Données massives et apprentissage profond

Cours 1 – Introduction à Apache Spark

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Polytech Paris-Saclay, 2024

Organisation du cours

- MapReduce et Spark.
- Programmation Spark.
- SQL et NoSQL.
- Pratique de MongoDB.
- Technologies Hadoop.
- Mise à l'échelle.

Matériel de cours

Disponible en ligne
https://tinyurl.com/p7jb5wra

- Diapositives des cours.
- Travaux pratiques et exercices.

Évaluation

- **Travaux pratiques**. Les travaux pratiques 1 et 2 seront notés.
  - Travail pratique 1. Programmation Spark.
  - Travail pratique 2. MongoDB.
  - **Soumission** : Code source + rapport écrit.

- **Examen écrit.** 1 heure.
  - Programmation Spark.
  - Modélisation de données dans MongoDB.
  - Language de requêtes MongoDB

Contact

Email : gianluca.quercini@centralesupelec.fr

Email : stephane.vialle@centralesupelec.fr

Introduction à Apache Spark

Apache Spark

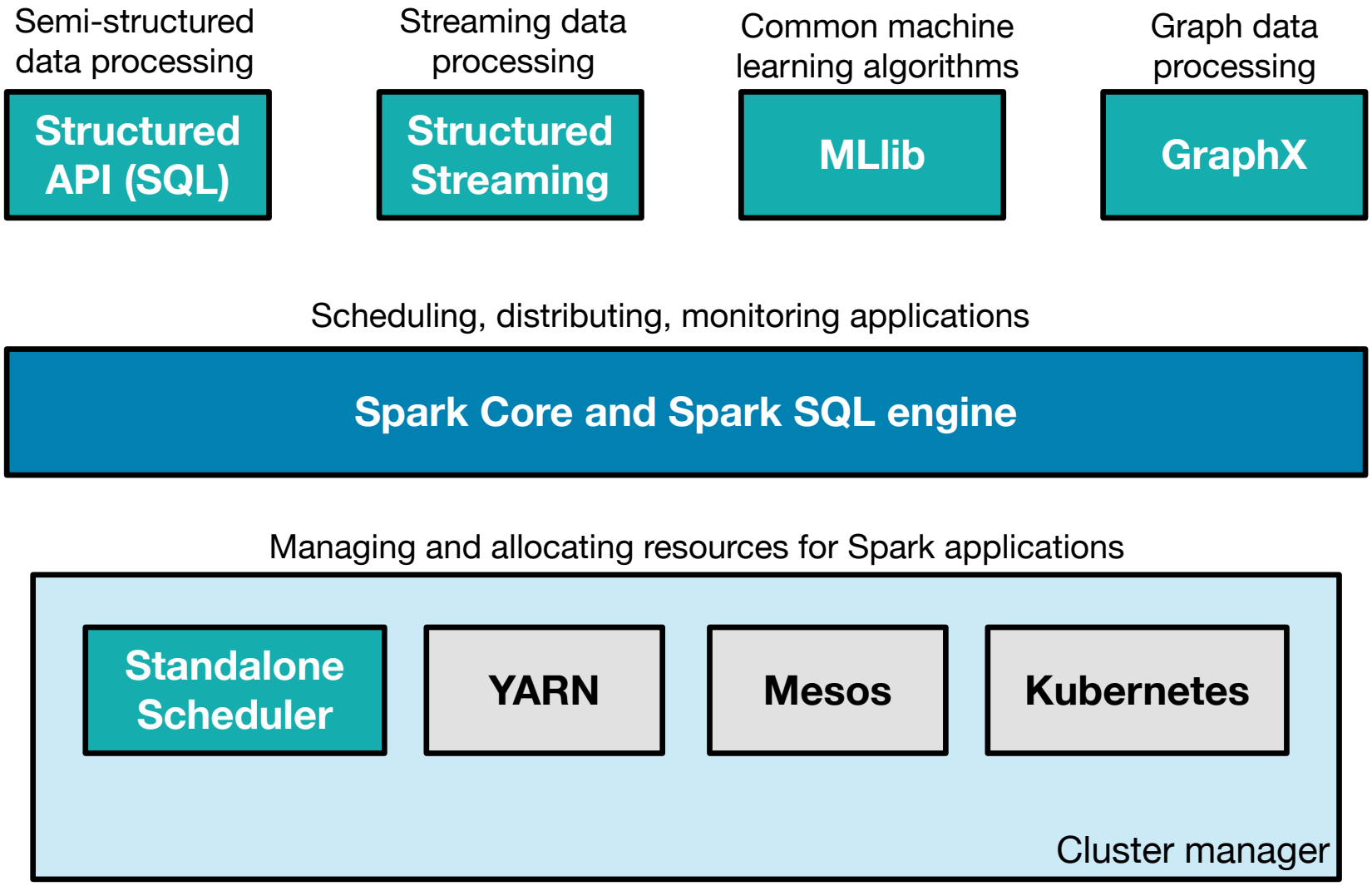
Definition (Apache Spark)

Apache Spark est un *framework de calcul distribué* conçu pour être *rapide* et à *usage général*. ▸ Source

Principales caractéristiques

- **Rapidité**. Exécution des calculs en **mémoire** (Hadoop s'appuie sur les disques).
- **Usage général**. Différents types de charges de travail dans le même système.
  - Traitement par lots, algorithmes itératifs.
  - Requêtes interactives, analyses de flux de données.
- **Accessibilité**. Python, Scala, Java, SQL et R ; riches bibliothèques intégrées.
- **Intégration**. Avec d'autres outils Big Data, comme **Hadoop**.

Composants de Spark



▸ Source de l'image

Environnement intégré de Spark : avantages

- **Courbe d'apprentissage peu élevée.** Le même **modèle de programmation** est utilisé dans tous les composants.
- **Propagation de l'optimisation.** Les composants de plus haut niveau bénéficient automatiquement des améliorations apportées aux composants de plus bas niveau.
- **Minimisation des coûts.** Pas besoin de composants logiciels supplémentaires.
- **Modèles de traitement hétérogènes** dans la même application.
  - Lire un flux de données.
  - Appliquer des algorithmes d'apprentissage automatique.
  - Utiliser SQL pour analyser les résultats.

Utilisation de Spark

Mode interactif

Utilisation d'une interface en ligne de commande (CLI).

- Shell Python et Scala.
- Shell SparkSQL.
- Shell SparkR.

Applications de traitement des données

Implémentation d'une **application** en utilisant les **API Spark**.

- Scala (langage natif de Spark).
- Python.
- Java.

Qui utilise Spark ?

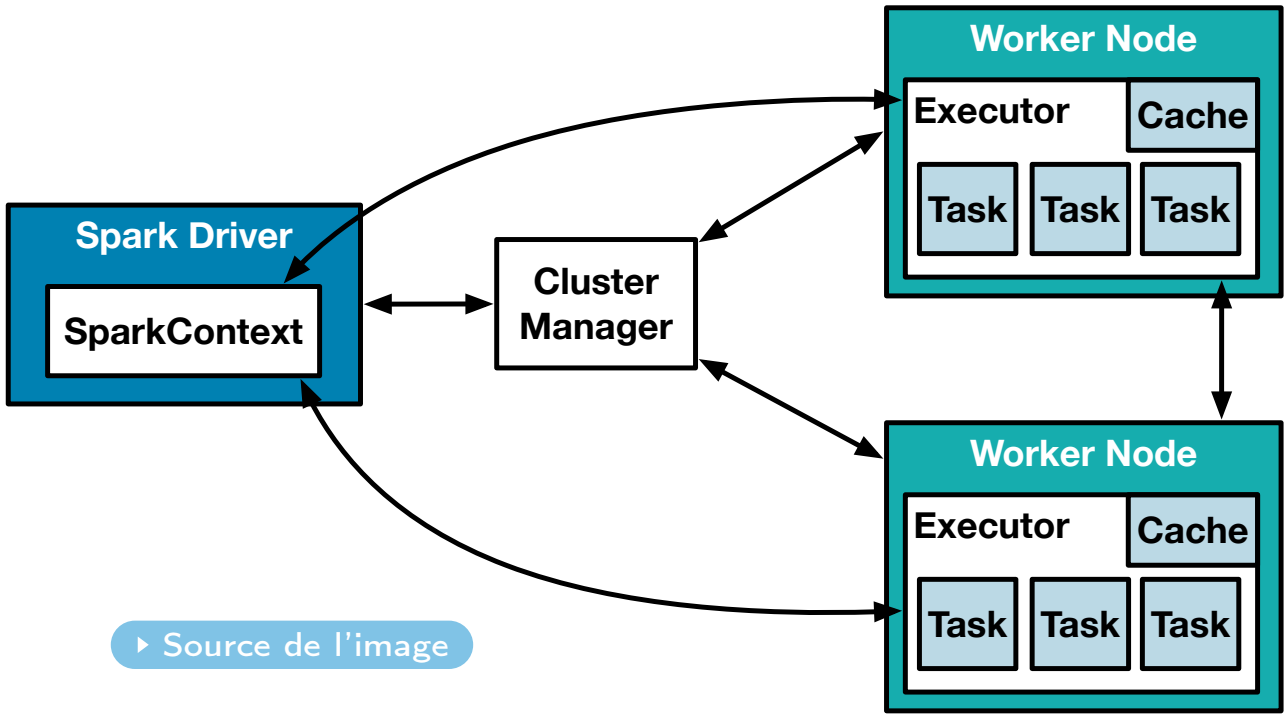
Plusieurs acteurs importants utilisent Spark :

- **Amazon**.
- **eBay**. Agrégation et analyse des journaux de transactions.
- **Groupon**.
- **Stanford DAWN**. Projet de recherche visant à démocratiser l'IA.
- **TripAdvisor**.
- **Yahoo!**

☞ Liste complète disponible ▸ ici

Application Spark

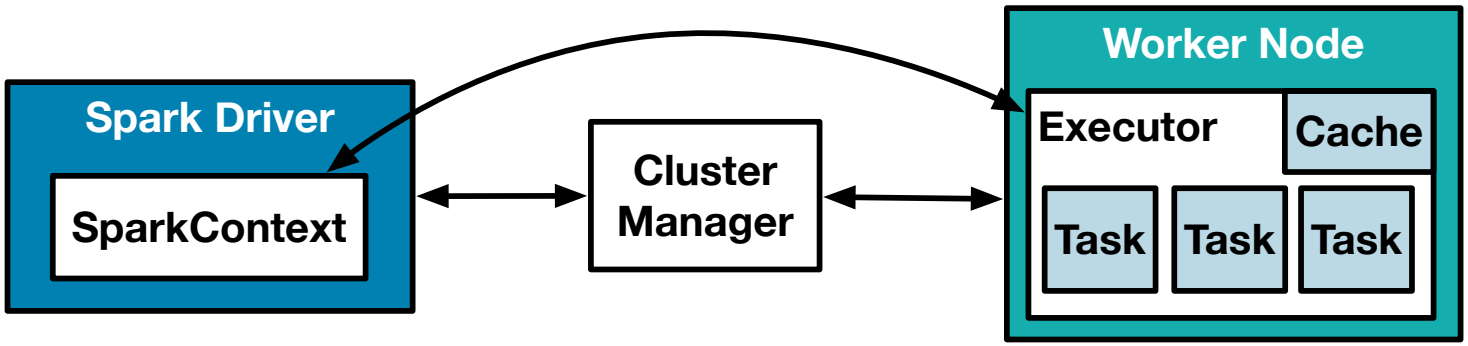
- Une application Spark est gérée par un programme appelé **Driver**.
- Le Driver communique avec le gestionnaire de cluster et les exécuteurs à traver un *SparkContext*.



▸ Source de l'image

Exécution d'une application Spark

- Le **Driver** est lancé et crée l'objet **SparkContext**.
- Le **SparkContext** obtient des **exécuteurs** auprès du **gestionnaire de cluster**.
- Le Driver envoie le **code de l'application** aux exécuteurs.
- Le Driver attribue à chaque exécuteur un ensemble de **tâches**.
- Une **tâche** est un calcul sur un **sous-ensemble de données**.



Exécution d'une application Spark

- Les applications sont **isolées** les unes des autres.
  - Chaque application a son propre *SparkContext*.
  - Un exécuteur exécute uniquement les tâches d'une seule application.
  - Un Driver planifie uniquement les tâches pour une seule application.
  - Les données ne peuvent pas être partagées entre différentes applications.
- Spark est **indépendant** du gestionnaire de cluster sous-jacent.
- Le Driver communique avec les exécuteurs via le réseau.
- Le Driver doit être dans le même réseau local que les exécuteurs.

☞ Deux applications Spark différentes peuvent tout de même partager des données via un système de stockage externe (par exemple, une base de données ou des fichiers HDFS).

Programmer en Spark

Deux options existent pour écrire une application Spark :

- Programmation **bas niveau**, en utilisant des opérations sur une structure de données de bas niveau appelée **resilient distributed datasets (RDD)**.
- Programmation **haut niveau**, en utilisant des bibliothèques de haut niveau, telles que Structured API et Structured Streaming.

☞ Dans cette présentation, nous nous concentrerons sur la programmation de bas niveau pour mieux comprendre le fonctionnement interne de Spark.

Programmation bas niveau avec Spark

- Un programme Spark utilise un objet appelé **SparkContext**.

Initialisation du SparkContext

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster(<URL du cluster>)\
      .setAppName(<nom_de_l'application>)
sc = SparkContext(conf = conf)
```

- Un programme Spark est une séquence d'opérations invoquées sur le **SparkContext (sc)**.

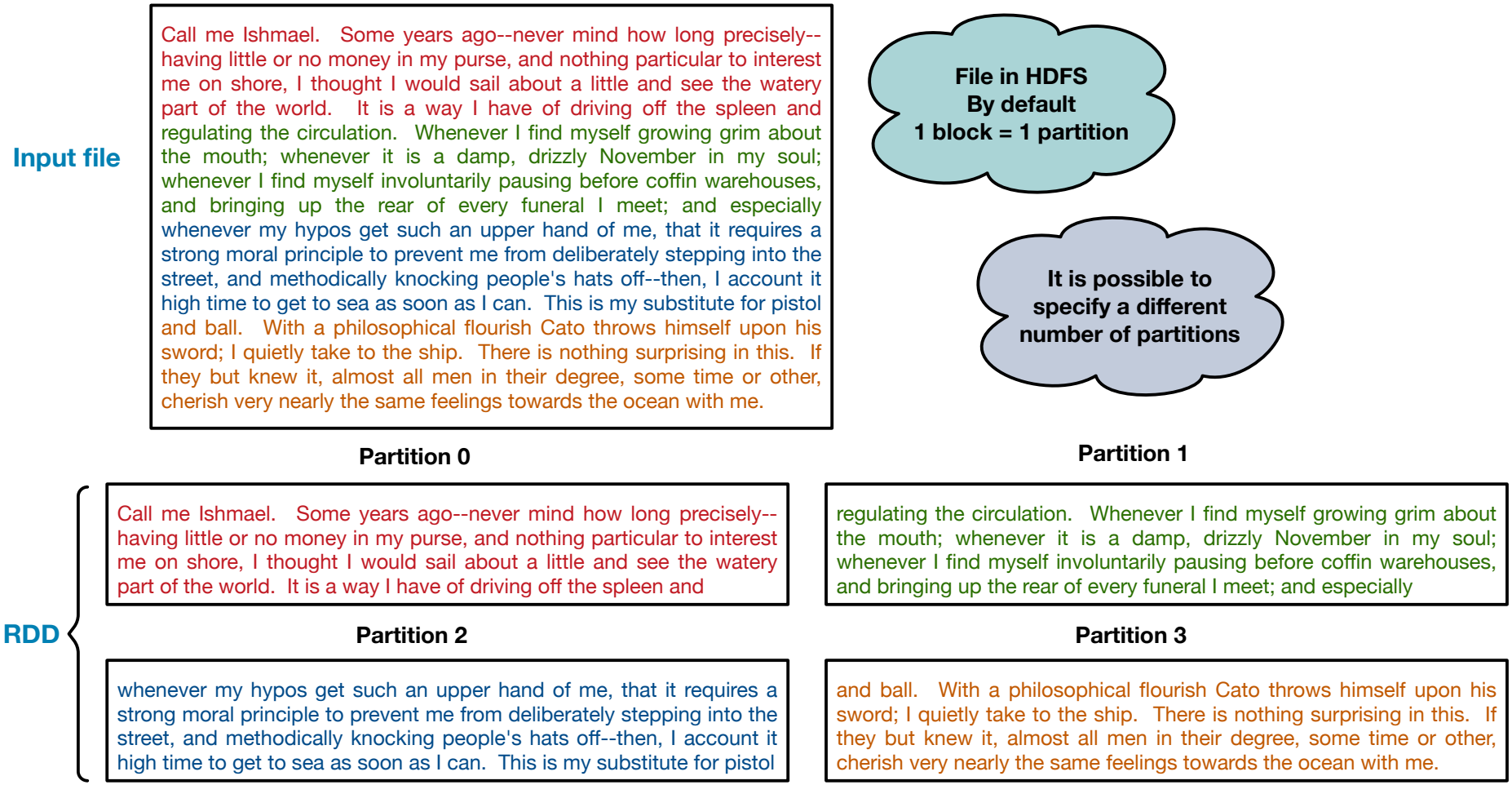
- Ces opérations manipulent des RDD.

Resilient Distributed Dataset (RDD)

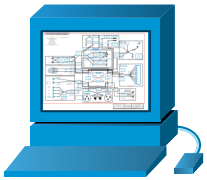
Un **RDD**, est une collection d'objets **immuable** et **distribuée**. ▸ Source

- Les données d'un RDD sont réparties sur plusieurs **partitions**.
- Une partition réside sur un serveur du cluster.
- Deux partitions peuvent résider sur le même serveur.

Resilient Distributed Dataset (RDD)

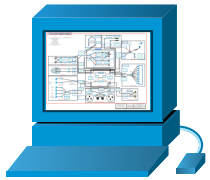


Resilient Distributed Dataset (RDD)

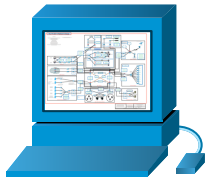


Call me Ishmael. Some years ago--never mind how long precisely--having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and

whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off--then, I account it high time to get to sea as soon as I can. This is my substitute for pistol



regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially



and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

Création d'un RDD

1. À partir d'une **collection en mémoire** (par exemple, une liste ou un ensemble).

```
sc.parallelize([1, 5, 3, 2, 6, 7])
```

> ☞ Cette méthode est utilisée pour le **débogage** et pour des **POCs** sur de **petits jeux de données**.

2. À partir d'une **source de données sur disque** (par exemple, un fichier ou une base de données).

```
sc.textFile("hdfs://sar01:9000/data/sample_text.txt")
```

> ☞ Cette méthode est utilisée **en production** pour traiter de **gros jeux de données**.

À propos du nombre de partitions

RDDs créés avec `parallelize`

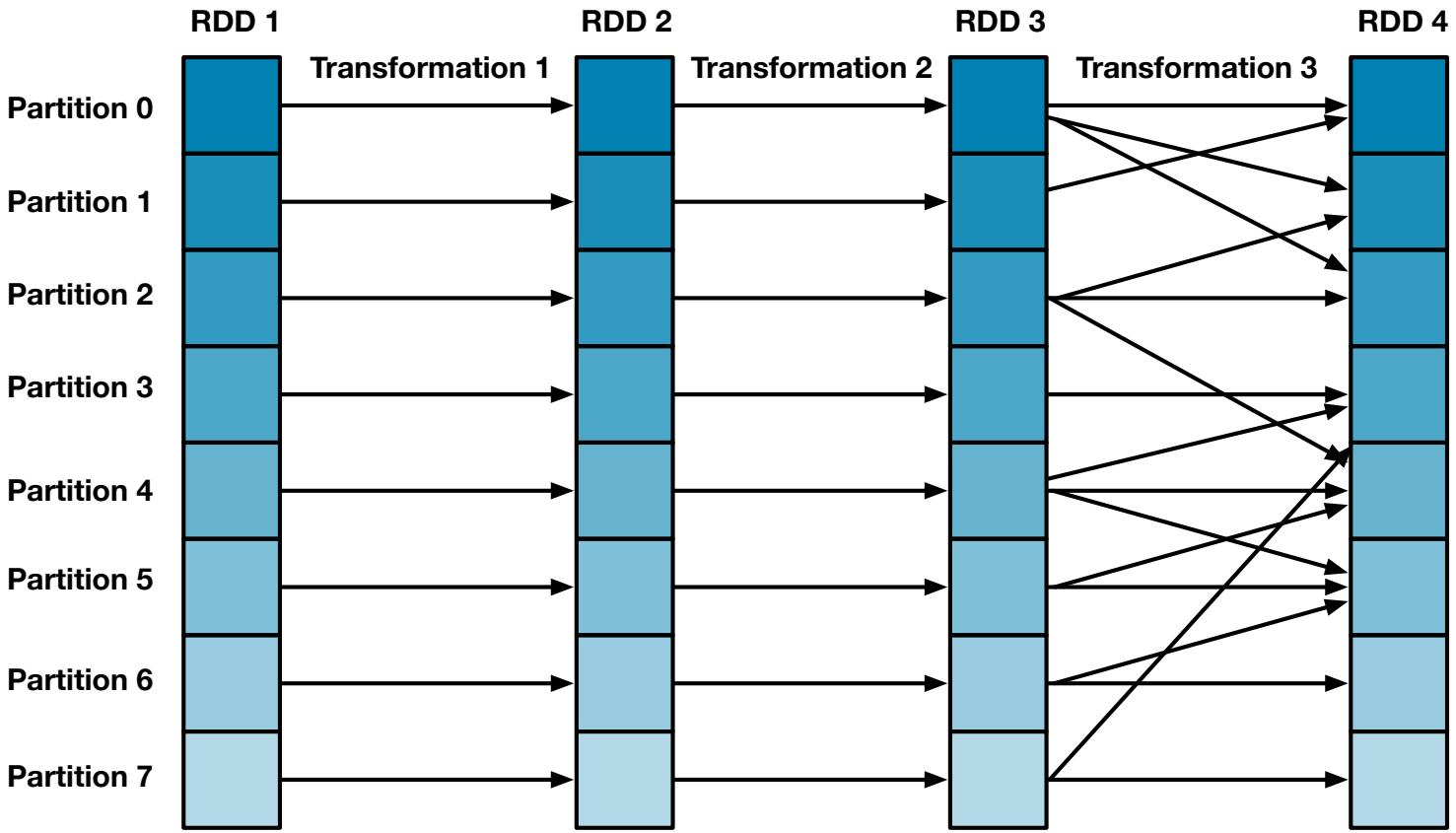
- **En local** : nombre de cœurs du serveur local.
- **Sur un cluster** : **nombre total de cœurs** de tous les serveurs, ou 2, selon le nombre le plus grand.

RDDs créés à partir de fichiers stockés dans HDFS

- Nombre de blocs HDFS du fichier, ou 2, selon le nombre le plus grand.

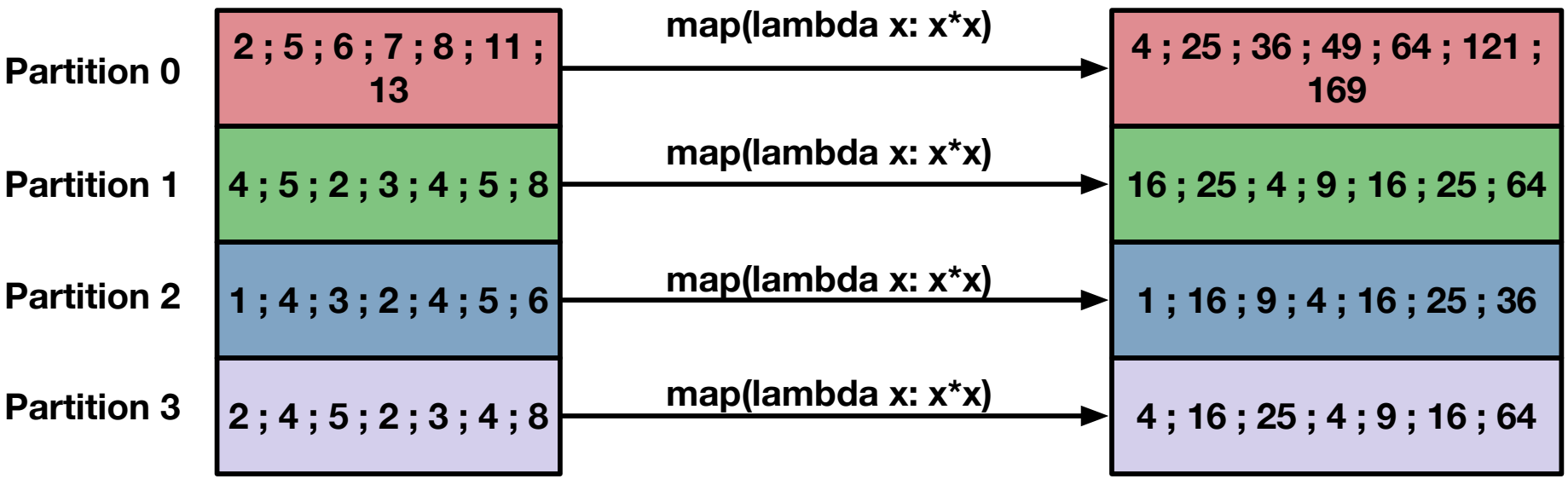
Transformations

Une **transformation** est une opération qui prend en argument un ou plusieurs RDD et renvoie un **nouveau RDD**. Une transformation est appliquée **en parallèle** sur chaque partition du RDD.



Transformations : `map`

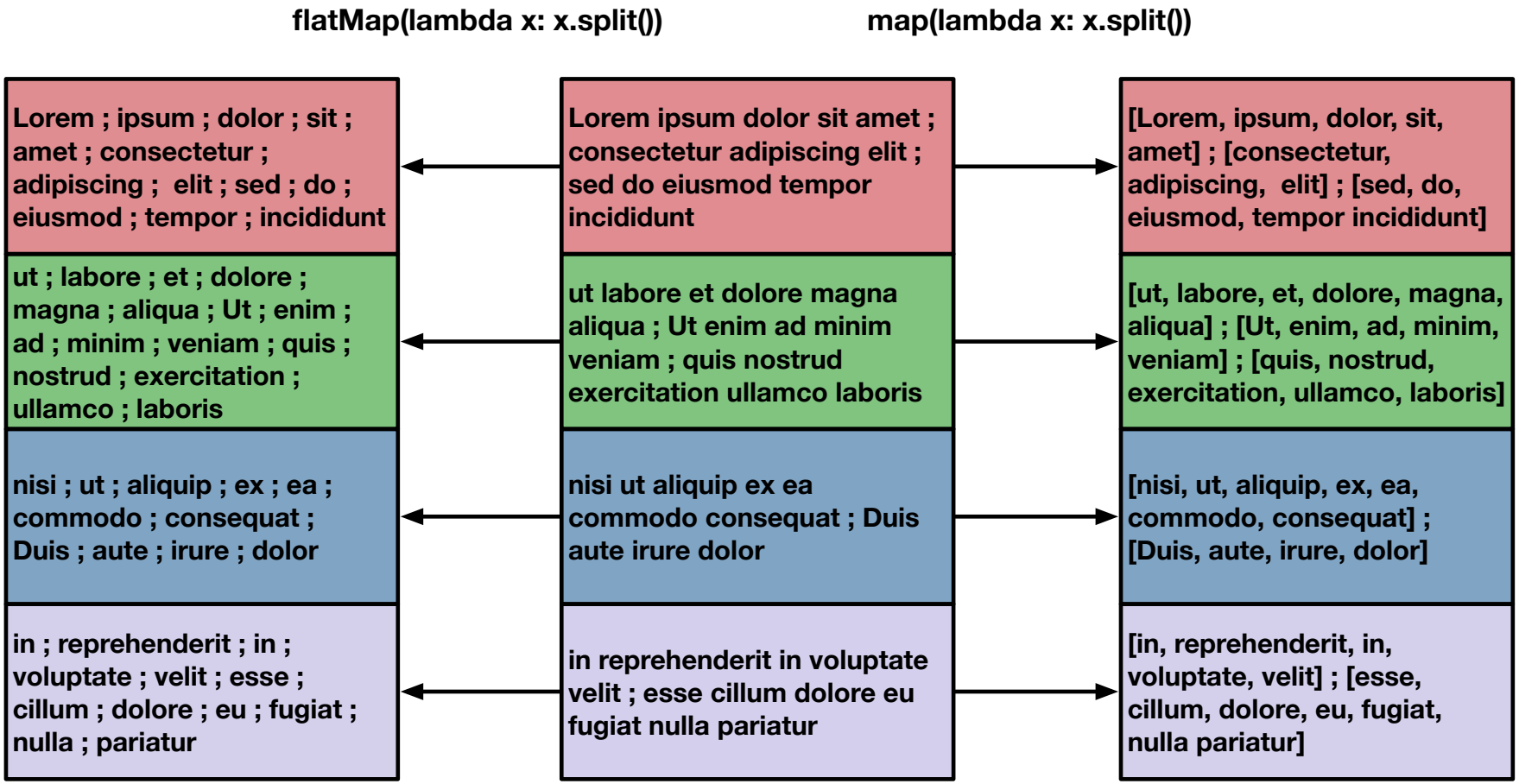
`map()` prend en arguments une **fonction** f et un RDD $< x_i \mid 0 \leq i \leq n >$; renvoie un **nouveau RDD** $< f(x_i) \mid 0 \leq i \leq n >$.



☞ La partition i du RDD en argument est sur le même serveur que la partition i du RDD de sortie.

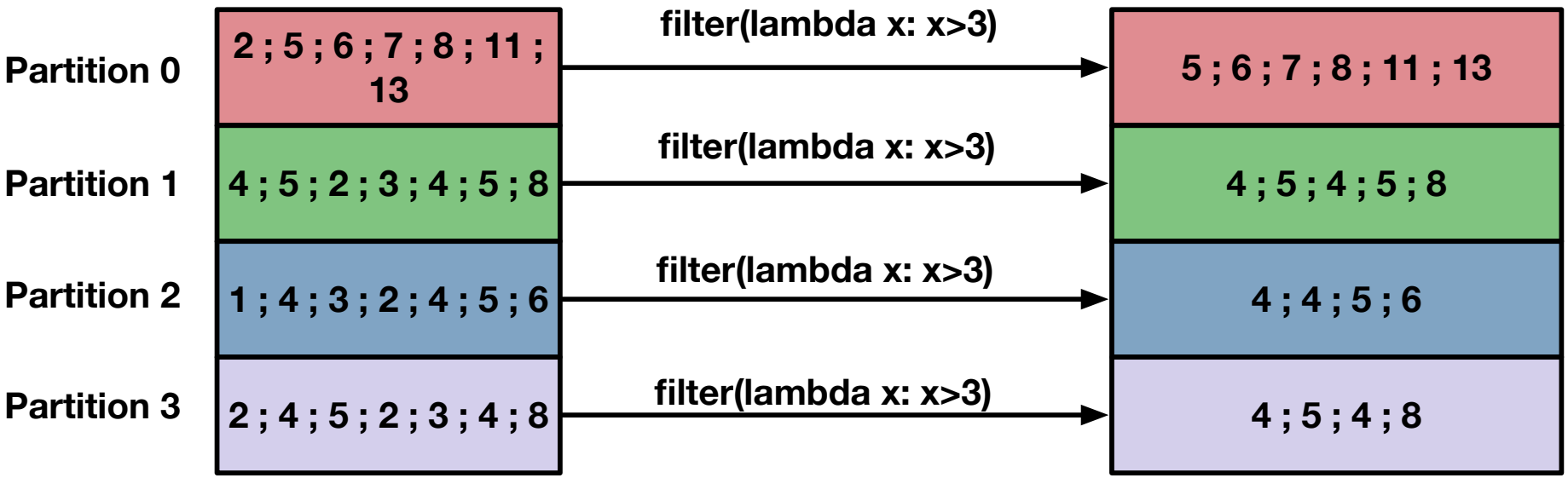
Transformations : `flatMap`

`flatMap` est utilisé à la place de `map` lorsque la fonction f renvoie une liste et que nous avons besoin que ces listes soient explosées.



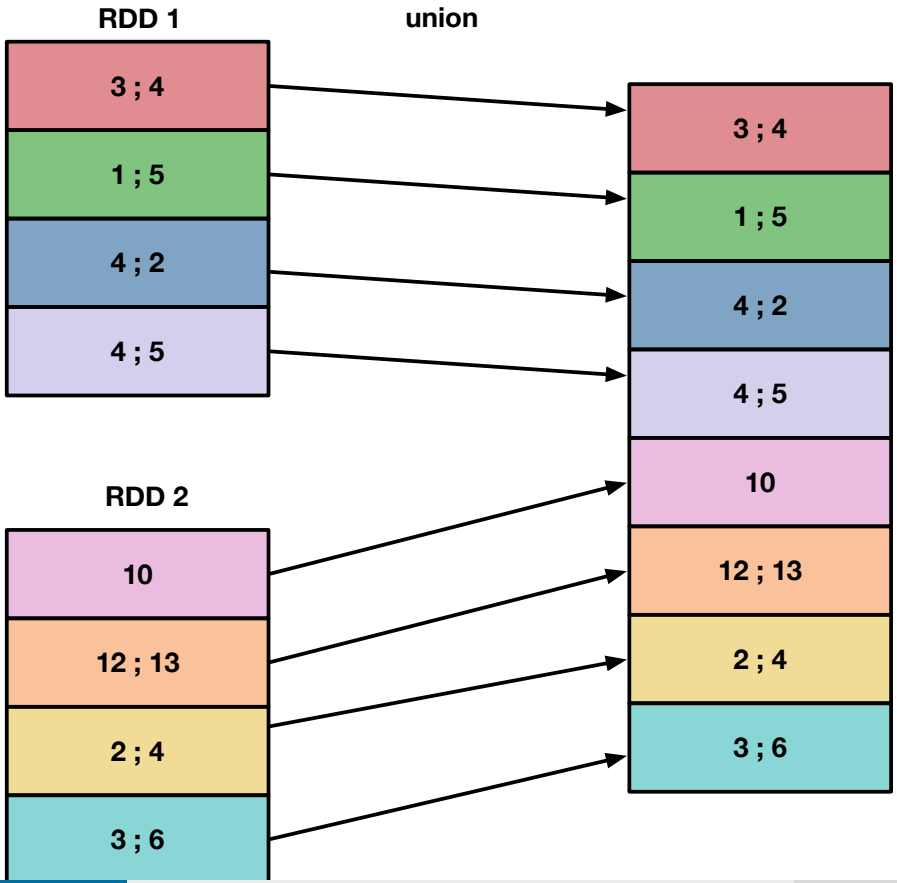
Transformations : `filter`

`filter()` prend en arguments un **prédicat** p et un RDD $< x_i \mid 0 \leq i \leq n >$; renvoie un **nouveau RDD** $< x_i \mid 0 \leq i \leq n,\ p(x_i)\ \textit{est vrai} >$



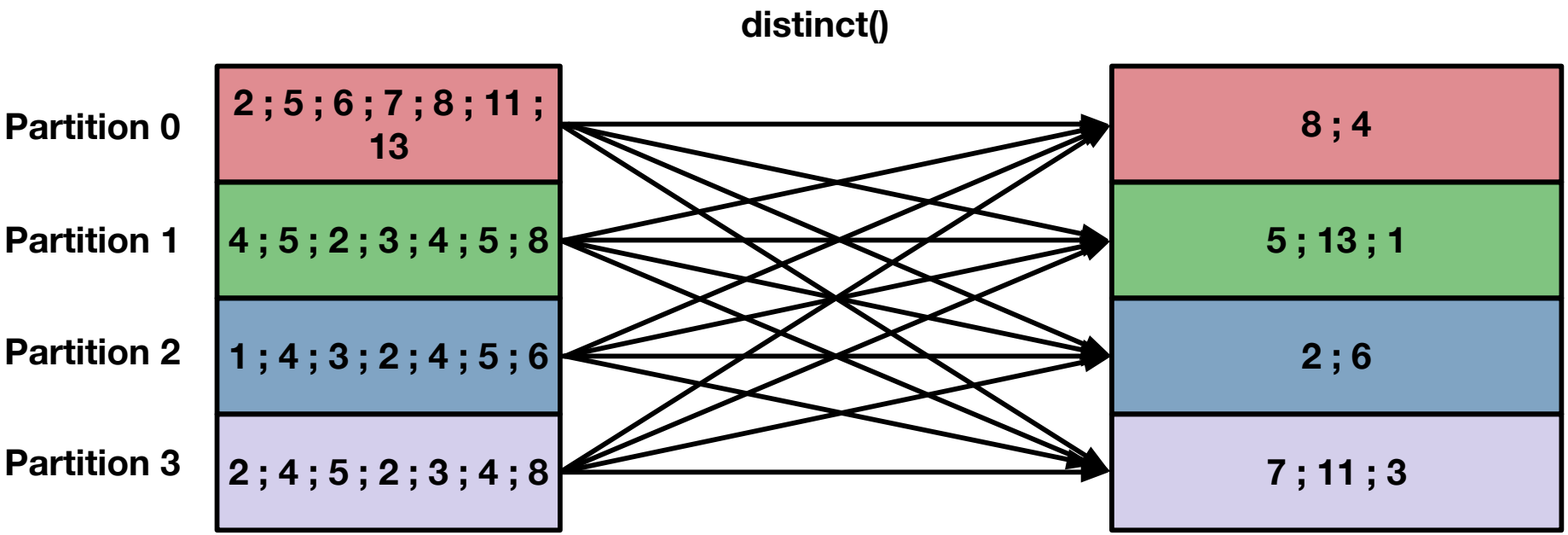
Transformations: `union`

`union()` prend en arguments deux RDDs et renvoie un **nouveau RDD** contenant les éléments du premier et du deuxième RDD **avec répétitions**.



Transformations: `distinct`

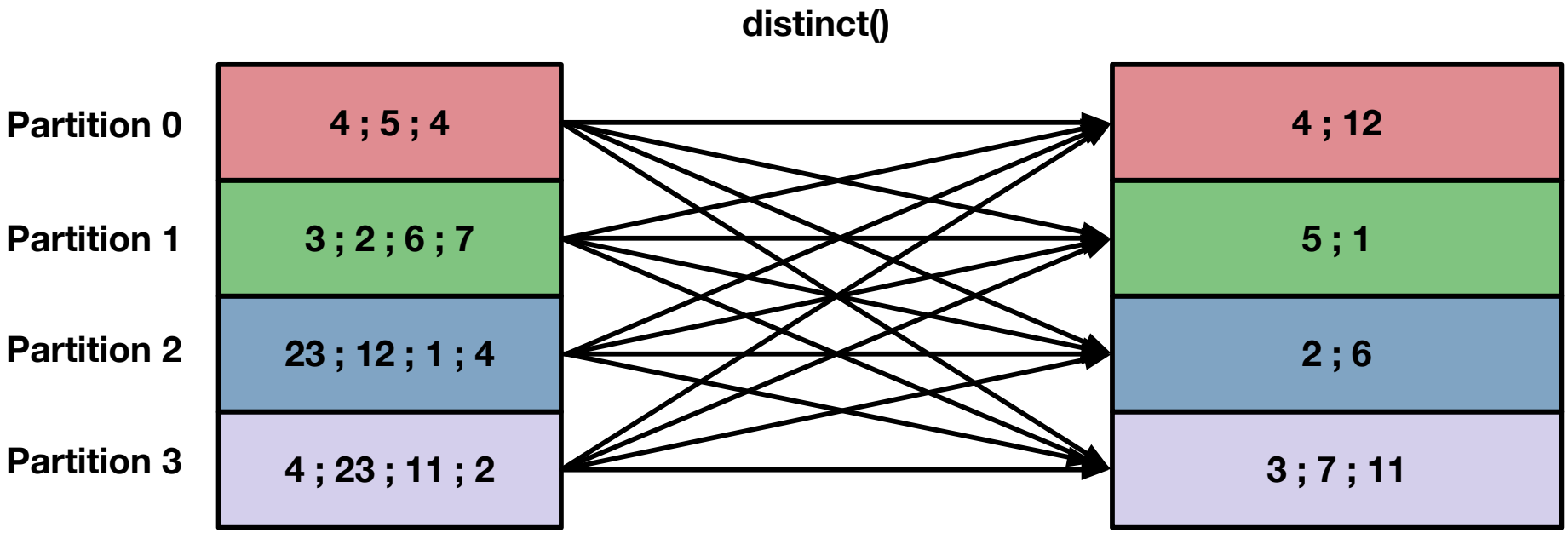
`distinct()` prend en argument un RDD et renvoie un **nouveau RDD** contenant les éléments du RDD donné **sans répétitions**.



☞ Contrairement aux transformations précédentes, `distinct` entraîne un **brassage de données** sur le réseau.

À propos du brassage de données ★

À quelle partition sera affecté l'élément 23 dans le RDD obtenu après l'application de la transformation `distinct`?



À propos du brassage de données ★

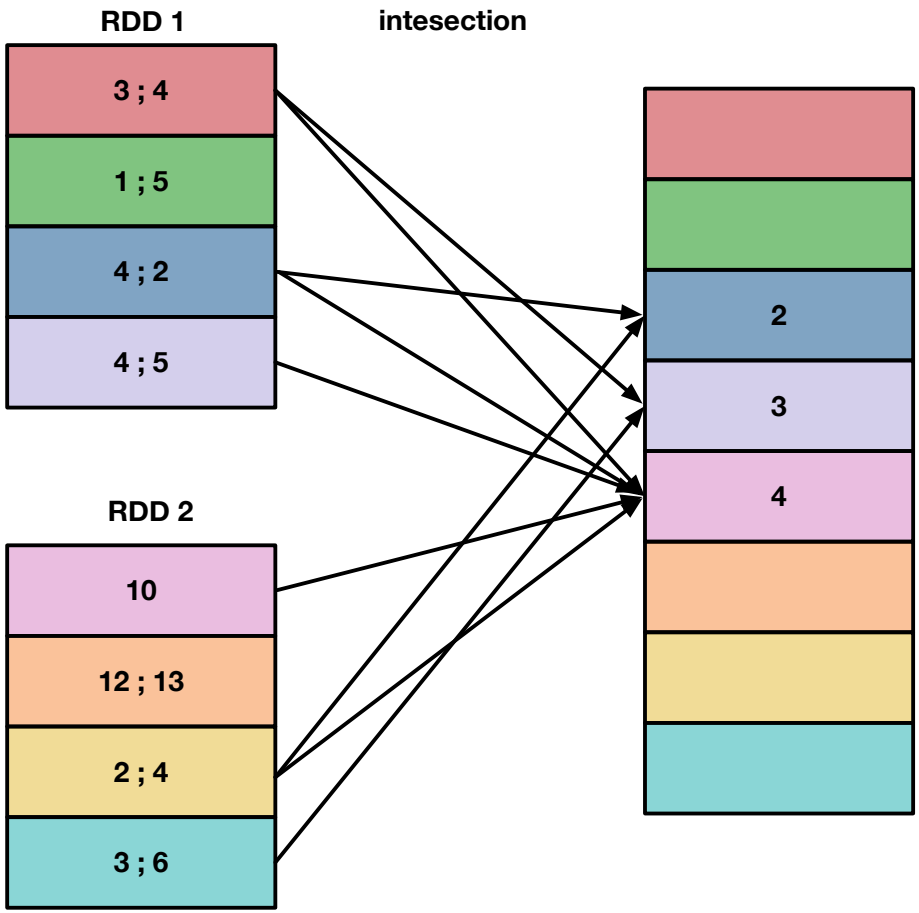
L'élément 23 sera affecté à la **partition 3**.

Lors du brassage, la **partition de destination** p d'un élément K dans un RDD avec n partitions est calculée comme suit:

$$p = hashCode(K) \ mod \ n$$

Transformations: `intersection`

`intersection()` prend en arguments deux RDDs et renvoie un **nouveau RDD** contenant les éléments qu'on trouve dans les deux RDD.



Transformations de type *narrow* ★

Une transformation de type **narrow** calcule le contenu de chaque partition du RDD renvoyé en utilisant le contenu d'une seule partition du RDD en argument.

- Les transformations de type narrow sont **peu coûteuses**.
- Pas de brassage de données.

Parmi les transformations étudiées, lesquelles sont de type narrow ?

Transformations de type *narrow* ★

Une transformation de type **narrow** calcule le contenu de chaque partition du RDD renvoyé en utilisant le contenu d'une seule partition du RDD en argument.

- Les transformations de type narrow sont **peu coûteuses**.
- Pas de brassage de données.

`filter`, `map`, `flatMap` et `union` sont des transformations de type narrow.

Transformations de type *wide* ★

Une transformation de type **wide** calcule le contenu de chaque partition du RDD renvoyé en utilisant le contenu de plusieurs partitions du RDD en argument.

- Les transformations de type wide sont plus **coûteuses**.
- Les données sont **brassées** via le réseau du cluster.

Parmi les transformations étudiées, lesquelles sont de type wide ?

Transformations de type *wide* ★

Une transformation de type **wide** calcule le contenu de chaque partition du RDD renvoyé en utilisant le contenu de plusieurs partitions du RDD en argument.

- Les transformations de type wide sont plus **coûteuses**.
- Les données sont **brassées** via le réseau du cluster.

`distinct` et `intersection` sont des transformations de type wide.

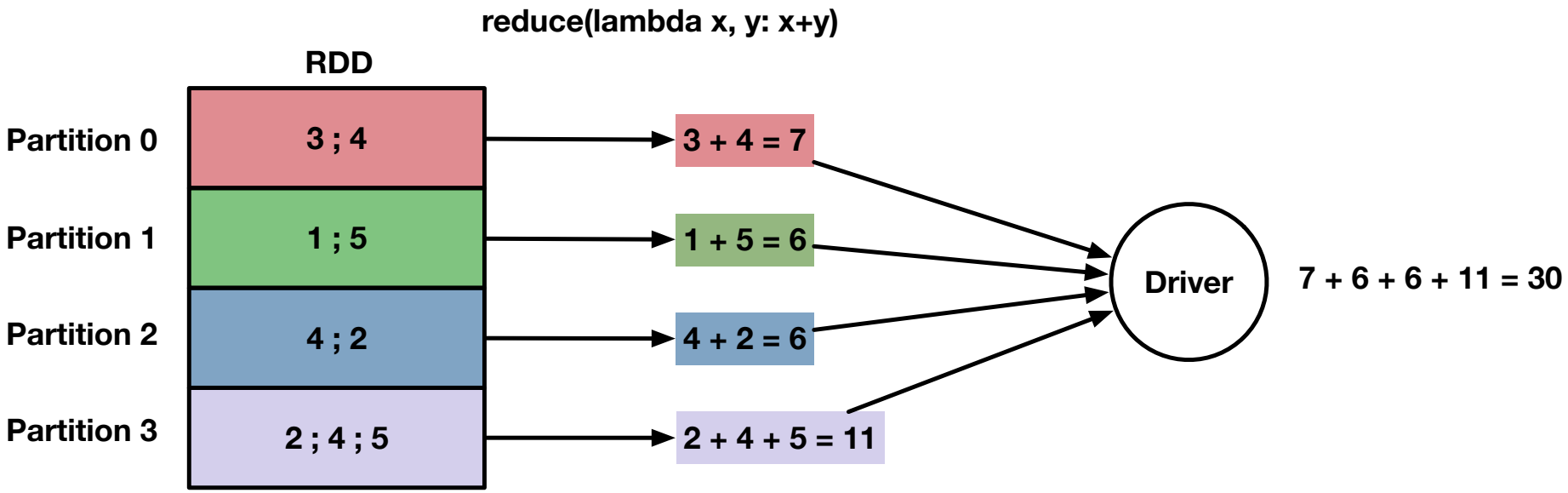
Actions

Une **action** est une opération qui prend un RDD en argument et renvoie une valeur au **Driver** après avoir exécuté un calcul sur ce RDD.

- Le résultat d'une action est envoyé au Driver.
- Si le résultat est une liste de valeurs, **toutes les valeurs** sont envoyées au Driver.
- Le résultat d'une action peut également être **écrit sur disque**.
- Les écritures sur disque peuvent se faire sur le **système de fichiers local** ou sur **HDFS**.

Actions : `reduce`

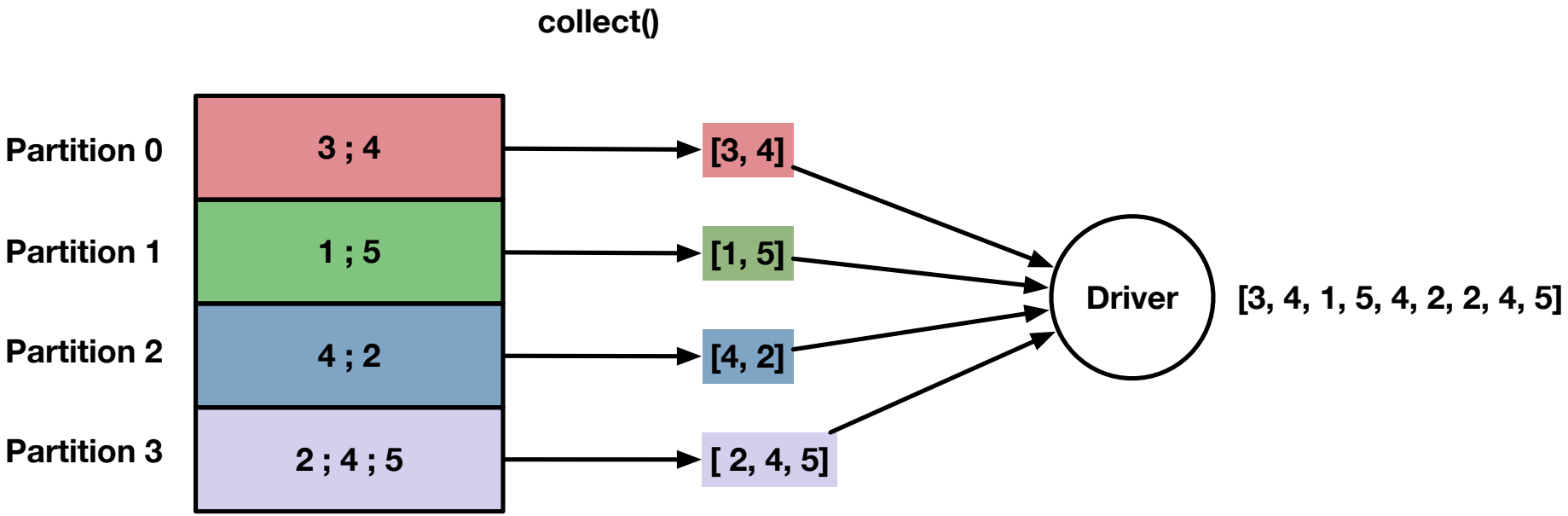
`reduce()` prend un RDD et une fonction f en arguments et applique la fonction f à tous les éléments du RDD deux à deux.



- La fonction f **doit** prendre **2 arguments**.
- Le type de la valeur renvoyée par la fonction f **doit être le même** que le type des éléments du RDD.

Actions : `collect`

`collect()` prend en argument un RDD et renvoie la **liste** des éléments du RDD.



`collect()` est-elle une action inoffensive ? ★

Quels sont les risques, s'il y en a, en invoquant `collect()` sur un grand RDD ?

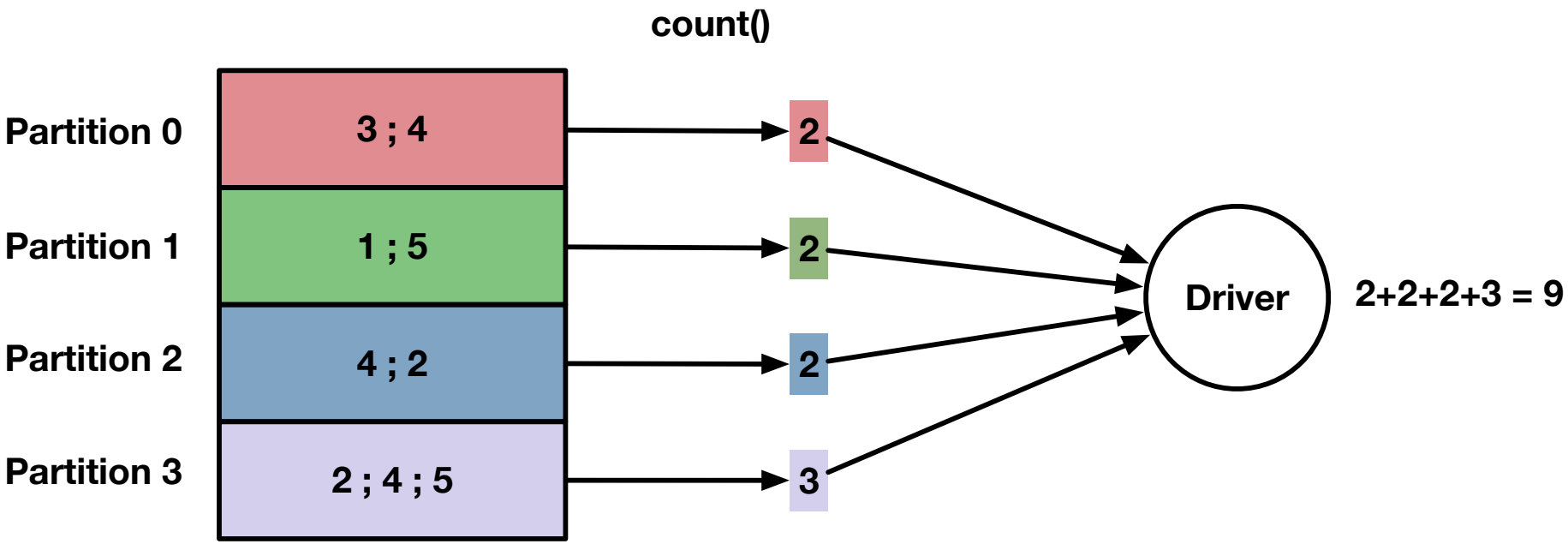
`collect()` est-elle une action inoffensive ? ★

Quels sont les risques, s'il y en a, en invoquant `collect()` sur un grand RDD ?

- Trafic réseau élevé.
- La mémoire du Driver peut ne pas suffire à stocker tous les éléments du RDD.

Actions : count

`count()` prend un RDD en argument et renvoie le nombre d'éléments du RDD.



Compréhension du code ★

Que fait le code suivant ?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.map(lambda x: x.capitalize())
```

Compréhension du code ★

Que fait le code suivant ?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.map(lambda x: x.capitalize())
```

- `r2` est un RDD (résultat d'une transformation).
- `r2` a autant d'éléments que `r1`.
- Chaque élément de `r2` est un mot de `r1` dont la première lettre est majuscule.

Compréhension du code ★

Que fait le code suivant ?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.filter(lambda x: len(x) > 10)
```

Compréhension du code ★

Que fait le code suivant ?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.filter(lambda x: len(x) > 10)
```

- `r2` est un RDD (résultat d'une transformation).
- `r2` a moins d'éléments que `r1`.
- `r2` contient seulement les éléments de `r1` qui ont plus de 10 caractères.

Compréhension du code ★

Que fait le code suivant ?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.reduce(lambda x, y: f"{x} - {y}")
```

Compréhension du code ★

Que fait le code suivant ?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.reduce(lambda x, y: f"{x} - {y}")
```

- `r2` est un mot, **pas un RDD** (résultat d'une action).
- `r2` est le mot "computer science - geology - chemistry - biology - astronomy".

Compréhension du code ★

Que fait le code suivant?

```
r1 = sc.parallelize(["informatique", "géologie", \
    "chimie", "biologie", "astronomie"])
r2 = r1.reduce(lambda x, y: [x + y])
```

Compréhension du code ★

Que fait le code suivant?

```
r1 = sc.parallelize(["informatique", "géologie", \
    "chimie", "biologie", "astronomie"])
r2 = r1.reduce(lambda x, y: [x + y])
```

Le code est incorrect, car le type de retour (`list`) de la fonction `reduce` est différent du type des éléments du RDD en argument (`string`).

Compréhension du code ★

Que fait le code suivant?

```
r1 = sc.parallelize(["auteur", "titre", "édition"])
r2 = r1.flatMap(lambda x: [c for c in x])
```

Compréhension du code ★

Que fait le code suivant?

```
r1 = sc.parallelize(["auteur", "titre", "édition"])
r2 = r1.flatMap(lambda x: [c for c in x])
```

- `r2` est un RDD (résultat d'une transformation).
- Chaque élément de `r2` est une lettre d'un mot de `r1`. Comment cela serait-il différent si nous avions utilisé `map` au lieu de `flatMap`?

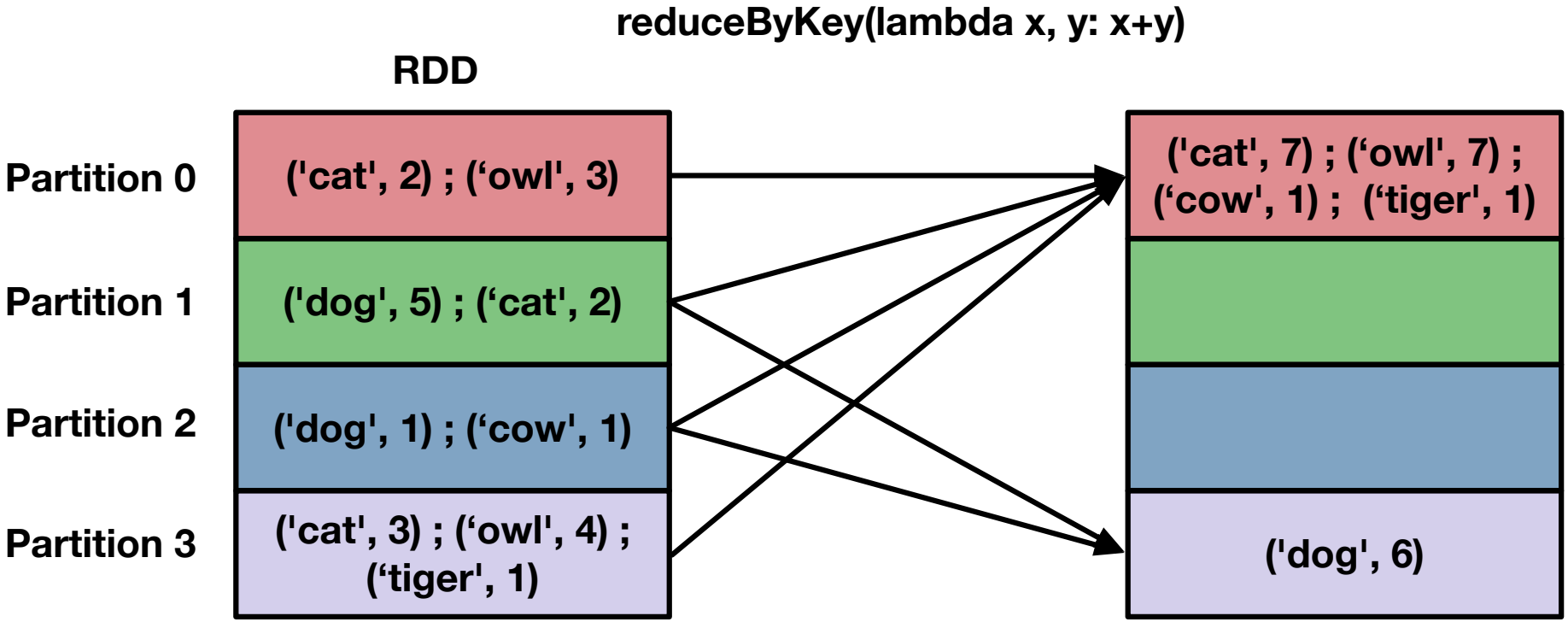
RDDs clé-valeur

Les **RDDs clé-valeur** (également appelés **Pair RDDs**) sont des RDDs où chaque élément est un couple (k, v), k étant la clé et v la valeur.

- Les RDDs clé-valeur sont des éléments de base importants dans de nombreuses applications.
- On peut utiliser sur des RDDs clé-valeur toutes les transformations et les actions étudiées précédemment.
- D'autres transformations et actions propres aux RDDs clé-valeur existent.

Transformations des RDDs clé-valeur: `reduceByKey`

`reduceByKey` prend en arguments un RDD avec des paires (K, V) et une fonction f et renvoie un **nouveau RDD** de paires (K, V) où les valeurs pour chaque clé sont agrégées en utilisant $f : (V, V) \rightarrow V$.



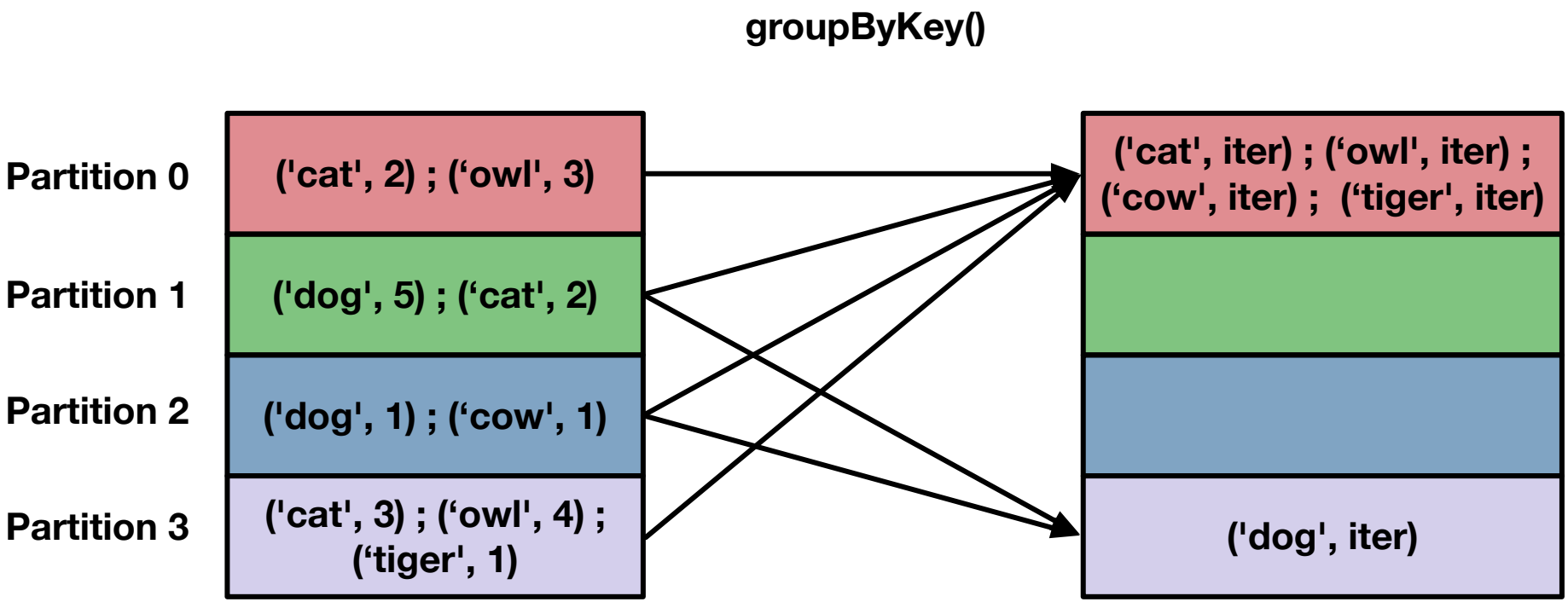
Transformations des RDDs clé-valeur: `reduceByKey`

- Le RDD en argument a un certain nombre de partitions n.
- Aucune hypothèse ne peut être faite sur l'appartenance des éléments à une partition.
- Le RDD retourné par `reduceByKey` est **partitionné par hachage**. Chaque élément appartient à une partition précise.
- Le numéro de partition p d'une paire (K, V) est déterminé comme suit:

$$p = hashCode(K) \ mod \ n$$

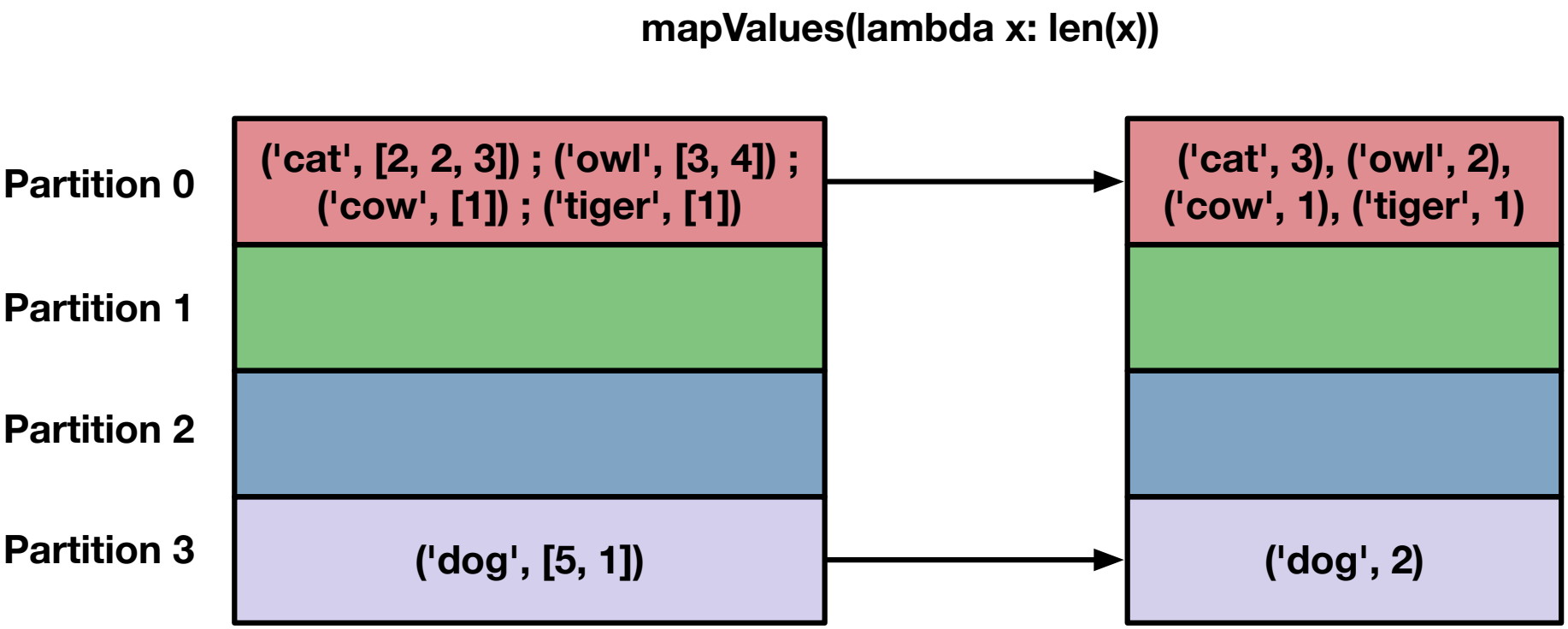
Transformations des RDDs clé-valeur: `groupByKey`

`groupByKey` prend en argument un RDD avec des paires (K, V) et renvoie un **nouveau RDD** de paires $(K, Iterable < V >)$.



Transformations des RDDs clé-valeur: `mapValues`

`mapValues` prend en arguments un RDD avec des paires (K, V) et une fonction f et renvoie un **nouveau RDD** où la fonction f est appliquée à chaque valeur V (les clés ne sont pas modifiées).



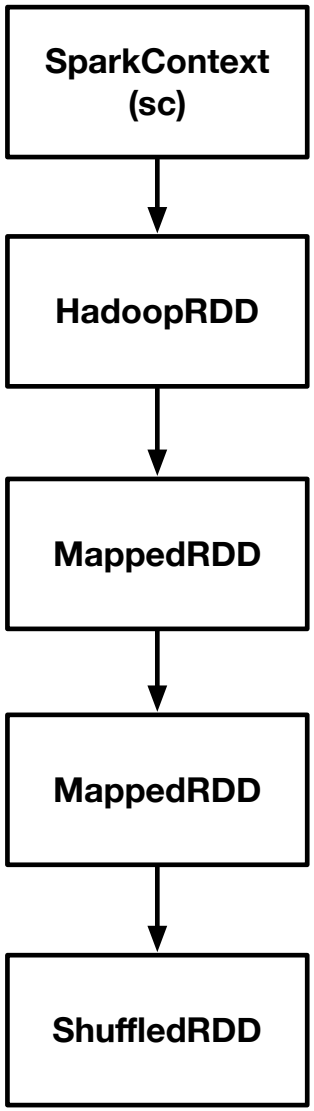
Exemple: comptage de mots

```
def word_count(input_file):
    text = sc.textFile(input_file)
    return text.flatMap(lambda line: line.split(" "))\
              .map(lambda word: (word, 1))\
              .reduceByKey(lambda x, y: x+y)
```

- La fonction `textFile` lit un fichier texte et met son contenu dans un RDD.
- Deux transformations de type narrow (`flatMap` et `map`) et une transformation de type wide (`reduceByKey`).

☞ Spark maintient un **plan d'exécution logique** (appelé **RDD lineage**) représenté à l'aide d'un **graphe orienté acyclique (DAG)**.

Plan d'exécution logique



Spark dispose d'un **ordonnanceur DAG** qui divise le graphe en plusieurs **étapes**.

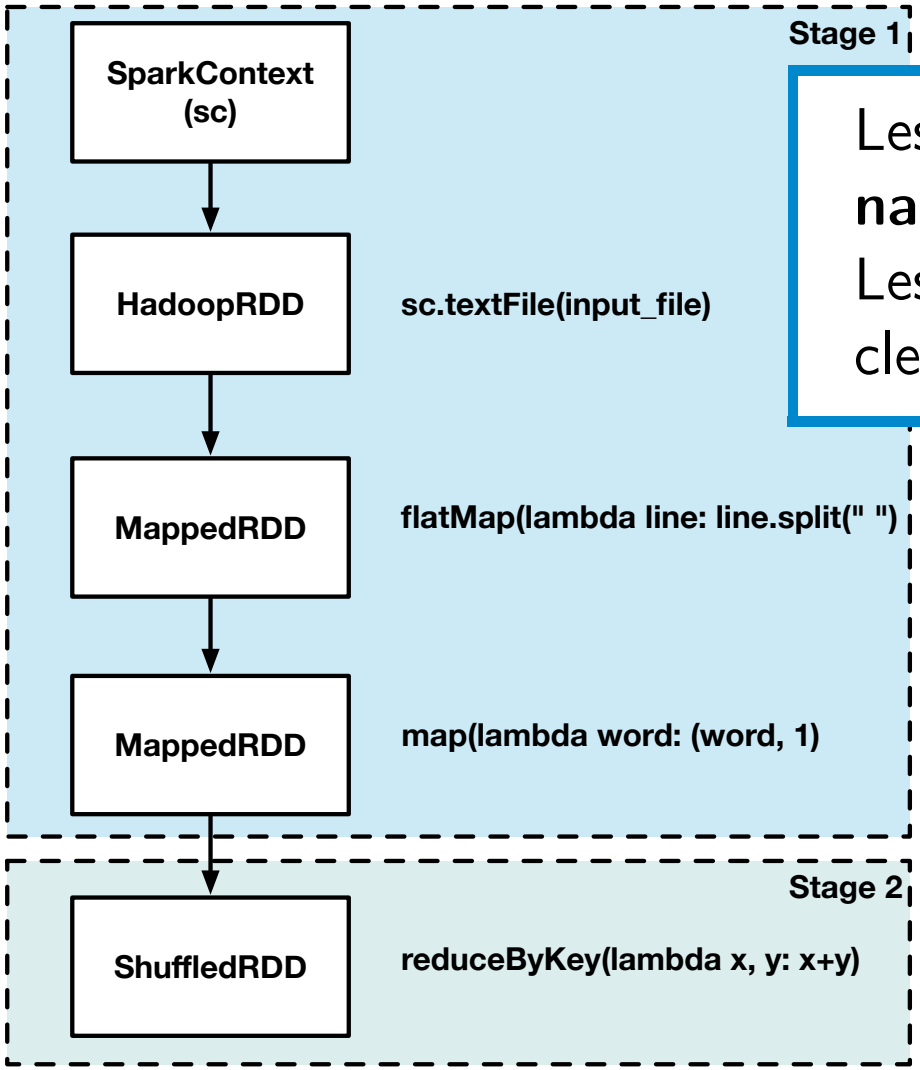
sc.textFile(input_file)

flatMap(lambda line: line.split(" ")

map(lambda word: (word, 1)

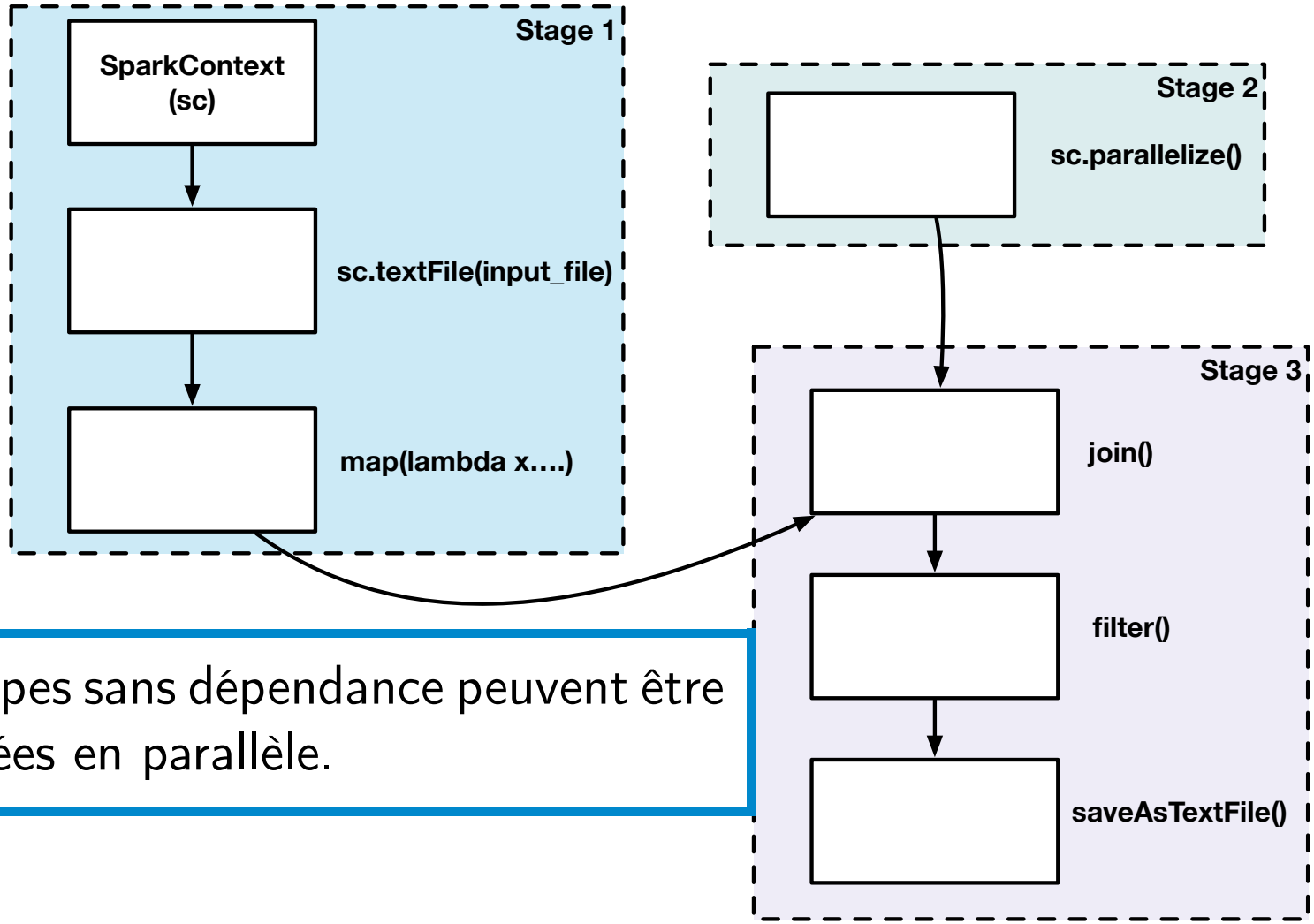
reduceByKey(lambda x, y: x+y)

Plan d'exécution logique: étapes



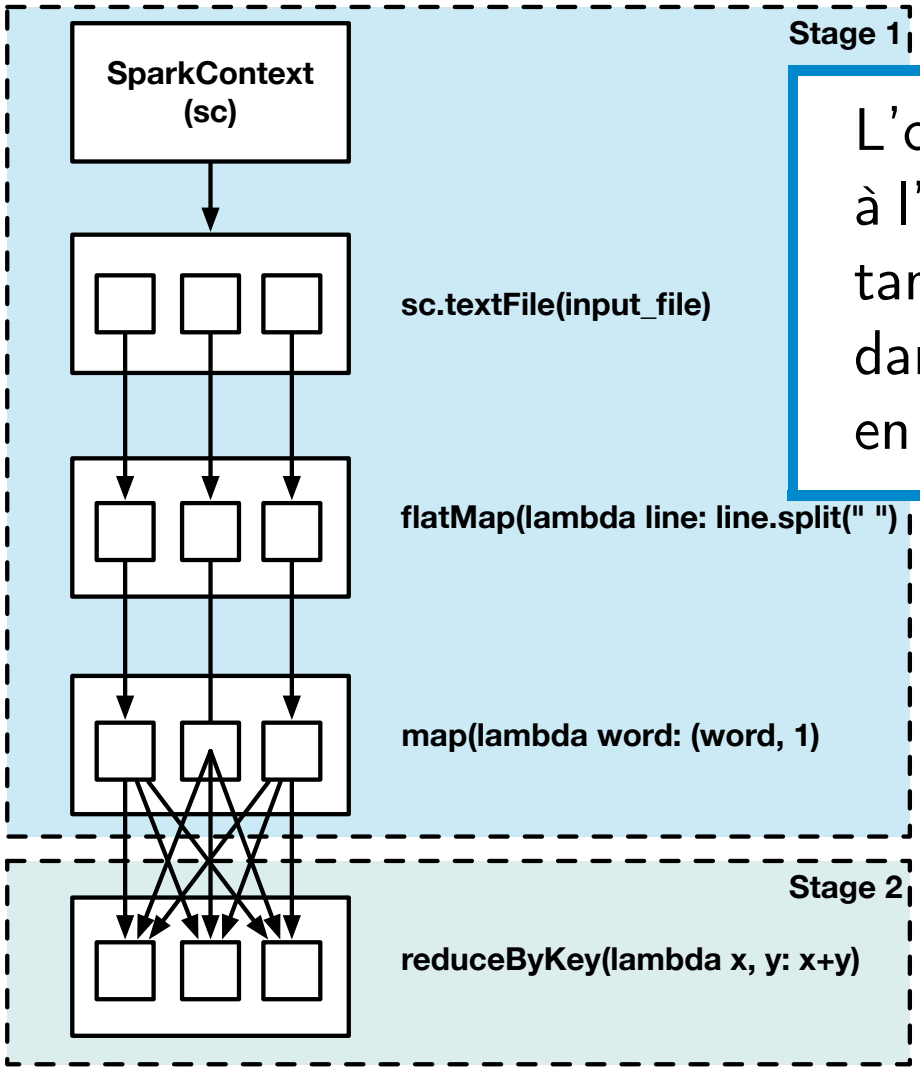
Les séquences de transformations de type **narrow** sont attribuées à la **même étape**. Les transformations de type **wide** déclenchent toujours une **nouvelle étape**.

Plan d'exécution logique: étapes



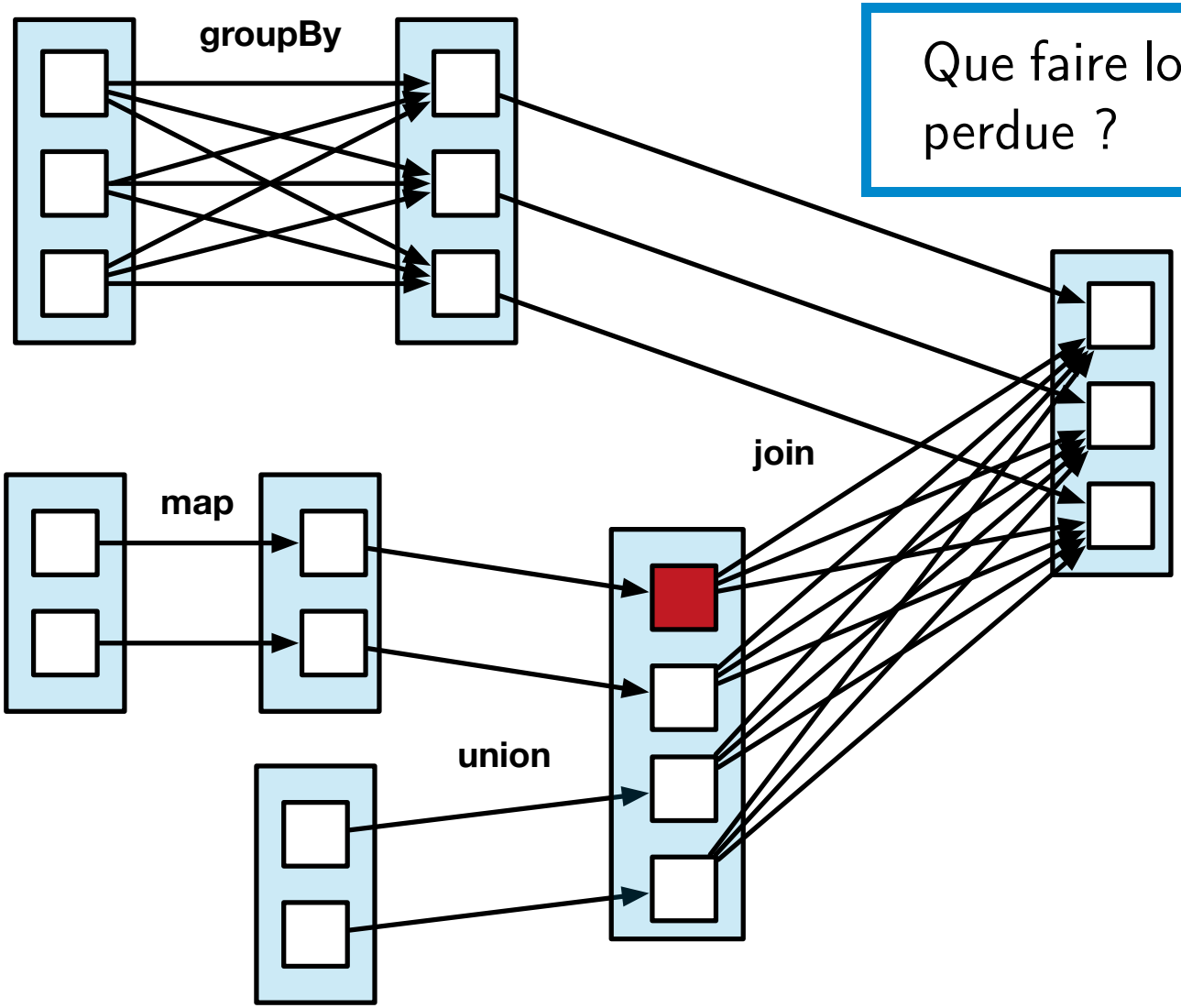
Les étapes sans dépendance peuvent être exécutées en parallèle.

Plan d'exécution logique: tâches



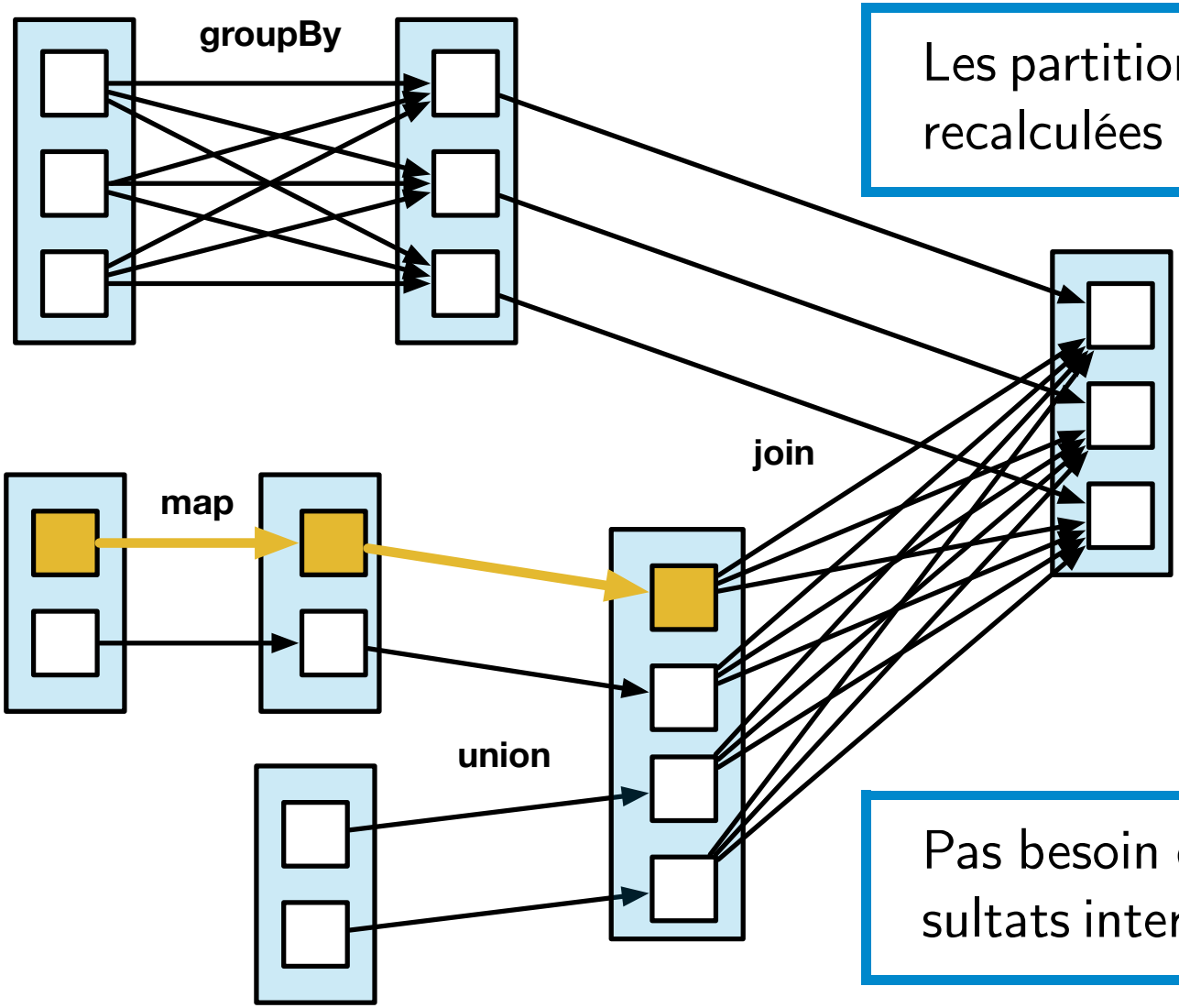
L'ordonnanceur DAG soumet les étapes à l'**ordonnanceur de tâches**. Crée autant de tâches qu'il y a de partitions dans le RDD. Les tâches sont exécutées en parallèle.

Plan d'exécution logique: tolérance aux pannes



Que faire lorsqu'une partition est perdue ?

Plan d'exécution logique: tolérance aux pannes



Les partitions perdues peuvent être recalculées grâce au graphe.

Pas besoin de sauvegarder les résultats intermédiaires sur le disque.

Évaluation paresseuse

- Dans Spark, les transformations sont **évaluées de manière paresseuse**.

Lorsqu'une transformation est invoquée, Spark **ne l'exécute pas** immédiatement. Les transformations ne sont exécutées que lorsque **la première action** est appelée.

- Un RDD peut être considéré comme un ensemble d'*instructions* sur la manière de calculer les données à travers les transformations.
- L'évaluation paresseuse aide à **réduire le nombre de passages** nécessaires pour charger et transformer les données.

Évaluation paresseuse : exemple ★

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
print("Number of exception lines ", nb_lines)
```

Que se passe-t-il si Spark exécute **immédiatement** chaque transformation ?

Évaluation paresseuse : exemple ★

- L'invocation de `sc.textFile()` ne charge pas immédiatement les données.
- La transformation `filter()` n'est pas appliquée lorsqu'elle est invoquée.
- Les transformations ne sont appliquées que lorsque l'action `count()` est invoquée.
- **Seules les données** qui satisfont la contrainte du `filter` sont chargées à partir du fichier.

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
print("Number of exception lines ", nb_lines)
```

☞ **Sans évaluation paresseuse**, nous aurions chargé dans la mémoire centrale **tout le contenu** du fichier.

Évaluation paresseuse : conséquences ★

- Le code suivant invoque **deux actions** : lesquelles ?
- Que se passe-t-il lorsque nous invoquons la **deuxième action** ?

Exemple

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
exceptions.collect()
```

Évaluation paresseuse : conséquences ★

- Avec l'évaluation paresseuse, les transformations sont calculées **chaque fois** qu'une action est invoquée sur un RDD donné.
- Dans l'exemple suivant, toutes les transformations sont calculées lorsque nous invoquons la fonction `count()` **et** la fonction `collect()`.

Exemple

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
exceptions.collect()
```

Pour éviter de calculer plusieurs fois les transformations, nous pouvons **persister** les données.

Persister les données

- Persister les données signifie **mettre en cache** le résultat des transformations.
  - Soit en mémoire centrale (par défaut), soit sur disque, ou les deux.
- Si un serveur du cluster échoue, Spark **recalcule** les partitions persistées.
  - Nous pouvons **répliquer** les partitions persistées sur d'autres serveurs pour faire face aux pannes sans besoin de recalculer.

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
exceptions.persist(StorageLevel.MEMORY_AND_DISK)
nb_lines = exceptions.count()
exceptions.collect()
```

- `persist()` est appelé juste avant la première action.
- `persist()` ne force pas l'évaluation des transformations.
- `unpersist()` peut être appelé pour supprimer les partitions persistées.