Spark : utilisation du RDD

Note: ref = `https://sparkbyexamples.com/pyspark-rdd` ref2 = `https://github.com/spark-examples/pyspark-examples`

Utilisation du RDD

Comme vu précédemment, le RDD peut se creer à partir :

- d'une methode parallelize() de la classe SparkContext
- de la lecture d'un fichier

Creation d'un RDD

Un Resilient Distributed Dataset (RDD) permet à spark de lire et charger des données itérables.

la creation d'un RDD se fait de plusieurs manières :

- creation vide sans données
- à partir de données brutes
- à partir d'un fichier texte (distribué ou non)
- à partir d'un dossier (distribué ou non)

Creation d'un RDD vide

```
mon_rdd = spark.sparkContext.emptyRDD()
print(mon_rdd)
```

Creation d'un RDD à partir de données brutes

Utilisation de la methode **parallelize()** de la classe SparkContext

```
# données brutes
data = [1,2,3,4,5]
# creation d'un RDD par parralelisation
```

```
mon_rdd = spark.sparkContext.parallelize(data, 2)
print(mon_rdd)
```

Creation d'un RDD à partir d'un fichier texte

La creation peut se faire à partir d'un fichier distribué ou non.

```
# fichier non distribué
mon_rdd = spark.sparkContext.textFile("./DataSource/test.txt")
print(mon_rdd)
# fichier distribué
mon_rdd = spark.sparkContext.textFile("hdfs://mon_hdfs/mon_fichier.txt")
print(mon_rdd)
```

Creation d'un RDD à partir d'un dossier

```
mon_rdd = spark.sparkContext.wholeTextFiles("./DataSource/")
print(mon_rdd)
```

Creation d'un RDD à partir d'un fichier csv ou json

```
# fichier csv
read_csv = spark.sparkContext.textFile("./DataSource/zipcodes.csv")
print(read_csv.__class__)
mon_rdd = read_csv.map(lambda line: line.split(","))
print(mon_rdd)
# fichier json
mon_rdd = spark.sparkContext.textFile("./DataSource/zipcodes.json")
print(mon_rdd)
```

Opérations sur un RDD

Les opérations sur un RDD se divisent en 2 catégories :

- les **transformations** : ce sont des opérations **lazy** qui sont planifiées les unes après les autres. Les données ne sont pas chargées ni calculées.
- les **actions** : ce sont des opérations **eager** qui sont exécutées immédiatement. Les données sont chargées et calculées.

Opérations sur un RDD

Les transformations

> Les transformations sont des opérations qui permettent de créer un nouveau RDD à partir d'un RDD existant.

- Les transformations sont **lazy** et ne sont pas exécutées immédiatement.
- L'execusion des transformations est planifiée et exécutée par spark qu'au moment de l'execution d'une action.

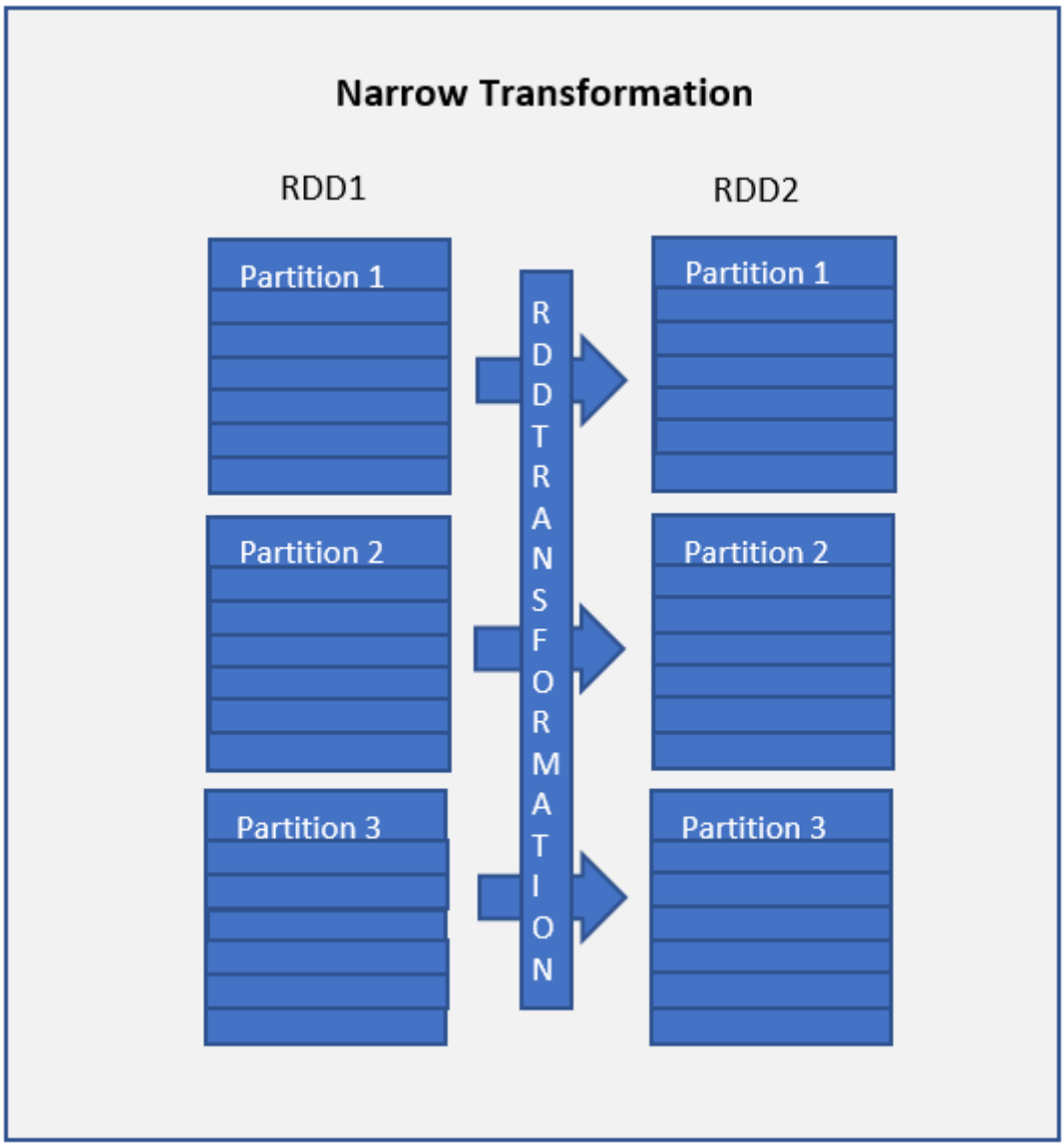
Les transformations

Les transformations sont classées en deux catégories :

- **Narrow transformations**
- **Wide transformations**

Les Narrow transformations

Les Narrow transformations ne nécessitent pas de communication entre les partitions.



Les Narrow transformations

map(func) : Applique une fonction à chaque élément d'un RDD et retourne un nouvel RDD où chaque élément est le résultat de l'application de la fonction.

```
rdd = sc.parallelize([1, 2, 3, 4])
squared_rdd = rdd.map(lambda x: x*x)
print(squared_rdd.collect()) # [1, 4, 9, 16]
```

Les Narrow transformations

filter(func) : Applique une fonction à chaque élément d'un RDD et retourne un nouvel RDD contenant uniquement les éléments pour lesquels la fonction retourne True.

```
rdd = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
filtered_rdd = rdd.filter(lambda x: x % 2 == 0)
```

```
print(filtered_rdd.collect()) # [2, 4, 6, 8, 10]
```

Les Narrow transformations

flatMap(func) : Applique une fonction à chaque élément d'un RDD et retourne un nouvel RDD où les éléments sont les résultats de l'application de la fonction (qui doit retourner un itérable).

```
rdd = sc.parallelize(["Hello World", "Goodbye World"])
words_rdd = rdd.flatMap(lambda x: x.split(" "))
print(words_rdd.collect()) # ["Hello", "World", "Goodbye", "World"]
```

Les Narrow transformations

distinct() : Retourne un nouvel RDD contenant uniquement les éléments uniques d'un RDD

```
rdd = sc.parallelize([1, 2, 3, 1, 2, 3, 4, 5, 4, 5])
distinct_rdd = rdd.distinct()
print(distinct_rdd.collect()) # [1, 2, 3, 4, 5]
```

Les Narrow transformations

sample(withReplacement, fraction, seed) : Retourne un échantillon aléatoire d'un RDD.

```
rdd = sc.parallelize(range(0, 10))
sampled_rdd = rdd.sample(False, 0.5, 42)
print(sampled_rdd.collect()) # [0, 1, 2, 3, 4, 6, 8, 9]
```

Les Narrow transformations

union(other) : Retourne un nouvel RDD contenant les éléments de deux RDDs. (equivalent du full-join en SQL)

```
rdd1 = sc.parallelize([1, 2, 3, 4])
rdd2 = sc.parallelize([3, 4, 5, 6])
union_rdd = rdd1.union(rdd2)
print(union_rdd.collect()) # [1, 2, 3, 3, 4, 4, 5, 6]
```

Les Narrow transformations

intersection(other) : Retourne un nouvel RDD contenant les éléments qui apparaissent dans les deux RDDs. (equivalent du inner-join en SQL)

```
rdd1 = sc.parallelize([1, 2, 3, 4])
rdd2 = sc.parallelize([3, 4, 5, 6])
intersection_rdd = rdd1.intersection(rdd2)
print(intersection_rdd.collect()) # [3, 4]
```

Les Narrow transformations

subtract(other, numPartitions=None) : Retourne un nouvel RDD contenant les éléments de l'un des RDDs qui ne sont pas présents dans l'autre RDD.

```
rdd1 = sc.parallelize([1, 2, 3, 4])
rdd2 = sc.parallelize([3, 4, 5, 6])
subtract_rdd = rdd1.subtract(rdd2)
print(subtract_rdd.collect()) # [1, 2]
```

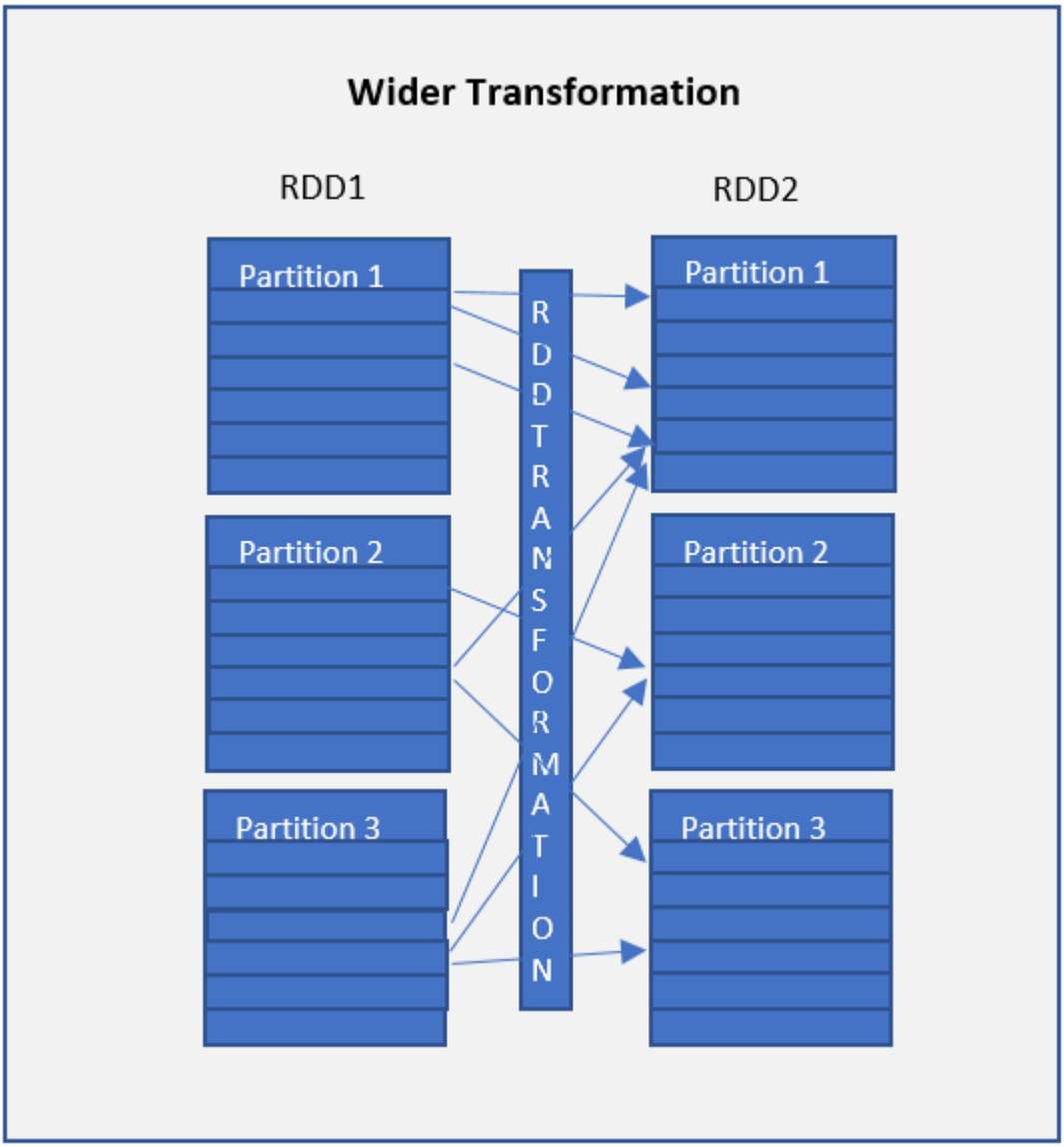
Les Narrow transformations

cartesian(other) : Retourne un nouvel RDD contenant toutes les combinaisons possibles entre les éléments des deux RDDs. (equivalent du union en SQL)

```
rdd1 = sc.parallelize([1, 2])
rdd2 = sc.parallelize(["a", "b"])
cartesian_rdd = rdd1.cartesian(rdd2)
print(cartesian_rdd.collect()) # [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

Les Wide transformations

Les Wide transformations nécessitent de la communication entre les partitions.



Les Wide transformations

groupByKey() : Groupe les éléments d'un RDD par clé et retourne un nouvel RDD de type PairRDD où les clés sont les clés originales et les valeurs sont des itérables contenant tous les éléments ayant cette clé.

```
rdd = sc.parallelize([(1, "a"), (1, "b"), (2, "c"), (2, "d"), (3, "e")])
grouped_rdd = rdd.groupByKey()
print(grouped_rdd.collect())
# [(1, <pyspark.resultiterable.ResultIterable object at 0x7f5f5d5b5c90>),
#  (2, <pyspark.resultiterable.ResultIterable object at 0x7f5f5d5b5f90>),
#  (3, <pyspark.resultiterable.ResultIterable object at 0x7f5f5d5b5e10>)]
```

Les Wide transformations

reduceByKey(func) : Applique une fonction de réduction à des éléments ayant la même clé d'un RDD de type PairRDD et retourne un nouvel RDD où les éléments ont été agrégés par clé.

```
rdd = sc.parallelize([(1, 2), (1, 3), (1, 4), (2, 3), (2, 5), (2, 6)])
reduced_rdd = rdd.reduceByKey(lambda x, y: x + y)
print(reduced_rdd.collect()) # [(1, 9), (2, 14)]
```

Les Wide transformations

join(other, numPartitions=None) : Effectue un join entre deux RDDs de type PairRDD sur les clés et retourne un nouvel RDD où les éléments ont été joint sur la clé.

```
rdd1 = sc.parallelize([(1, "a"), (2, "b"), (3, "c")])
rdd2 = sc.parallelize([(1, 1), (2, 2), (3, 3)])
joined_rdd = rdd1.join(rdd2)
print(joined_rdd.collect()) # [(1, ('a', 1)), (2, ('b', 2)), (3, ('c', 3))]
```

Les Wide transformations

cogroup(other, numPartitions=None) : Groupe les éléments des deux RDDs de type PairRDD ayant la même clé et retourne un nouvel RDD où les valeurs sont des itérables contenant les valeurs des deux RDDs pour chaque clé.

```
rdd1 = sc.parallelize([(1, "a"), (2, "b"), (3, "c")])
rdd2 = sc.parallelize([(1, 1), (2, 2), (3, 3)])
cogrouped_rdd = rdd1.cogroup(rdd2)
print(cogrouped_rdd.collect())
# [(1,
#   (iter(['a']), iter([1]))),
#  (2,
#   (iter(['b']), iter([2]))),
#  (3,
#   (iter(['c']), iter([3])))]
```

Les Wide transformations

groupWith(other, *others) : Groupe les éléments de plusieurs PairRDDs ayant la même clé et retourne un nouvel RDD avec des itérables contenant les valeurs des RDDs originaux pour chaque clé.

```
rdd1 = sc.parallelize([(1, "a"), (2, "b"), (3, "c")])
rdd2 = sc.parallelize([(1, 1), (2, 2), (3, 3)])
rdd3 = sc.parallelize([(1, "x"), (2, "y"), (3, "z")])
grouped_rdd = rdd1.groupWith(rdd2, rdd3)
print(grouped_rdd.collect())
    # [(1, (iter(['a']), iter([1]), iter(['x']) )),
```

```
    # (2, (iter(['b']), iter([2]), iter(['y']) )),
    # (3, (iter(['c']), iter([3]), iter(['z']) )) ]
```

Les Actions

> C'est une operation qui permet de d'executer le calcul plannifié par les transformations.

Ce n'est qu'a cette etape que les données sont utilisé et que le traitement est effectué.

Les actions sont des opérations qui ne retoune pas un RDD mais un **résultat**.

Les Actions

collect() : Cette méthode permet de récupérer tous les éléments d'un RDD sur le noeud driver.

```
# Action de collect()
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
result = rdd.collect()
print(result) # [1, 2, 3, 4, 5]
```

Les Actions

count() : Cette méthode permet de compter le nombre d'éléments dans un RDD.

```
# Action de count()
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
result = rdd.count()
print(result) # 5
```

Les Actions

countByKey() : Cette méthode permet de compter le nombre d'occurrences de chaque clé dans un RDD à clé-valeur.

```
# Action de countByKey()
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
result = rdd.countByKey()
print(result) # {"a": 2, "b": 1}
```

Les Actions

lookup(key) : Cette méthode permet de récupérer toutes les valeurs associées à une clé donnée dans un RDD à clé-valeur

```
# Action de lookup()
rdd = spark.sparkContext.parallelize([("a", 1), ("b", 2), ("a", 3)])
result = rdd.lookup("a")
print(result) # [1, 3]
```

Les Actions

first() : Cette méthode permet de récupérer le premier élément d'un RDD.

```
# Action de first()
rdd = sc.parallelize([1, 2, 3, 4, 5])
result = rdd.first()
print(result) # 1
```

Les Actions

take(n) : Cette méthode permet de récupérer les n premiers éléments d'un RDD.

```
# Action de take(n)
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
result = rdd.take(3)
print(result) # [1, 2, 3]
```

Les Actions

takeOrdered(n, key=None) : Cette méthode permet de récupérer les n premiers éléments d'un RDD triés par ordre croissant.

```
# Action de takeOrdered(n)
rdd = spark.sparkContext.parallelize([5, 4, 3, 2, 1])
result = rdd.takeOrdered(3)
print(result) # [1, 2, 3]
```

Les Actions

min()/max() : Cette méthode permet de récupérer le minimum/maximum d'un RDD.

```
# Action de min()
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
result_min = rdd.min()
result_max = rdd.max()
print(result_min, result_max) # 1, 5
```

Les Actions

reduce(func) : Cette méthode permet de réduire les éléments d'un RDD en utilisant une fonction de réduction donnée.

```
# Action de reduce()
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
result = rdd.reduce(lambda x, y: x + y)
print(result) # 15
```

Les Actions

fold(nb, func) : Cette methode est similaire à la méthode reduce() mais elle permet de spécifier une valeur initiale pour la reduction et prend en compte le nombre de partitions.

```
# Action de fold(zeoValue, lambda param: operation)
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5],2)
result = rdd.fold(10, lambda x, y: x + y)
# 10 + 1 + 2 + 3 + 4+ 5 => 25
print("initial partition count:"+str(rdd.getNumPartitions()))
# ajoute zeroValue * nb partition au resultat
# 25 + 2*10 => 45
print(result) # 45
```

Note: Kamoulox pour celle ci, je comprends pas ce qu'elle represente.

Les Actions

foreach(func) : Cette méthode permet d'appliquer une fonction donnée à chaque élément d'un RDD.

```
# Action de foreach()
rdd = spark.sparkContext..parallelize([1, 2, 3, 4, 5])
rdd.foreach(lambda x: print(x*2))
# 2
# 4
# 6
```

```
# 8
# 10
```

Les Actions

saveAsTextFile(path) : Cette méthode permet de sauvegarder les éléments d'un RDD dans un fichier texte.

saveAsSequenceFile(path) : Cette méthode permet de sauvegarder les éléments d'un RDD dans un fichier de séquence.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.saveAsTextFile("output.txt")
rdd = sc.parallelize([(1, "a"), (2, "b"), (3, "c")])
rdd.saveAsSequenceFile("output.seq")
```

UtilisationDataframe