Université Mohammed V- Rabat
Ecole Mohammadia d'Ingénieurs
Département Génie Informatique
Filière Génie Informatique et Digitalisation

Traitement Big Data

Pr. N. EL FADDOULI
nfaddouli@gmail.com

2023-2024

Chapitre 3: Les Resilient Distributed Dataset (RDD)

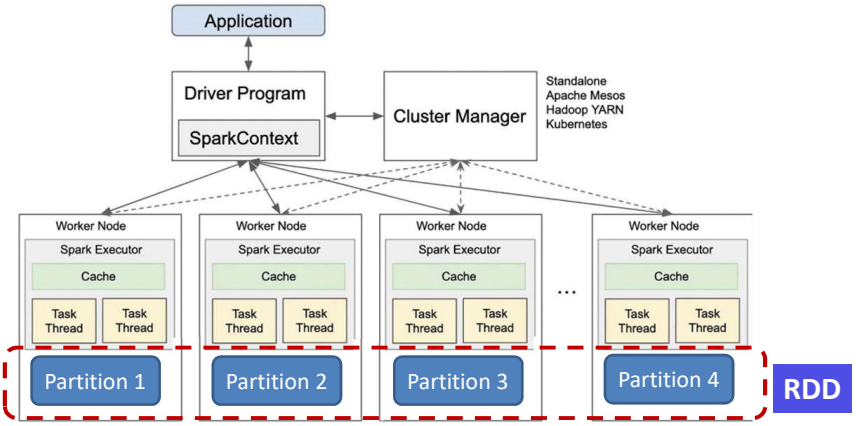
- ❑ Présentation des RDD.
- ❑ Créer, manipuler et réutiliser des RDD.
- ❑ Caractéristiques des RDD.
- ❑ Accumulateurs et variables broadcastées.

TP :

Manipulation de différents Datasets à l'aide de RDD et utilisation de l'API fournie par Spark.

RDD: Présentation

- Un **RDD** (**Resilient Distributed Dataset**) est **une structure de données** qui permet de stocker des données de manière **répartie** sur un cluster de machines, de les traiter de manière **parallèle** et de **tolérer les pannes** de manière transparente.



RDD: Présentation

- Les **RDDs** forment la **base du modèle de programmation** de Spark et sont utilisés pour effectuer des **opérations de traitement (pipeline Spark)** de données distribuées dans Spark.
- Un **RDD** est composé des éléments suivants :
  - **Données sous-jacentes**: Un RDD contient les données qu'on veut tr… …les peuvent être de différents types (des lignes de texte, des objets Python, des valeu… …ériques, etc). Les données sont réparties sur plusieurs nœuds du cluster Spark…
  - **Partitions**: Les données d'un RDD sont divisées en p… …*s petits*). Chaque partition est une unité de base pour le traitement … Par exe… …ns HDFS chaque partition d'un RDD peut correspondre à un bloc H…
  - **Métadonnées**: Ces métadonnées sont stockées da… …émoire du… On y trouve des informations (*identifiants et emplacements des pa*… …e **Driver** pour la **planifications** et **l'ordonnancement des tâches** l'appl… …ent ces **RDDs** .

RDD: Présentation

❑ Ces métadonnées incluent:

- **Identifiants et emplacements des partitions**: Ils sont utilisés pour localiser les partitions des RDD manipulés.
- **Fonction de partitionnement**: Un RDD peut également inclure une fonction de partitionnement personnalisée qui spécifie comment les données sont réparties entre les partitions (*Par exemple: partitionnement basé sur le hachage*).

RDD: Présentation

- **Informations de Dépendances**: Informations sur les dépendances des RDD, c'est-à-dire **comment chaque RDD a été dérivé d'autres RDD**. Cela permet de **suivre l'ensemble du flux de données** et de **garantir la tolérance aux pannes**.
  - Ce sont des informations de **Lineage ou Lignage** (***Lineage Information*** ou ***RDD Lineage***) qui sont un **enregistrement des opérations** appliquées à des données distribuées pour **produire** un RDD spécifique. Ces informations permettent de récupérer les partitions de données perdues en cas de panne de nœud.
  - Ces informations de **lineage** sont cruciales pour **recalculer les partitions perdues** et garantir la **tolérance aux pannes** dans Spark.

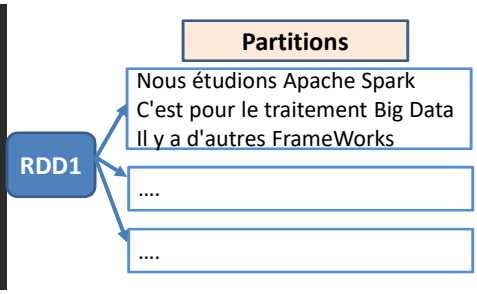
RDD: Création à partir d'un fichier

❑ On peut créer un **RDD** à partir du contenu d'un **fichier** (***local***, ***HDFS***, …) chargé pour traitement

```
# Importer SparkSession
from pyspark.sql import SparkSession

# Créer une session Spark
spark = SparkSession.builder.appName("Exemple Map")\
        .master("local[*]").getOrCreate()

# Charger un fichier  texte local en tant que RDD
rdd1 = spark.sparkContext.textFile("test.txt")
```




```
# Importer SparkContext
from pyspark import SparkContext

# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exemple Map")

# Charger un fichier  texte local en tant que RDD
rdd1 = sc.textFile("test.txt")
```

❑ Le **nombre de partitions** dépend du **nombre de cœurs CPU** disponibles en local ou dans le cluster, de la **localisation** et la **taille des données**

❑ On peut le personnaliser dans l'application: rdd1 = rdd1.**repartition**(4)

RDD: Création à partir d'une liste de valeurs

❑ On peut créer un RDD à partir de données (*texte*, *entier*, …) **fixées dans l'application**

```
from pyspark import SparkContext

# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exemple Map")

# Données à traiter
data = ["Nous sommes à l'EMI", "Au département informatique", "On apprend Spark"]

# Crée un  RDD à partir du contenu de data
rdd1 = sc.parallelize(data)
```

❑ Le nombre par défaut des partitions obtenues par **parallelize** est égal au nombre de cœurs CPU logiques (en local)

❑ On peut personnaliser le nombre de partitions obtenues par **parlalelize** en renseignant le deuxième paramètre comme suit: rdd1 = sc.**parallelize**(data, **1**)

RDD: Transformations

- On peut créer un nouveau RDD à partir d'un autre RDD sachant qu'**on ne peut pas modifier un RDD** ➔ **Un RDD est en lecture seul**
- Une **transformation** est une **fonction** qui permet de créer un nouveau RDD à partir d'un RDD existant.
- Exemples de transformations permettant d'obtenir un nouveau RDD:
  - **map**: appliquer une **fonction** à **chaque élément** d'un RDD pour produire un nouveau RDD.
  - **filter**: filtrer les éléments d'un RDD en fonction d'une condition donnée.
  - **flatMap**: appliquer une fonction à chaque élément du RDD pour produire un nouveau RDD avec une liste d'éléments en sortie pour chaque élément en entrée.
  - **sortBy**: trier les éléments du RDD en fonction d'une condition donnée
  - **union**: Fusionne deux RDD en un seul.
  - **distinct**: Retourne un nouveau RDD avec des éléments uniques.
  - …

La transformation map(*fonction*)

- Elle permet d'appliquer une **fonction** à chaque **élément** (*ligne, tuple, objet*, …) du RDD en **entrée** pour créer un élément du RDD de **sortie**.
- La fonction doit avoir **un seul argument** et retourne une valeur qui peut être scalaire ou complexe (tuple, liste, …)
- **Exemple 1**:

Chaque ligne converti donnera lieu à un tableau de mots

```
from pyspark import SparkContext
import re
# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exercice 1")
# Définition de la fonction de map
def map_fct_mot1(mots):
    return len(mots)
rdd1 = sc.textFile('D:/data/test.txt')
rdd2 = rdd1.map(lambda ligne: re.split(r'[;,:.\s+]', ligne))\
          .map(map_fct_mot1)
```

Chaque tableau de mots donnera lieu à un entier (*sa taille*)

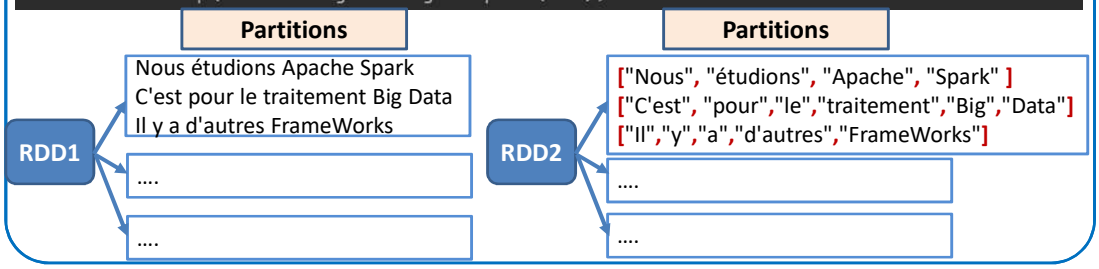
lambda mots: len(mots)

Le paramètre d'appel est implicitement un élément de rdd1

La transformation map(*fonction*)

❑ **Exemple 2**: Appliquer la transformation **map** pour diviser chaque ligne en mots en appliquant une fonction anonyme (**lambda**) qui **retourne un tableau des mots** d'une ligne (argument)

```
# Importer SparkSession
from pyspark.sql import SparkSession
# Créer une session Spark
spark = SparkSession.builder.appName("Exemple Map").master("local[*]").getOrCreate()
# Charger un fichier  texte local en tant que RDD
rdd1 = spark.sparkContext.textFile("test.txt")
# Appliquer la transformation map pour diviser chaque ligne en mots
rdd2 = rdd1.map(lambda ligne: ligne.split(" "))
```



La transformation flatMap(*fonction*)

❑ Elle permet d'appliquer une **fonction** à chaque **élément** (*ligne, tuple, objet*, …) du RDD en **entrée** pour créer un élément du RDD de **sortie**.

❑ La fonction doit avoir **un seul argument** et retourne une valeur qui peut être scalaire ou complexe (tuple, liste, …)

Chaque ligne converti donnera lieu à un tableau de mots dont chacun sera un élément du RDD de sortie

❑ **Exemple**:

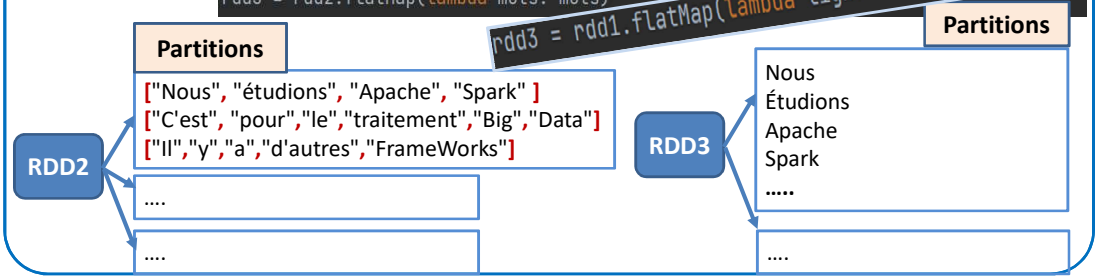
```
from pyspark import SparkContext
import re
# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exercice 1")
# Définition de la fonction de map
def map_fct_mot1(mots):
    return len(mots)
rdd1 = sc.textFile('D:/data/test.txt')
rdd2 = rdd1.flatMap(lambda ligne: re.split(r'[;,:.\s+]', ligne))\
          .map(map_fct_mot1)
print(rdd2.collect())
```

Chaque mot donnera lieu à un entier (*sa taille*)

La transformation flatMap(*fonction*)

❑ **Exemple 2**: Appliquer la transformation **flatmap** pour diviser chaque ligne (élément de ***rdd1***) en **liste** de mots pour avoir les éléments de ***rdd2*** dont chacun (une liste) sera aplati pour avoir les éléments de ***rdd3***

```
# Importer SparkSession
from pyspark.sql import SparkSession
# Créer une session Spark
spark = SparkSession.builder.appName("Exemple Map").master("local[*]").getOrCreate()
# Charger un fichier  texte local en tant que RDD
rdd1 = spark.sparkContext.textFile("test.txt")
# Appliquer la transformation map pour diviser chaque ligne en mots
rdd2 = rdd1.map(lambda ligne: ligne.split(" "))
rdd3 = rdd2.flatMap(lambda mots: mots)
```



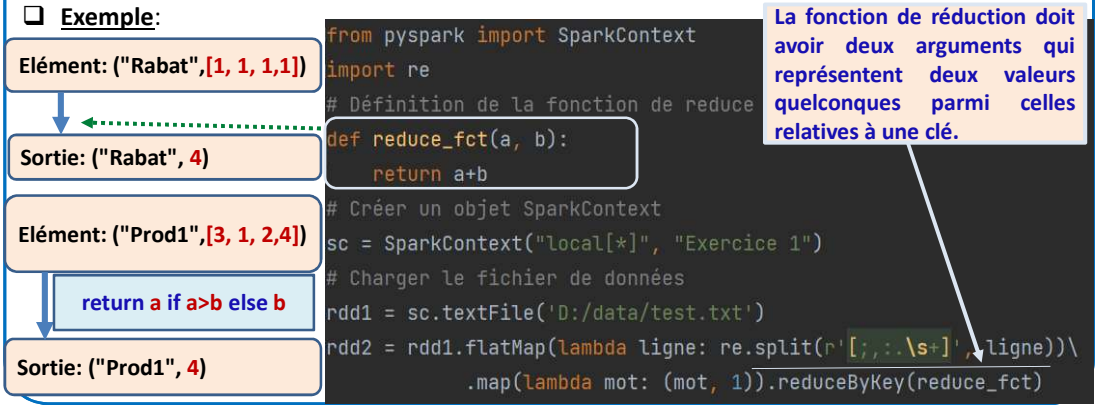
La transformation reduceByKey(*fonction, [numPartitions]*)

❑ Les **paires clé-valeur** du RDD en entrée sont d'abord regroupées **selon leurs clés** (**Shuffle**)

❑ Une **fonction de réduction** (***Premier paramètre de reduceByKey***) est ensuite appliquée à toutes les **valeurs associées à chaque clé** pour avoir **un élément du RDD de sortie**.

❑ Le second paramètre facultatif représente le **nombre de partitions maximal** à obtenir après le shuffle: **spark.conf.set**("**spark.sql.shuffle.partitions**", 16)

❑ **Exemple**:



La transformation groupByKey(*[numPartitions]*)

- C'est une opération de transformation pour **grouper** toutes les paires **clé-valeur** d'un RDD par clé pour avoir un RDD en sortie dont les élément sont des paires clé-valeur et où la valeur est une **liste** par exemple ("Ipad", [3, 4, 2])
- Elle peut entraîner des problèmes de performance à case d'un déplacement important de données sur le réseau.
- **Exemple**:

```
from pyspark import SparkContext
# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exemple groupByKey")
# Créer un RDD avec des paires clé-valeur
data = [("fruit", "pomme"), ("fruit", "orange"),\
        ("animal", "lion"), ("animal", "elephant")]
rdd1 = sc.parallelize(data)
# Regrouper les valeurs par clé
rdd2 = rdd1.groupByKey()
# Afficher les résultats
for key, values in rdd2.collect():
    print(f"{key}: {list(values)}")
```

Résultat

```
fruit: ['pomme', 'orange']
animal: ['lion', 'elephant']
```


La transformation filter(*fonction_bool*)

- La transformation filter crée un nouveau RDD en filtrant les éléments d'un RDD existant en fonction d'une condition spécifiée.
- Cela permet de sélectionner uniquement les éléments qui satisfont la condition.
- **Exemple**:

```
from pyspark import SparkContext
# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exemple filter")
# Créer un RDD avec des paires clé-valeur
data = [("fruit", "pomme"), ("fruit", "orange"),\
        ("animal", "lion"), ("animal", "elephant")]
rdd1 = sc.parallelize(data)
# filtrer les animaux
rdd2 = rdd1.filter(lambda x: x[0]=="animal")
# Afficher les résultats
for key, value in rdd2.collect():
    print(f"{key}: {value}")
```

Résultat

```
animal: lion
animal: elephant
```

La transformation sortByKey ([**ascending**=*True* | *False*])

- La transformation **sortByKey** crée un RDD en triant les paires clé-valeur de départ par clé.
- Le tri par défaut est croissant mais on peut l'indiquer avec le paramètre optionnel **ascending= True** ou **False**.
- **Exemple**:

```
from pyspark import SparkContext
# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exemple filter")
# Créer un RDD avec des paires clé-valeur
data = [("fruit", "pomme"), ("légume", "tomate"),\
        ("fruit", "orange"), ("animal", "tigre"),\
        ("animal", "elephant"), ("animal", "lion")]
rdd1 = sc.parallelize(data)
# Trier par clé dans l'ordre décroissant
rdd2 = rdd1.sortByKey(ascending=False)
# Afficher les résultats
for t in rdd2.collect():
    print(t)
```

Résultat

```
('légume', 'tomate')
('fruit', 'pomme')
('fruit', 'orange')
('animal', 'tigre')
('animal', 'elephant')
('animal', 'lion')
```


La transformation sortBy (*Fonction, [**ascending**=True | False]*)

- La transformation **sortBy** crée un RDD de sortie en triant les paires clé-valeur du RDD d'entrée par la **valeur de retour** de la fonction qui peut retourner la **clé**, la **valeur** ou **autre**.
- Le tri par défaut est croissant mais on peut l'indiquer avec le paramètre optionnel **ascending= True** ou **False**.
- **Exemple**:

```
from pyspark import SparkContext
# Créer un objet SparkContext
sc = SparkContext("local[*]", "Exemple filter")
# Créer un RDD avec des paires clé-valeur
data = [("fruit", "pomme"), ("légume", "tomate"),\
        ("fruit", "orange"), ("animal", "tigre"),\
        ("animal", "elephant"), ("animal", "lion")]
rdd1 = sc.parallelize(data)
# Trier par valeur dans l'ordre croissant
rdd2 = rdd1.sortBy(lambda x: x[1])
# Afficher les résultats
for t in rdd2.collect():
    print(t)
```

Résultat

```
('animal', 'elephant')
('animal', 'lion')
('fruit', 'orange')
('fruit', 'pomme')
('animal', 'tigre')
('légume', 'tomate')
```

La transformation mapPartitions *(**Fonction**, [**ascending**=True | False])*

- La transformation **mapPartitions** permet d'appliquer une fonction à **chaque partition** de du RDD d'entrée et non pas à chaque élément comme **map**.
- Cela peut être plus efficace que **map** lorsqu'on a besoin de faire des opérations par partition au lieu de les appliquer sur chaque élément individuel.
- **Exemple**:

```
from pyspark import SparkContext
# Initialiser le contexte Spark
sc = SparkContext("local", "mapPartitions Example")
# Données à traiter
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Créer un RDD de 3 partitions
rdd1 = sc.parallelize(data, 3)
# Définir une fonction à appliquer sur chaque partition
def map_partition_func(iterator):
    yield sum(iterator)  # somme des éléments dans la partition
# Appliquer mapPartitions
rdd2 = rdd1.mapPartitions(map_partition_func)
# Afficher le résultat
print(rdd2.collect())
```

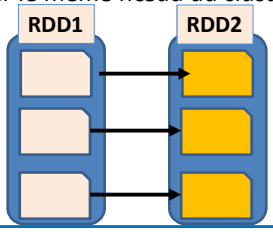
Résultat

```
[6, 15, 34]
```


Transformations Etroite

- Il existe deux principaux types de transformations sur les RDD: **Transformations étroites** (**Narrow Transformations**) et **Transformations larges** (**Wide Transformations**)
- Les **transformations étroites** n'entraînent pas de réorganisation des données entre les partitions.
  - Chaque partition d'entrée est utilisée pour **créer au plus une partition de sortie**
  - Ces transformations sont plus efficaces en termes de performance, car elles minimisent le déplacement des données sur le réseau: **chaque partition de sortie dépend uniquement des données contenues dans la même partition d'entrée**. Il n'est pas garanti que la partition en entrée et celle en sortie restent sur le même nœud du cluster.
  - Exemples de transformations étroites : **map**, **filter**, **…**

```
RDD2 = RDD1.filter(lambda ligne: len(ligne) > 10)
```



  - **N.B: Les partitions n'ont pas forcément la même taille.**

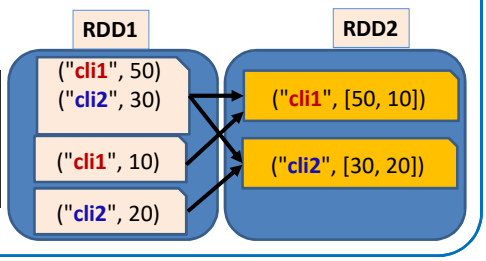
Transformations Etroite

- Quelques transformations **étroites**:
  - **map(*fonction*):** applique *fonction* à chaque élément du RDD pour former un nouveau RDD de sortie.
  - **filter(*fonction*):** sélectionne les éléments du RDD pour lesquels *fonction* retourne **true** et retourne un nouveau RDD de sortie.
  - **flatmap(*fonction*):** similaire à **map** mais à chaque élément on applique *fonction* qui retourne une collection d'éléments (int, string, …). Chaque collection sera ensuite aplatie pour avoir des éléments **simples** du nouveau RDD en sortie.
  - **union(*autre_RDD*):** retourne un nouveau RDD constitué des partions du RDD en entrée et *autre_RDD*.
  - …

Transformations Larges

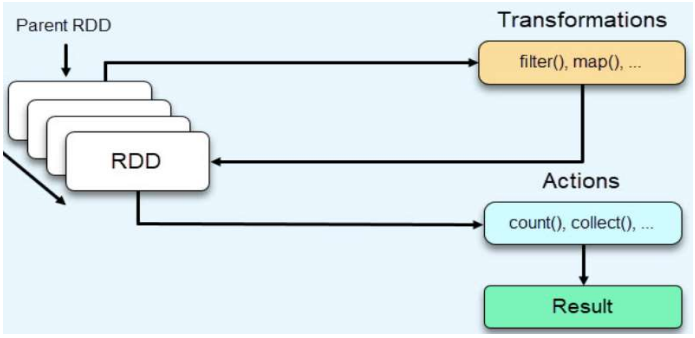
- Les **transformations larges** nécessitent une **réorganisation** des données entre les partitions.
- **Plusieurs partitions d'entrée** peuvent contribuer à la **création d'une ou plusieurs partitions de sortie**, ce qui peut entraîner des **échanges de données sur le réseau**.
- Les données des partitions des RDD en entrée sont combinées pour produire les partitions du RDD en sortie (*C'est similaire à **Reduce** dans **MapReduce** où **Shuffle** est réalisé*).
- Les transformations larges sont généralement plus **coûteuses en termes de performance** par rapport aux transformations étroites (*transfert de données entre les nœuds*)
- Exemples de transformations larges: **distinct**, **groupByKey**,**reduceByKey**, **join**, … etc

```
# Utiliser groupByKey pour regrouper les pairs
# de RDD1 par clé dans RDD2
RDD2 = RDD1.groupByKey()
```



RDD: Actions

- Une **action** Spark est une opération finale qui **déclenche le traitement** des **transformations précédentes** pour **produire une valeur** ou **écrire des données**.
- Le résultat d'une action n'est pas un **RDD**, mais généralement une valeur scalaire, une liste ou une écriture de données dans un stockage externe, tel qu'un fichier ou une base de données.
- **Les transformations ne seront exécutées que lorsqu'un action est lancée.**



https://phoenixnap.com/kb/resilient-distributed-datasets

RDD: Actions

- Les résultats des actions sont récupérés par le **Driver**.
- Voici quelques exemples d'actions courantes sur un RDD en Spark :
  - **collect()**: renvoie tous les éléments du RDD sous forme d'un tableau local.
  - **reduce(*fonction*)**:agrège les éléments du RDD en utilisant ***fonction*** qui a deux argument et qui retourne une valeur.
  - **count()**: renvoie le nombre d'éléments (entier) dans le RDD.
  - **first()**: renvoie le premier élément du RDD.
  - **take(*n*)**: renvoie les premiers **n** éléments du RDD sous forme de tableau local.
  - **foreach(*fonction*)**: appliquer une fonction donnée à chaque élément du RDD (*pas de nouveau RDD*)
  - **saveAsTextFile(*path*)**: enregistre le contenu du RDD dans un ou plusieurs fichiers texte d' un répertoire (*path*)
  - …

L'action count

- Elle permet de calculer le nombre d'éléments d'un RDD
- **Exemple** : Calculer le nombre de mots dans un fichier

```
# Importer SparkSession
from pyspark.sql import SparkSession
# Créer une session Spark
spark = SparkSession.builder.appName("Exemple Map").master("local[*]").getOrCreate()
# Charger un fichier  texte local en tant que RDD
rdd1 = spark.sparkContext.textFile("test.txt")
# Appliquer la transformation flatmap pour diviser les lignes en mots
rdd3 = rdd1.flatMap(lambda ligne: ligne.split(" "))
# calculer le nombre d'éléments de rdd3
nbre_mots = rdd3.count()
print("Nombre de mots du fichier:", nbre_mots)
# fin de la session
spark.stop()
```


L'action collect

- Cette action renvoie toutes les données du RDD vers le programme driver sous forme d'un **tableau**.
- Cela peut entraîner un **débordement de mémoire** sur le driver lorsqu'on a un grand ensemble de données.
- **Exemple** :

```
from pyspark.sql import SparkSession
# Créer un objet SparkContext
spark = SparkSession.builder.appName("Exemple collect")\
                            .master("local[*]").getOrCreate()
sc = spark.sparkContext
# Créer un RDD avec des paires clé-valeur
data = [("fruit", "pomme"), ("fruit", "orange"),\
        ("animal", "lion"), ("animal", "elephant")]
rdd1 = sc.parallelize(data)
res = rdd1.collect()
print(res)
print("Nombre de tiples:", len(res))
```

```
[('fruit', 'pomme'), ('fruit', 'orange'), ('animal', 'lion'), ('animal', 'elephant')]
Nombre de tiples: 4
```

L'action saveAsTextFile

❑ Elle permet de sauvegarder le contenu des partitions d'un RDD dans un dossier

```
# Données à traiter
data = [("id1", 50), ("id2", 30), ("id1", 10), ("id2", 20)]
# Crée un  RDD à partir du contenu de data
RDD1 = sc.parallelize(data)
# Utiliser groupByKey pour regrouper les pairs de RDD1 par clé
RDD2 = RDD1.groupByKey()
# Itérer sur les groupes et agir sur les valeurs
for key, values in RDD2.collect():
    print(f"(Client: {key}, Montants:{list(values)})")
```

```
(Client: id1, Montants:[50, 10])
(Client: id2, Montants:[30, 20])
```

Les partitions contiennent des tuples (*id du client*, *liste des montants*)

Somme des valeurs groupées dont a et b sont des éléments.

```
# Sauvegarder les partitions du RDD dans un dossier "./client"
RDD2.saveAsTextFile("./client")
```

```
RDD3 = RDD1.reduceByKey(lambda a, b: a+b)
```

```
# Calculer la somme des montants par client depuis RDD2
RDD3 = RDD2.map(lambda x: (x[0], sum(list(x[1]))))
print("RDD3:", RDD3.collect())
```

RDD3: [('id1', 60), ('id2', 50)]

Chaque élément de RDD2 est un tuple de clé (x[0]) et valeur (x[1]) convertie en une liste d'entiers pour appliquer SUM

L'action foreach*(fonction)*

❑ Elle permet d'appliquer une fonction sur chaque élément du RDD

❑ **Exemple** :

```
from pyspark import SparkContext
# Créer le contexte Spark
sc = SparkContext("local", "Exemple Spark")
# Créer un RDD avec une liste d'entiers
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
# Définir une fonction à appliquer à chaque élément
def ma_fonction(x):
    print(x * 2)
# Utiliser foreach pour appliquer la fonction à chaque élément
rdd.foreach(ma_fonction)
```

Résultat

```
2
4
6
8
10
```

L'action reduce*(fonction)*

- Elle agrège les éléments d'un RDD en utilisant une fonction **associative** et **commutative**.
- Elle combine les éléments de manière itérative pour produire un seul résultat.
- **Exemple** :

```
from pyspark import SparkContext
# Création d'un SparkContext
sc = SparkContext("local", "ReduceExample")
# Création d'un RDD avec une liste d'entiers
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
# Utilisation de l'action reduce pour additionner tous les éléments du RDD
result = rdd.reduce(lambda x, y: x + y)
# Affichage du résultat
print("Résultat de la réduction (somme) :", result)
```

Résultat

```
Résultat de la réduction (somme) : 15
```


L'action take*(n)*

- Elle agrège les éléments d'un RDD en utilisant une fonction **associative** et **commutative**.
- Elle retourne un tableau contenant les n premiers éléments du RDD en entrée.
- **Exemple** :

```
from pyspark.sql import SparkSession
# Créer un objet SparkContext
spark = SparkSession.builder.appName("Exemple take")\
                            .master("local[*]").getOrCreate()
sc = spark.sparkContext
# Créer un RDD avec des paires clé-valeur
data = [("fruit", "pomme"), ("fruit", "orange"),\
        ("animal", "lion"), ("animal", "elephant")]
rdd1 = sc.parallelize(data, 2)
# Regrouper les valeurs par clé
rdd2 = rdd1.sortBy(lambda x: x[1])
# Afficher 2 premiers éléments de rdd2
print(rdd2.take(2))
```

Résultat

```
[('animal', 'elephant'), ('animal', 'lion')]
```

Les variables broadcastées (1/3)

- Lorsqu'une tâche (fonction) est envoyée à un exécuteur, elle est accompagné par ses **paramètres** (**variables**) dont la taille peut être grande.
- Les variables broadcastées permettent de stocker sur chaque **Worker** des **données en lecture seule** au lieu des les envoyer à chaque fois avec une tâche à exécuter.
- Ces variables seront utilisées par les exécuteurs lors de l'exécution de plusieurs opérations.
- Les tâches d'une application sur le même worker sont indépendantes: chacune a ses propres variables dans la RAM.
- Si on a donc des tâches ayant les mêmes variables, elles seront dupliquées dans la RAM

  ➔ Surcharge du réseau et la RAM

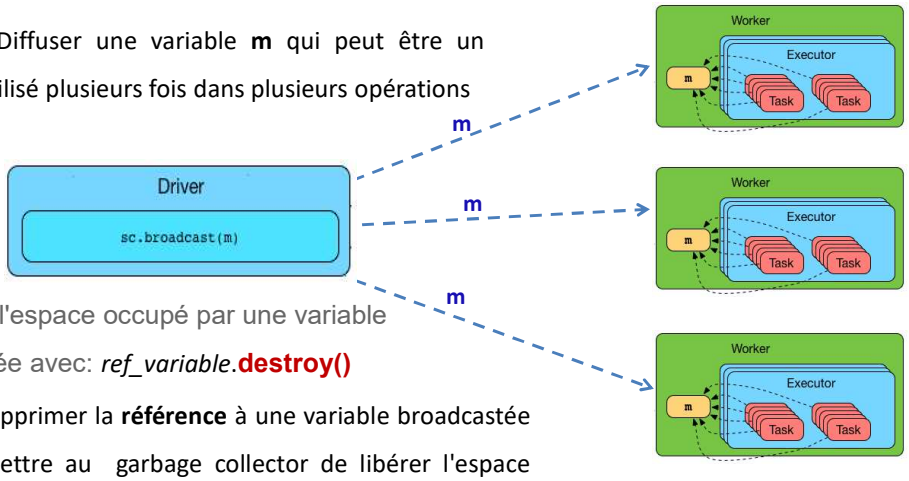
  ➔ Solution: les variables broadcastées

Les variables broadcastées (2/3)

- On peut créer une variable broadcastée avec la méthode broadcast de l'objet SparkContext:

  ref_variable = **sc.broadcast(***variable***)**

- **Exemple**: Diffuser une variable **m** qui peut être un dataset utilisé plusieurs fois dans plusieurs opérations



- On libère l'espace occupé par une variable broadcastée avec: *ref_variable*.**destroy()**
- On peut supprimer la **référence** à une variable broadcastée pour permettre au garbage collector de libérer l'espace mémoire associé sur les workers : **del** *ref_variable*

Les variables broadcastées (3/3)

❑ **Exemple**:

```
from pyspark.sql import SparkSession
# Initialisation de Spark
spark = SparkSession.builder.appName("BroadcastExample").getOrCreate()
sc = spark.sparkContext
# Création d'une variable à diffuser ensemble de paires clé:valeur
data_broadcast = {'Ma': 'Maroc', 'It': 'Italie', 'Es': 'Espagne'}
broad_var = sc.broadcast(data_broadcast)
# Création d'une RDD à traiter
data = [('Ma', 'Casa', 'Economie'), ('It', 'Rome', 'Tourisme'),\
        ('Es', 'Madrid', 'Economie'), ('Ma', 'Rabat', 'Administration')]
rdd = sc.parallelize(data)
# Utilisation de la variable broadcast dans la transformation map
result = rdd.map(lambda x: (x[1], broad_var.value.get(x[0], 'Not Found'), x[2]))
# Affichage des résultats
result.foreach(print)
```

N.B:
La propriété **value** permet d'avoir la valeur d'une variable broadcastée

Résultat

```
('Rome', 'Italie', 'Tourisme')
('Madrid', 'Espagne', 'Economie')
('Rabat', 'Maroc', 'Administration')
('Casa', 'Maroc', 'Economie')
```


Les accumulateurs (1/2)

❑ Un **accumulateur** est une **variable partagée** utilisée pour effectuer des opérations d'agrégation parallèles de données.

❑ Il est utilisé pour implémenter des opérations de type réduction calculant une **somme** ou effectuant un **comptage** de manière efficace et **distribuée**.

❑ On crée un accumulateur avec la méthode **accumulator** de l'objet **SparkContext**:

ref_accumulateur = **sc.accumulator(***val_initiale***)**

❑ Une fonction qui modifiera l'accumulateur peut être appelée dans une **action** comme **foreach** ou une **transformation** comme **map**, **filter** ou **flatmap**

N.B: Si cette fonction est appelée dans une **transformation**, elle ne sera exécutée que lorsqu'il y aura une action qui déclenchera cette transformation.

Les accumulateurs (2/2)

- **Exemple**

```
# Création d'une RDD avec des tuples
data = [("animal", "tigre"), ("fruit", "pomme"),\
        ("animal", "lion"), ("animal", "elephant")]
rdd1 = spark.sparkContext.parallelize(data)
# Initialisation de l'accumulateur avec une valeur initiale de 0
nbre_animal = spark.sparkContext.accumulator(0)
# Fonction pour ajouter chaque élément de la RDD à l'accumulateur
def add_to_nbre_animal(elt):
    global nbre_animal
    if elt[0] == "animal":
        nbre_animal.add(1)
# Application de la fonction à chaque élément de la RDD
rdd1.foreach(add_to_nbre_animal)
# Affichage de la somme calculée à partir de l'accumulateur
print("Nombre d'animaux:", nbre_animal.value)
```

Résultat

```
Nombre d'animaux: 3
```


RDD: Caractéristiques

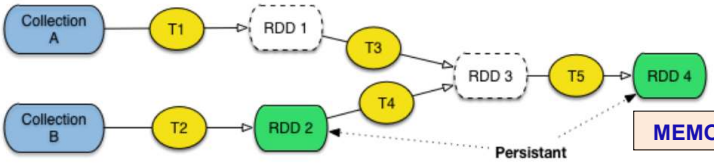
- **Résilient** (**Resilient**): Les RDD peuvent **tolérer les pannes**, en cas de défaillance d'un nœud du cluster, les **données (*partitions*) perdues** sont **automatiquement reconstruites** à partir du **lignage** qui enregistre les transformations appliquées sur le RDD d'origine.
- **Distribué** (**Distributed**):Les données d'un RDD sont distribuées sur plusieurs nœuds d'un cluster Spark. Chaque nœud stocke une partie des données, ce qui permet un **traitement parallèle**.
- **Dataset**: Un RDD est essentiellement une **collection d'éléments de données**. Ces éléments peuvent être des objets Python, des valeurs numériques, des lignes de texte, etc. Un RDD peut contenir des **données structurées ou non structurées**.

RDD: Caractéristiques

- **Partitionné** (**Partitioned**): Les données d'un RDD sont divisées en partitions, qui sont des unités de base de traitement parallèle. **Chaque partition est traitée indépendamment** sur un nœud du cluster. Le nombre de partitions dépend généralement du nombre de cœurs de calcul disponibles dans le cluster.
- **Calcul en mémoire** (**In memory**): les RDD sont partitionnés et calculés en mémoire.
- **Immuabilité** ou **Immutabilité** (**Immutability**)**:** Les RDD sont immuables, càd qu'on ne peut pas modifier un RDD existant. Au lieu de cela, on crée de nouveaux RDD à partir de transformations appliquées à des RDD existants. Cela garantit la stabilité des données.

RDD: Caractéristiques

- **Evaluation paresseuse** (**Laziness**): Les transformations sur un RDD sont évaluées de manière paresseuse, càd qu'elles ne **sont effectivement exécutées que lorsqu'on déclenche une action**. Les transformations sont planifiées et optimisées avant l'exécution pour maximiser l'efficacité.
- **Persistance** (**Persistence**) :On peut mettre en cache mémoire un RDD, avec **persist()** ou **cache()**, pour accélérer l'accès ultérieur à ses éléments. Cela est particulièrement utile pour les données fréquemment utilisées dans des **itérations** ou des actions répétées.




```
RDD2 = sc.parallalize(B)
RDD2.cache()
```

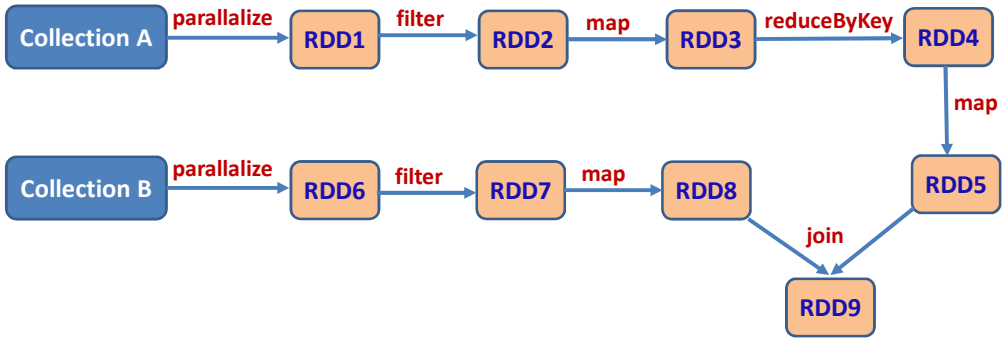
MEMORY_AND_DISK, DISK_ONLY, ...

```
RDD2.persist(pyspark.StorageLevel.MEMORY_ONLY)
```

- **Opération à grain grossier**(**Coarse-Grained Operation**): les transformations doivent être appliquées à l'**ensemble du dataset du RDD**, mais pas aux éléments individuels.

RDD: DAG (Directed Acyclic Graph)

- Le **lignage** (**lineage**) représente le **plan logique** d'exécution des différentes transformations permettant d'obtenir les RDD ➔ Dépendances entre les différentes opérations Spark.
- **Exemple de plan logique:**



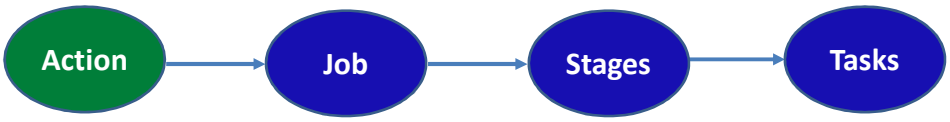
- À l'appel d'une **action**, le **plan logique** est passé au **Catalyst Optimizer** qui optimise les transformations appliquées (par ***réarrangement***, ***fusion*** , …).

RDD: DAG (Directed Acyclic Graph)

- Le **plan logique optimisé** est soumis ensuite au **DAG Scheduler** **(planificateur)** qui le convertit en un **plan physique d'exécution** sous forme d'un **Graphe Acyclique Dirigé** (**DAG**)

- Le **DAG** représente le **Job** à exécuter.
- Un **Job** représente la séquence des **transformations** **déclenchées** par une **action** (*collect(), saveAsTextFile(), …*) pour avoir un **résultat**.
- Un **job** est composé de plusieurs **stages** (*niveaux*) dont chacun contient une ou plusieurs **tâches**
- Le **DAG** est utilisé aussi pour assurer la **tolérance aux pannes** pour récupérer les partitions en cas d'échec d'un Worker.

RDD: DAG (Directed Acyclic Graph)



- **Stage** : c'est une suite de transformations qu'on peut exécuter en **parallèle** sur plusieurs partitions RDD.
- Un **stage** est une **unité de parallélisme** dans l'exécution d'un **job**.
- La constitution des **stages** est **basée sur les deux types de transformations**:
  - **Transformations étroites**: transformation d'une partition sans mélange avec d'autres partitions (sans **shuffle**)
  - **Transformations larges**: transformation nécessitant un mélange des données entre les partitions (avec **shuffle**).
- **Tâche** (**Task**): c'est l'unité fondamentale d'exécution.

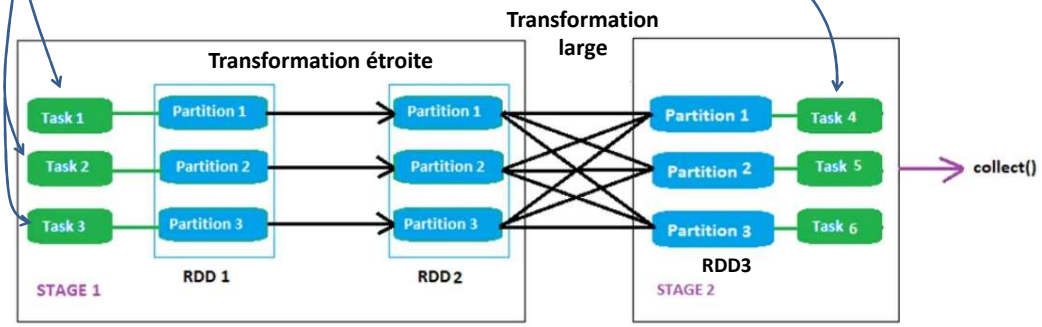
➔ **Chaque partition est traitée par une tâche**.

RDD: DAG (Directed Acyclic Graph)

RDD2 = RDD1.**filter**(.....)

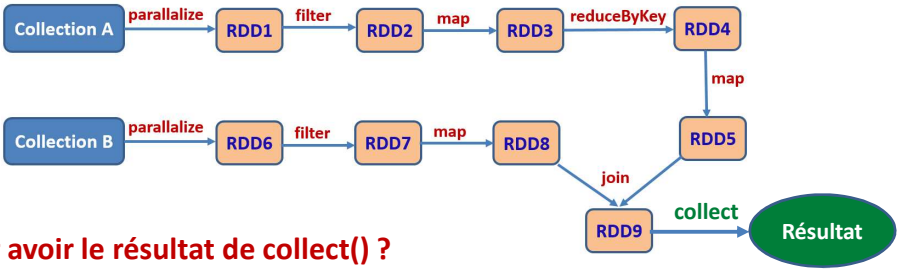
RDD3 = RDD2.**reduceByKey**(.....)

Output = RDD3.**collect**()

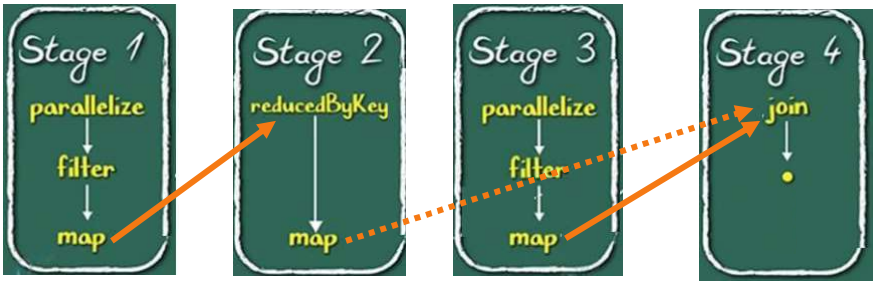


RDD: DAG (Directed Acyclic Graph)

❑ **Exemple:**



Le Job pour avoir le résultat de collect() ?



N.B: On peut fusionner les deux stages 1 et 3 (ou et 3) en un seul.

RDD: Quelques Limites

Par rapport à d'autres structures de données Spark, les RDD ont des limites comme:

❑ **Manque de typage fort**: Les RDD sont moins axés sur le typage fort que les **DataFrames**. Les erreurs liées au type de données ne sont souvent détectées qu'à l'exécution plutôt qu'à la compilation.

N.B: On peut appliquer un **map** sur un RDD pour faire un **cast** des types de ses données.

❑ **Moins d'optimisations intégrées**: Les RDD offrent moins d'optimisations intégrées du plan logique d'exécution ➔ moins de performance.

❑ **Moins adaptés aux traitements SQL**: Si on a besoin d'opérations de type **SQL**, les **DataFrames** ou les **Datasets** peuvent être plus adaptés, car ils sont conçus pour fonctionner de **manière transparente** avec **Spark SQL** et ont un **schéma tabulaire**.

❑ **Complexité du code**: Les RDD nécessitent souvent plus de code qui peut être exprimée de manière plus concise avec les **DataFrames** et **Datasets** à l'aide de l'API fonctionnelle et du **langage SQL** intégré.