorties dans un tampon (ou sur le disque si tampon à 80%) • pour un nœud **un seul gros fichier** partitionné Shuffle coté reduce : • (télé)chargement en mémoire des sorties de map (déchargement sur le disque si tampon à 70%) • fusion des fichiers de déchargement	Shuffle coté map : • un fichier par tâche • déchargement sur le disque délégué à l'OS sous-jacent (Buffer-Cache) Shuffle coté reduce : • les données sont directement chargées en mémoire et déchargées si la mémoire allouée est insuffisante

Spark privilégie la mémoire vive aux disques

Interface Web

machine driver, port 4040