PySpark

PySpark

- PySpar shell exposes the Spark programming model to Python.
- PySpark shell links the Python API to the Spark Core and initializes the SparkContext.
- PySpark require Python to be available on the system PATH and uses it to run programs.

Submitting a PySpark Job

- Many times, our code depend on other project or source codes. So, we need to package them alongside our application to distribute the code to a Spark cluster.
    - We create an assembly .jar file containing our code and its dependencies.
    - SBT and Maven have assembly plugins to help us out on this task.
    - When creating assembly jars, list spark and Hadoop as provided dependencies.
    - Once we have an assemble .jar file, we can call the bin/spark-submit script while passing our .jar.
    - The spark-submit script is used to launch applications on the cluster.
- Spark is preconfigured for YARN, and it does not require any additional configuration to run
- Yarn control resource management, scheduling and security when we run spark applications on it.
- It's possible to run an application in any mode, whether it's cluster mode or client mode (See Fig 1.)

```
$ ./bin/spark-submit \
 --class org.apache.spark.examples.SparkPi \
 --master yarn \
 --deploy-mode cluster \
 --driver-memory 4g \
 --executor-memory 2g \
 --executor-cores 1 \
 --queue thequeue \
 examples/jars/spark-examples*.jar \
 10
```

Figure 1. Launching a spark application in the cluster/client mode.

Spark-submit -master yarn -deploy client PySparkJob.py to submit a park job using the YARN cluster.

The first PySpark Job

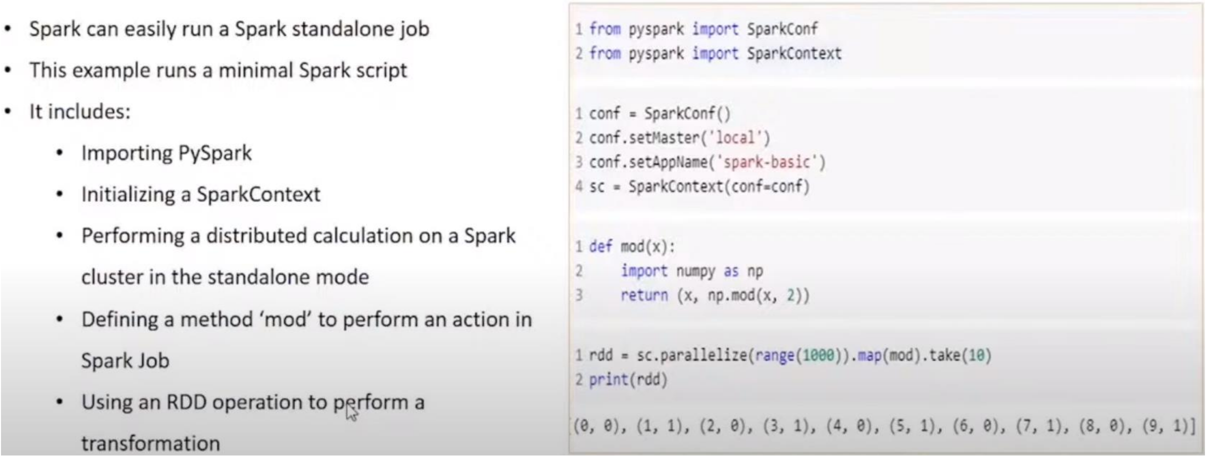


Figure 2. The first PySpark Job

What is Spark RDDs

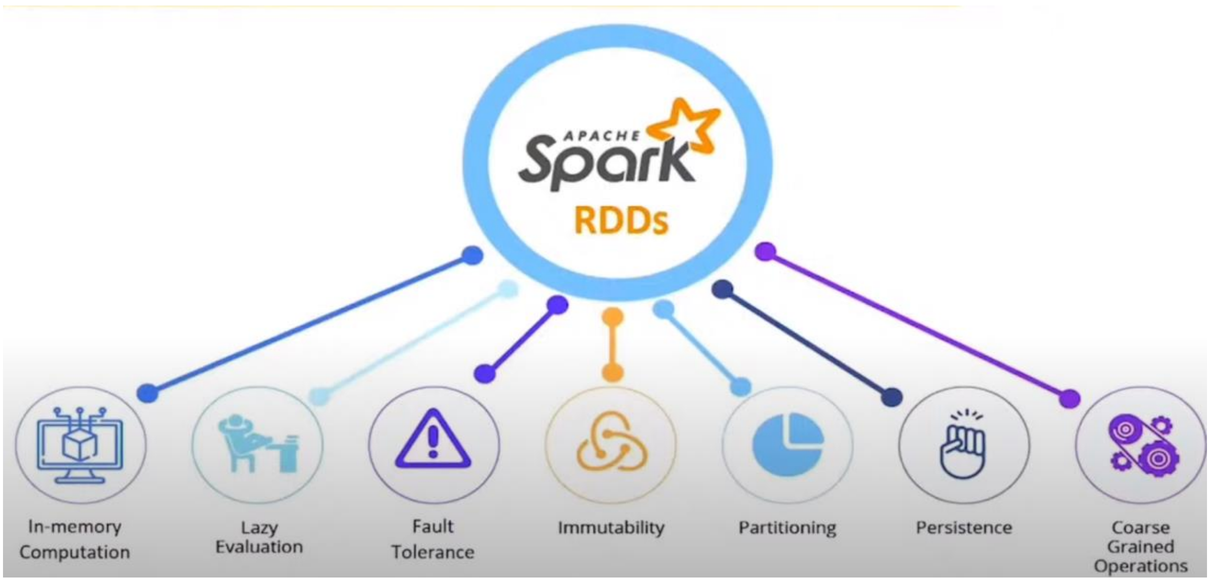


Figure 3. What are RDDs

- Main logical data unit in Spark.
- Distribute collection of objects.
  - Stored in memory or on disks.
  - Stored on different machines of a cluster.
- Can be divided into multiple logical partitions.
  - Those partitions can be stored and processed on different machines of a cluster.
- Can be cached and used again for future transformations.
- Lazily evaluated.
- It performs transformations and actions on RDDs.

Why do we need RDD?

Technologies prior to RDD.

- In-Disk computation.
    - Which is slower than in-memory computation.
    - It uses memory capacity to process stored data.
- Job execution.
    - The existing distributed systems (MapReduce) need to store data in some intermediate stable distributed store, namely, HDFS.
    - Operations get slowed, as this involves loads of I/O, replication and serialization processes.
- Parallel Processing.
    - Besides of Spark, most of the Data Processing Framework are less efficient to perform parallel processing.

How RDD solve the problem

- RDD is a is distributed collection of JVM objects and functional operators.
- Multiple logical partitions to handle huge volume of data parallelly.
- Just executed when they are needed (Lazy Evaluation).
- Enhanced Distributed Computing.
    - Processing data over multiple jobs.
- Partitioning.
    - RDDs are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- Location stickiness.
    - RDDs can define their placement preference (their location) to compute partitions.

Features of Spark RDDs

- In-memory computation.
    - The data in a RDD can be stored in memory for as long as possible.
- Immutable or Read-only
    - RDD can’t be changed once created and can only be transformed using transformations.
- Lazy Evaluation
    - Data in RDD isn’t available or transformed until is required.
- Cacheable.
    - Cache stores the intermediate RDD results in memory only.
    - The default storage of RDD cache is in memory.
- Parallel Data Processing.
- Typed.
    - RDDs Records have types (Int, String, Float, etc)

Ways of Creating RDDs in PySpark

```python
from pyspark import SparkConf
from pyspark import SparkContext
```

Creating a Spark Context:

```python
conf = SparkConf().setAppName('SparkRDD').setMaster('local')
sc = SparkContext(conf=conf)
```

- The **conf** object is the configuration for a Spark application
- We define the App Name and the Master URL in it
- **sc** is an object of SparkContext

Figure 4. Ways of creating RDDs in PySpark

Creating an RDD Using a List:

```python
values = [1, 2, 3, 4, 5]
rdd = sc.parallelize(values)
```

Printing the RDD Values:

```python
rdd.take(5)
```

```
[1, 2, 3, 4, 5]
```

Figure 5. Creating a RDD using a List.

Uploading a File to Google Colab:

```
from google.colab import files
uploaded = files.upload()
```

Initializing an RDD Using a Text File:

```
rdd = sc.textFile("Spark.txt")
```

Printing the Text from the RDD:

```
rdd.take(1)
```

```
['Apache Spark with Python is PySpark']
```

Figure 6.Uplading a file to Google colab.

RDD Persistence and Caching

- As RDDs are lazy Evaluated, multiple use of the same RDD, make recomputing the RDD every time.
    - To avoid recomputing, ask Spark to persist the data -> Node which computes RDD stores its partitions.
    - If the persisting process fails, Spark will recompute the lost partitions of the data whenever they're required.
- `Unpersist()` to remove partitions from the cache.

```
aba = sc.parallelize(range(1,10000,2))
aba.persist()
```

```
aba = sc.parallelize(range(1,10000,2))
aba.persist()
```

```
PythonRDD[15] at RDD at PythonRDD.scala:53
```

Figure 7. RDD Persistence and caching.

Persistence Level

Level	Space Used	CPU Time	In Memory	On Disk	Comments
MEMORY_ONLY	High	Low	Yes	No	Stores an RDD as a deserialized Java object in JVM. If it does not fit in memory, some partitions will not be cached and will be recomputed when needed
MEMORY_ONLY_SER	Low	High	Yes	No	Stores an RDD as a serialized Java object. It stores one-byte array per partition
MEMORY_AND_DISK	High	Medium	Some	Some	Stores an RDD as a deserialized Java object in JVM. If the full RDD does not fit in memory, the remaining partition is stored on the disk, instead of recomputing it
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to the disk if there is too much data to fit in memory and stores serialized representation in memory
DISK_ONLY	Low	High	No	Yes	Stores the RDD partitions only on disk

Figure 8. Persistence levels

RDD Persistence

- Caching.
    - It's very useful when data is accessed repeatedly, such as, querying a small "hot"dataset or running an iterative algorithm.
    - Cache if fault-tolerant.

Operations on RDDs

- There are two kind of operations we can perform on RDDs: *Transformations* and *Actions*.
    - *Transformations* will return a new RDD as RDDs are generally immutable.
    - *Actions* will return a value.

RDD Transformations

- Lazy operations on RDDs that create one or more new RDDs
- Return a pointer to the new RDD and allow us to create dependencies between RDDs.
- Each RDD is a dependency chain and every single of them has a function for calculating its data and a pointer (dependency) to its parent RDD.
- Nothing will be executed unless we call some transformation or action that will trigger the job creation and execution (lazy evaluation).
- It's a step in a program (It could be the only one) rather than a set of data.
- See Figure 9.

Figure 9. Data transformation.

List of transformations

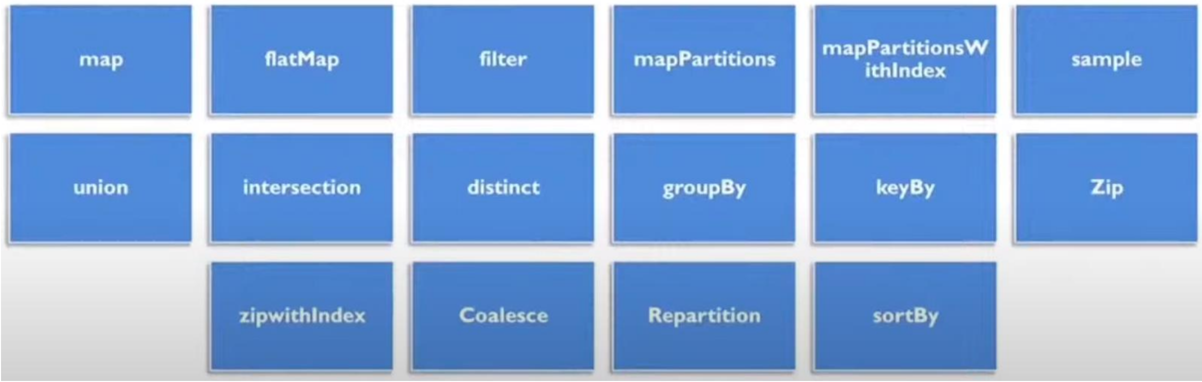


Figure 10. List of transformations.

Map

- Passes each element through a function.
- It returns an output for every input

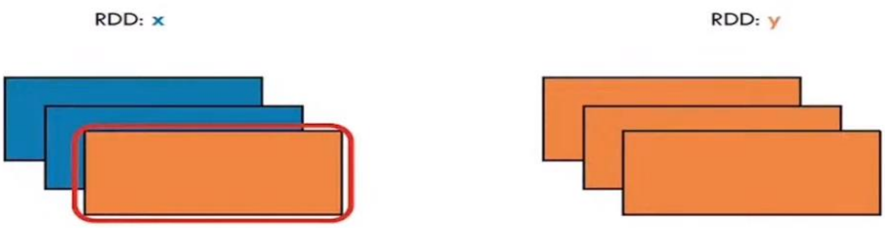


Figure 11.Effect of maps over a list of elements .

FlatMap

- It's a map which every input can be mapped to 0 or more output items
- It returns a sequence (or not) rather than just one single item, for every item.

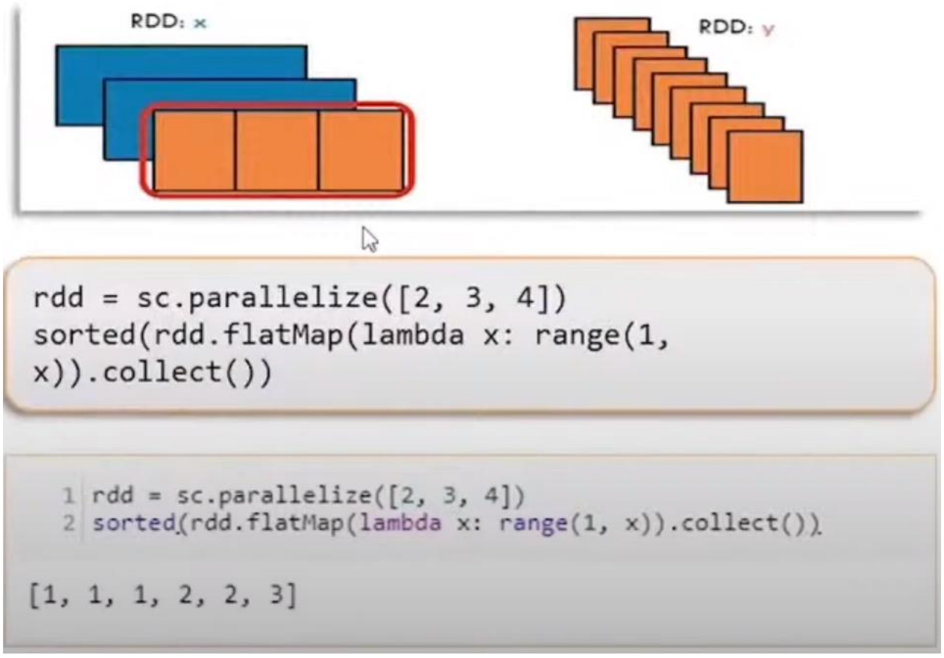


Figure 12.FlatMap

Filter

- It returns a collection of elements on the basis of the condition provided in the function.

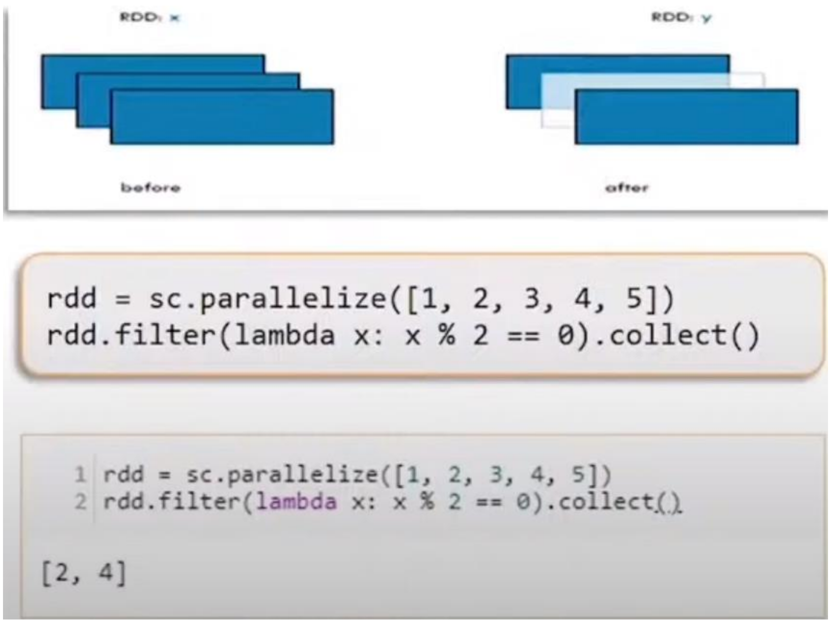


Figure 13. Filter over a list of numbers to filer even numbers.

Sample

- Sample a fraction of the data. With or without replacement, using a given random number generator seed.
- Parameters can be added.
    - *withReplacement*: Elements can be sampled multiples times (replaced when sampled out).
    - Fraction: Make the size of the sample as a fraction of the RDDs size without replacement.
    - Seed: A number as a see for the random number generator.

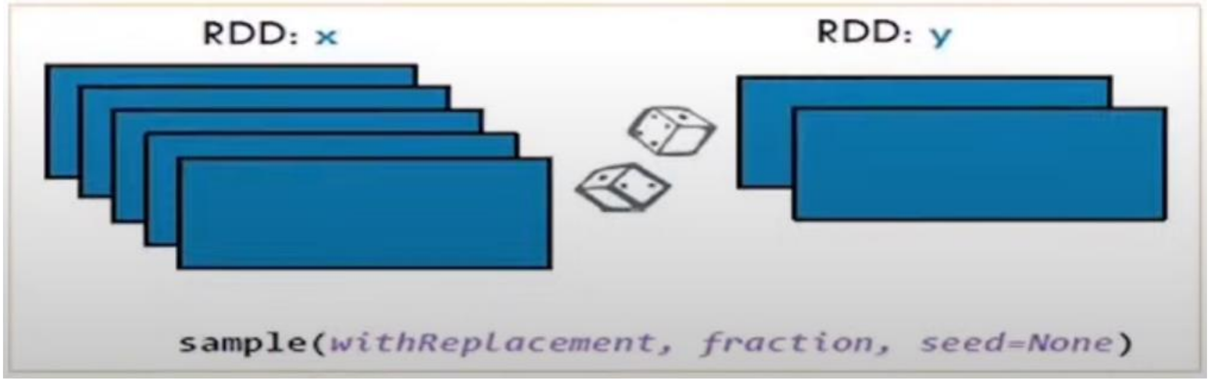
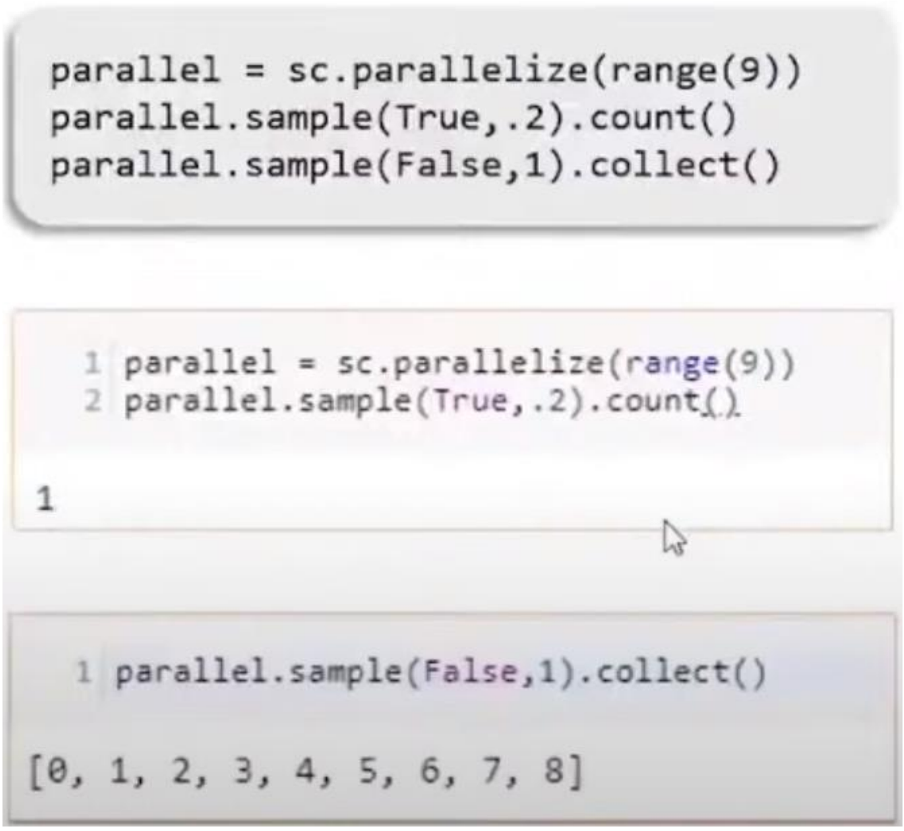


Figure 14.Sampling example.



```
parallel = sc.parallelize(range(9))
parallel.sample(True,.2).count()
parallel.sample(False,1).collect()
```

```
parallel = sc.parallelize(range(9))
parallel.sample(True,.2).count()

1
```

```
parallel.sample(False,1).collect()

[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Figure 15.Samping example.

Union

- Returns the union of two RDDs after concatenating their elements. It supports repeated elements.

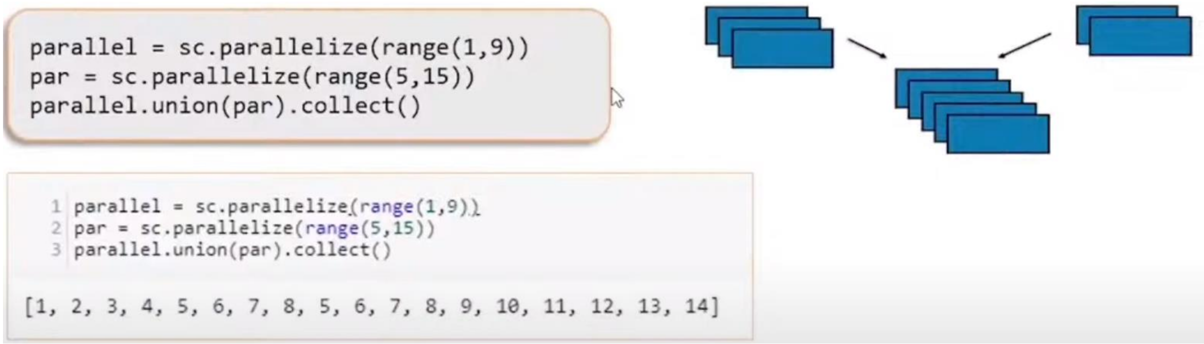


Figure 16. Union/concatening two RDDs.

Intersection

- Returns the intersection of two RDDs.

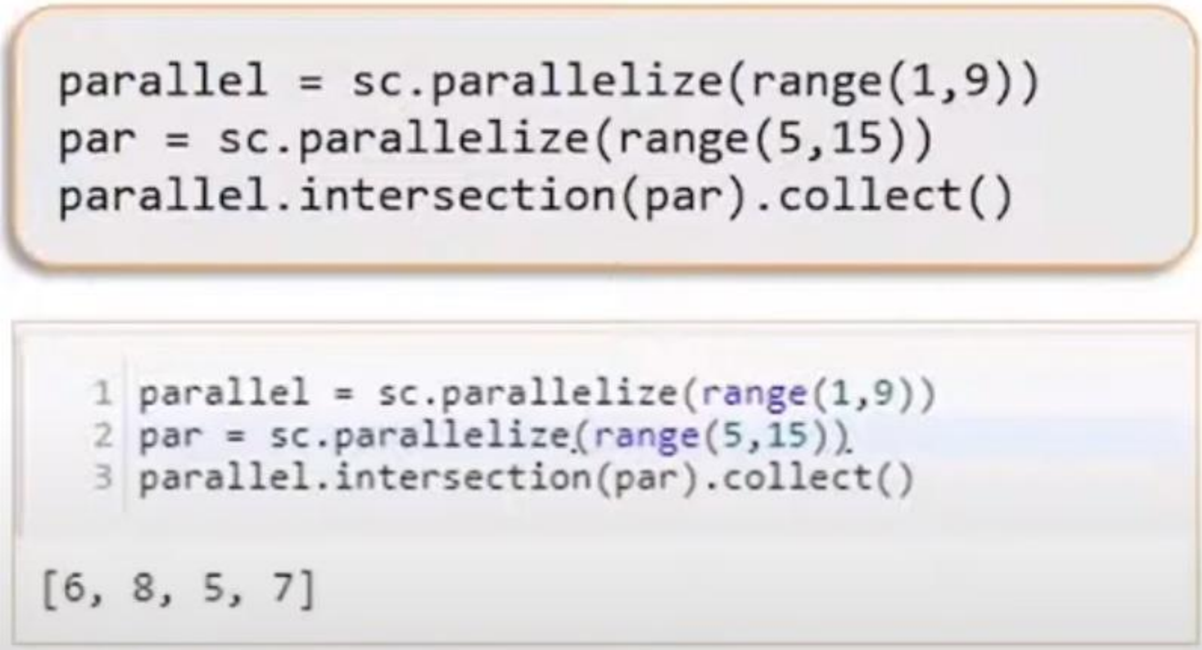


Figure 17. Interception of two RDD.

Distinct

- It returns a new RDD with distinct elements within the source data.

```
parallel = sc.parallelize(range(1,9))
par = sc.parallelize(range(5,15))
parallel.union(par).distinct().collect()
```

```
parallel = sc.parallelize(range(1,9))
par = sc.parallelize(range(5,15))
parallel.union(par).distinct().collect()

[2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13]
```

Figure 18. Distinct example.

SortBy

- It returns the RDD sorted by the given key function.

```
y = sc.parallelize([5, 7, 1, 3, 2, 1])
y.sortBy(lambda c: c, True).collect()
z = sc.parallelize([("H", 10), ("A", 26), ("Z", 1), ("L", 5)])
z.sortBy(lambda c: c, False).collect()
```

```
y = sc.parallelize([5, 7, 1, 3, 2, 1])
y.sortBy(lambda c: c, True).collect()

[1, 1, 2, 3, 5, 7]
```

```
z = sc.parallelize([("H", 10), ("A", 26), ("Z", 1), ("L", 5)])
z.sortBy(lambda c: c, False).collect()

[('Z', 1), ('L', 5), ('H', 10), ('A', 26)]
```

Figure 19. Sorting RDD by a given function.

MapPartitions

- Can be used as an alternative to ma() and foreach()
- It’s called once for each partition unlike map() and foreach(), which are called for each element in the RDD.
- They’re not indexed

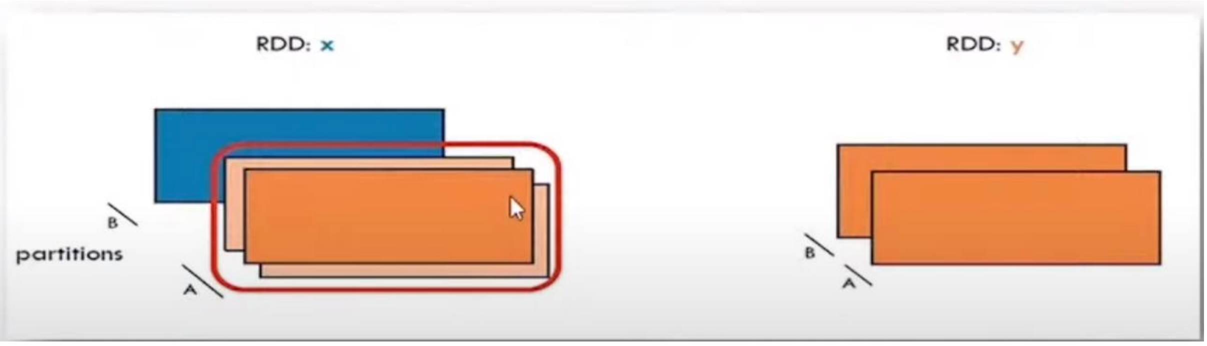


Figure 20.MapPartitions.

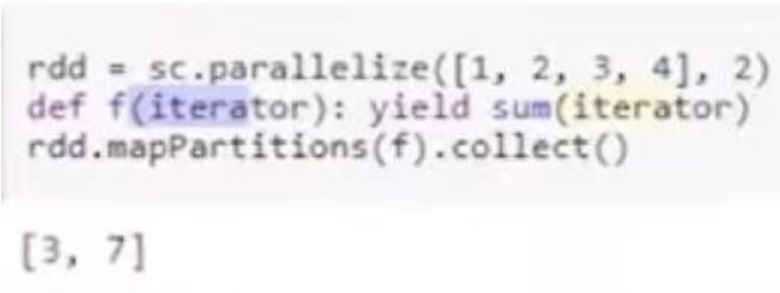


Figure 21.mapPartitions in python coding.

MapPartitions with Index

- It returns a new RDD by applying a function to each partition of the RDD, while tracking the index of the original partition.
- Returns a collection of element on the basis of the condition provided in the function.

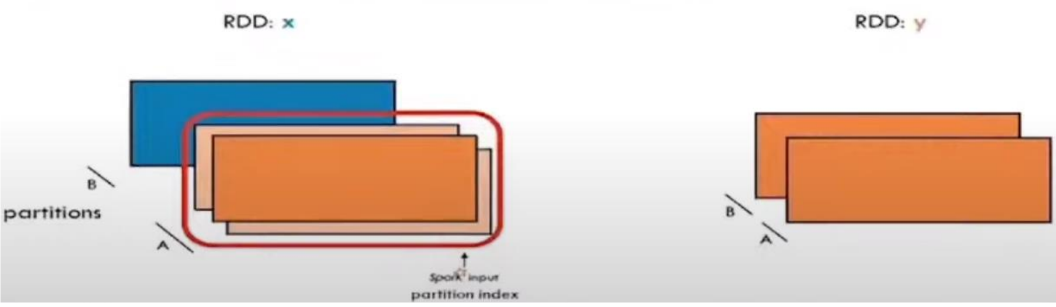


Figure 22.MapPartitions with index

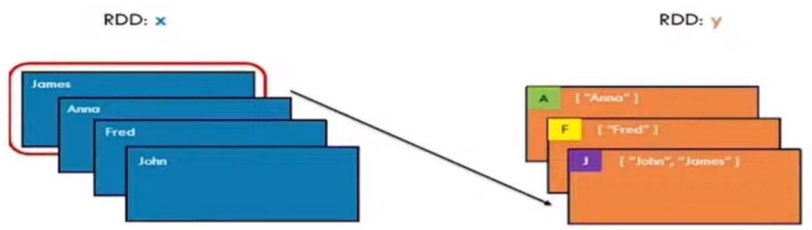
```
rdd = sc.parallelize([1, 2, 3, 4], 4)
def f(splitIndex, iterator): yield splitIndex
rdd.mapPartitionsWithIndex(f).sum()
```


```
rdd = sc.parallelize([1, 2, 3, 4], 4)
def f(splitIndex, iterator): yield splitIndex
rdd.mapPartitionsWithIndex(f).sum()
```


Figure 23. MapPartitions with index, code.

GroupBY

It returns a new RDD by grouping object in the existing RDD using the given grouping key or function.




```
rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
result = rdd.groupBy(lambda x: x %
2).collect()
sorted([(x, sorted(y)) for (x, y) in
result])
```

```
rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
result = rdd.groupBy(lambda x: x % 2).collect()
sorted([(x, sorted(y)) for (x, y) in result])

[(0, [2, 8]), (1, [1, 1, 3, 5])]
```

Figure 24. RDD groupBy

KeyBY

- It returns a new RDD by changing the key of the RDD element using the given key object

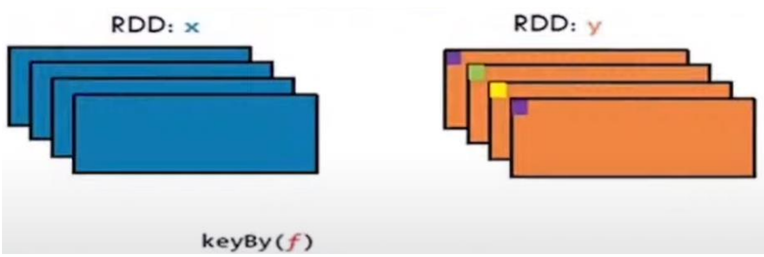


Figure 25. KeyBy

```
x = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
y = sc.parallelize(zip(range(0,5), range(0,5)))
[(x, list(map(list, y))) for x, y in sorted(x.cogroup(y).collect())]

[(0, [[0], [0]]),
 (1, [[1], [1]]),
 (2, [[], [2]]),
 (3, [[], [3]]),
 (4, [[2], [4]])]
```

Figure 26. KeyBy, coding version.

Zip

- Join two RDDs by combining the ith part of either partition with each other.
- Returns an RDD formed from this list and another iterable collection by combining the corresponding elements in pairs.
- If one of the two collections is longer than the other, its remaining elements are ignored.

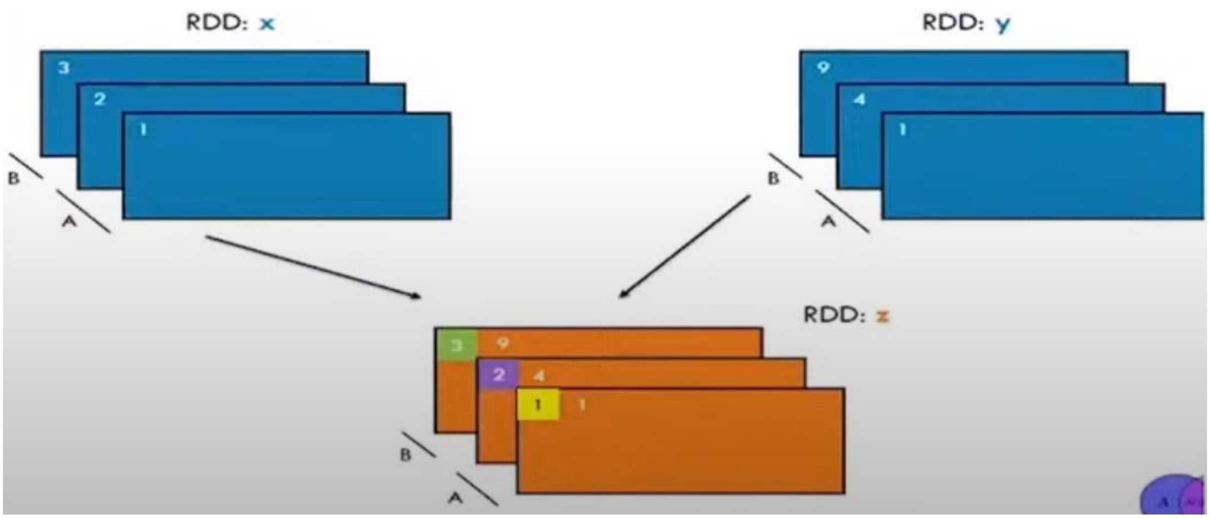


Figure 27. Zip.

```
x = sc.parallelize(range(0,5))
y = sc.parallelize(range(1000, 1005))
x.zip(y).collect()

[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

Figure 28. Coding example of ZIP.

ZIP with index

- Returns a new RDD that contains pairs consisting of all elements of the given list paired with their indices. Indices start at 0. See Fig 29.

```
sc.parallelize(["a", "b", "c", "d"],
3).zipWithIndex().collect()
```

```
1 sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()

[('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

Figure 29. Coding example ZIP with indexes

Repartition

- Used to either increase or decrease the number of partitions in a RDD.
- Does full shuffle an creates new partitions with the data that are distributed evenly. Fig 30.

```
rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
sorted(rdd.glom().collect())
len(rdd.repartition(2).glom().collect())
```

```
1 rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
2 sorted(rdd.glom().collect())

[[1], [2, 3], [4, 5], [6, 7]]

1 len(rdd.repartition(2).glom().collect())

2
```

Figure 30. Repartition.

Coalesce

- This method is used to reduce the number of partitions in an RDD.

```
sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
sc.parallelize([1, 2, 3, 4, 5],
3).coalesce(2).glom().collect()
```

```
1 sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()

[[1], [2, 3], [4, 5]]

1 sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(2).glom().collect()

[[1], [2, 3, 4, 5]]
```

Figure 31.Coalesce coding example.

Coalesce vs répartition

Coalesce()	Repartition()
Uses the existing partitions to minimize the amount of data that's shuffled	Creates new partitions and does a full shuffle
Results in partitions with different amounts of data (at times with much different sizes)	Results in roughly equal-sized partitions
Faster	Not so faster

Figure 32. Coalesce vs Répartition.

RDD Actions

- Unlike transformations that produce RDDs, action functions produce a value back to the spark driver program.
- Actions may trigger a previously constructed, lazy RDD to be evaluated.

Figure 33.List of RDD actions.

Reduce

- Aggregates element of a dataset through a function.

```
from operator import add
sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
sc.parallelize((2 for _ in range(10))).map(lambda x:
1).cache().reduce(add)
```

```
1 from operator import add
2 sc.parallelize([1, 2, 3, 4, 5]).reduce(add)

15

1 sc.parallelize((2 for _ in range(10))).map(lambda x: 1).cache().reduce(add)

10
```

Figure 34. Reduce coding example

Figure 35. Visual example of reduce.

First

- Returns the first element in an RDD

```
sc.parallelize([2, 3, 4]).first()
```

```
1 sc.parallelize([2, 3, 4]).first()

2
```

Figure 36. First coding example.

Take Ordered

Returns an array with the given number of ordered values in an RDD.

```
nums = sc.parallelize([1,5,3,9,4,0,2])
nums.takeOrdered(5)
```

```
nums = sc.parallelize([1,5,3,9,4,0,2])
nums.takeOrdered(5)

[0, 1, 2, 3, 4]
```

Figure 37. Coding example of takeOrdered.

Count

Returns a long value indicating the number of elements present in an RDD.

```
nums = sc.parallelize([1,5,3,9,4,0,2])
nums.count()
```

```
nums = sc.parallelize([1,5,3,9,4,0,2])
nums.count()

7
```

Figure 38. Coding example of count.

Collect

- Returns the elements of the dataset as an array back to the driver program.
- Should be used wisely as all worker nodes return the data to the driver node.
- If the dataset is huge in size, then this may result in an OutOfMemoryError.

```
c = sc.parallelize(["Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"], 2)
c.collect()
c = sc.parallelize(["Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"], 2)
c.distinct().collect()
```

```
c = sc.parallelize(["Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"], 2)
c.collect()

['Gnu', 'Cat', 'Rat', 'Dog', 'Gnu', 'Rat']

c = sc.parallelize(["Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"], 2)
c.distinct().collect()

['Cat', 'Rat', 'Gnu', 'Dog']
```

Figure 39. Coding example of collect.

SaveAsTextFile

- Writes the entire RDDs dataset as a text file on the path specified in the local filesystem or HDFS.

```
a = sc.parallelize(range(1,10000), 3)
a.saveAsTextFile("/usr/bin/mydata_a1")
x = sc.parallelize([1,2,3,4,5,6,6,7,9,8,10,21], 3)
x.saveAsTextFile("/usr/bin/sample1.txt")
```

```
a = sc.parallelize(range(1,10000), 3)
a.saveAsTextFile("/usr/bin/mydata_a1")

x = sc.parallelize([1,2,3,4,5,6,6,7,9,8,10,21], 3)
x.saveAsTextFile("/usr/bin/sample1.txt")
```

Figure 40. SaveAsText file coding example.

ForEach

- Passes each element in an RDD through the specified function

```
def f(x): print(x)
sc.parallelize([1,2,3,4,5]).foreach(f)
```


```
def f(x): print(x)
sc.parallelize([1,2,3,4,5]).foreach(f)
```


Figure 41. ForEach coding example.

Foreach - partition

Executes the function for each partition. Access to the data item contained in the partition is provided via the iterator argument.

```
def f(iterator):
  for x in iterator:
    print(x)
sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)
```


```
def f(iterator):
  for x in iterator:
    print(x)
sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)
```


Figure 42. ForEach partition coding example.

Mathematical functions

- Spark RDD supports some mathematical actions like “max”, “min”, “sum”, “mean”, “variance” and “stdev”.

```
numbers = sc.parallelize(range(1,100))
numbers.sum
numbers.min
numbers.variance
numbers.max
numbers.mean
numbers.stdev
```

Figure 43. Mathematical functions on RDDs.

RDD Functions

Functions	Arguments	Returns
cache	()	Caches an RDD to use without computing again
collect	()	Returns an array of all elements in an RDD
countByValue	()	Returns a map with the number of times each value occurs
distinct	()	Returns an RDD containing only distinct elements
filter	(f: T => Boolean)	Returns an RDD containing only those elements that match with the function **f**
foreach	(f: T => Unit)	Applies the function **f** to each of the elements of an RDD
persist	(); (newLevel: StorageLevel)	Sets an RDD with the default storage level (MEMORY_ONLY); sets the storage level that causes the RDD to be stored after it is computed (different storage levels are there in StorageLevel)

Figure 44. RDD list of functions (1/2).

Functions	Arguments	Returns
sample	(fraction: double)	Returns an RDD of that fraction
toDebugString	()	Returns a handy function that outputs the recursive steps of an RDD
count	()	Returns the number of elements in an RDD
unpersist	()	Removes all the persistent blocks of an RDD from the memory/disk
union	(other: RDD[T])	Returns an RDD containing elements of two RDDs; duplicates are not removed

Figure 45. RDD list of functions (2/2).

CountByValue

```
a = sc.parallelize([1,2,3,4,5,6,7,8,2,4,2,3,3,3,1,1,1])
a.countByValue()
```

```
a = sc.parallelize([1,2,3,4,5,6,7,8,2,4,2,3,3,3,1,1,1])
a.countByValue()

defaultdict(int, {1: 4, 2: 3, 3: 4, 4: 2, 5: 1, 6: 1, 7: 1, 8: 1})
```

Figure 46. Coding example of countByValue function.

toDebugString

```
a = sc.parallelize(range(1,19),3)
b = sc.parallelize(range(1,13),3)
c = a.subtract(b)
c.toDebugString()
```

```
a = sc.parallelize(range(1,19),3)
b = sc.parallelize(range(1,13),3)
c = a.subtract(b)
c.toDebugString()

b'(6) PythonRDD[165] at RDD at PythonRDD.scala:53 []\n
```

Figure 47. toDebugString coding example.

Creating Paired RDDs

- There are a number of ways to get paired RDDs in Spark
- There are many formats that directly return paired RDDs for their key-value data, whereas other cases, we have regular RDDs that need to be turned into paired RDDs.
- We can do this by running a map() function that return key-value pairs.

Transformations on Paired RDDs

- Transformations on one paired RDD
- E.G… RDD = {(1,2),(3,4),(3,6)}

Function	Purpose	Example	Result
reduceByKey(func)	Combines values with the same key	rdd.reduceByKey(add)	{(1, 2), (3, 10)}
groupByKey()	Groups values with the same key	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
mapValues(func)	Applies a function to each value of a paired RDD without changing the key	rdd.mapValues(lambda x: x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Applies a function that returns an iterator to each value of a paired RDD and, for each element returned, produces a key–value entry with the old key; often used for tokenization	rdd.flatMapValues(lambda x: range(x, 5))	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}
keys()	Returns an RDD of just the keys	rdd.keys()	{1, 3, 3}
sortByKey()	Returns an RDD sorted by the key	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Figure 48. Transformations on one paired RDD.

Transformations on Paired RDDs

- Transformations on two paired RDDs
- **E.g.: RDD1** = {(1, 2), (3, 4), (3, 6)} **RDD2** = {(3, 9)}

Function	Purpose	Example	Result
subtractByKey	Removes elements with the key present in the other RDD	rdd.subtractByKey(other)	{(1, 2)}
join	Performs an inner join between both RDDs	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Performs a join between two RDDs where the key must be present in the first RDD	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Performs a join between two RDDs where the key must be present in the other RDD	rdd.leftOuterJoin(other)	{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}
cogroup	Groups data from both RDDs sharing the same key	rdd.cogroup(other)	{(1,([2],[])), (3, ([4, 6],[9]))}

Figure 49. Transformations on Pair RDDs

RDD Lineage

- RDD Lineage is a graph of all parent RDDs of an RDD.
- It’s built by applying transformations to the RDD and creating a logical execution plan.
- Consider the following series of transformations.

```
rdd.toDebugString #to print rdd lineage
```

```
PythonRDD[141] at RDD at PythonRDD.scala:53 []\n
MapPartitionsRDD[140] at mapPartitions at
PythonRDD.scala:133 []\n
ShuffledRDD[139] at partitionBy at
NativeMethodAccessorImpl.java:0 []\n
PairwiseRDD[138] at subtract at <ipython-input-58-
e1f9a4054d92>:3 []\n
```

Figure 50. RDD Lineage.

- The following RDD graph is the result of the series of transformation mentioned earlier.
- An RDD lineage graph is hence a graph of transformations that need to be executed after an action has been called.
- We can create an RDD lineage graph using the RDD.toDebugString method.

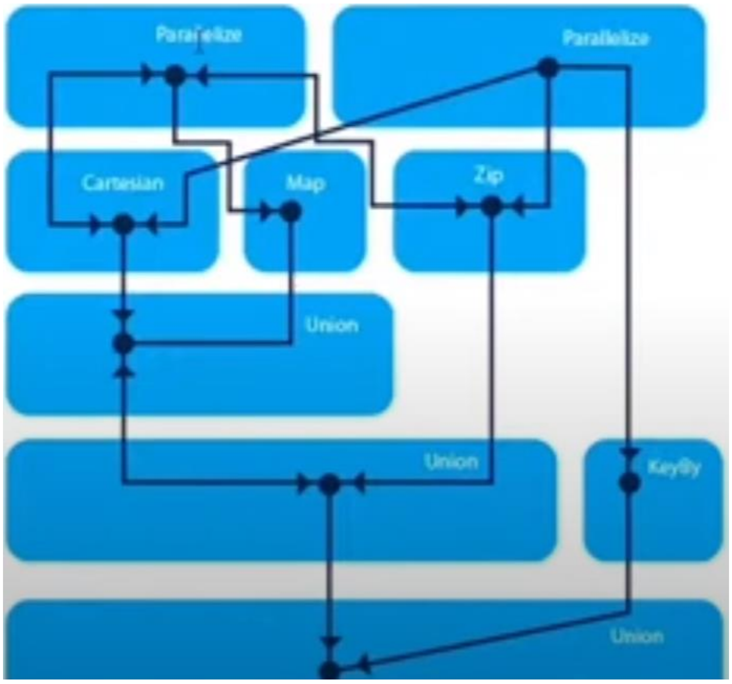


Figure 51. RDD graph result of transformations at Fig 50.

Word Count Program

- A word count program will return the frequency of every word.
- An RDD is created to store a text file data.
- A few RDD operations can complete this program.

```
rdd =sc.textFile("/content/Pyspark.txt")
nonempty_lines = rdd.filter(lambda x: len(x) > 0)
words = nonempty_lines.flatMap(lambda x: x.split(' '))
wordcount = words.map(lambda x:(x,1)).reduceByKey(lambda x,y: x+y).map(lambda x: (x[1], x[0])).sortByKey(False)
for word in wordcount.collect():
   print(word)
wordcount.saveAsTextFile("/content/Wordcount")
```

Figure 52. Wordcount Program

RDD Partitioning

- A partition is a logical chunk of a large, distributed dataset.
- Spark manages data using partition that help parallelize distributed data processing with minimal network traffic for sending data between executors.
- By default, Spark tries to read data into RDD from the nodes that are close to it.
- Here, an RDD is split into 5 Partitions.

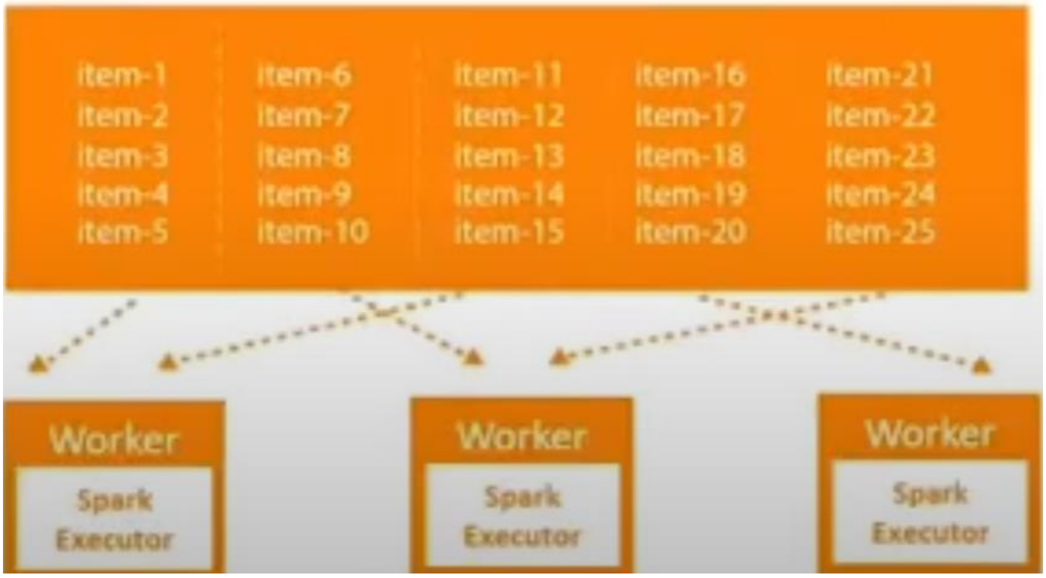


Figure 53. RDD Partitioning

- Since Spark usually accesses distributed partitioned data to optimize transformations, it creates partitions that can hold the data chunks.
- RDD get partitioned automatically without a programmer's intervention.
- However, there are times when we would like to adjust the size and number of partitions or the partitioning scheme according to the needs of our application.
- We can use the def getPartitions:Array[Partition] method on an RDD to know the number of partitions n the RDD.

RDD Partitioning: Types

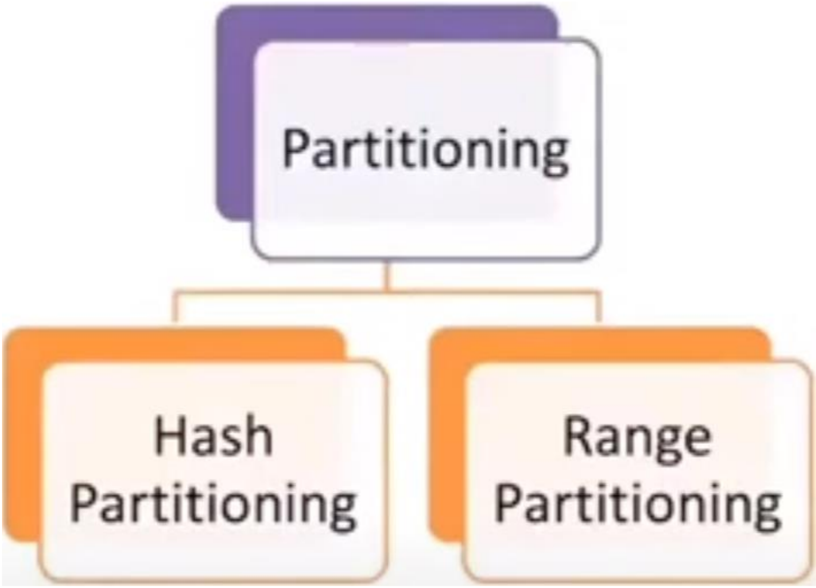


Figure 54. Types of partitioning.

- Customizing partitioning is possible only on paired RDDs.
- One of the basic advantages is that as similar kinds of data is co-located, shuffling of data across clusters reduces in transformations like groupByKey, reduceByKey, etc. which in turn increases job performance.

HashPartitioner

Hash partitioning is a partitioning technique where a hash key is used to distribute element evenly across different partitions.

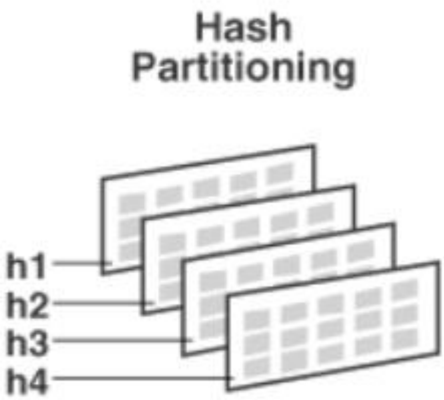


Figure 55. Hash partitioning.

Range partitioning

- Some Spark RDDs have keys that follow a particular order; for such RDDs, range partitioning is an efficient partitioning technique.
- In the range partitioning method, tuples having keys within the same range will appear on te same machine.
- Key in a RangePartitioner is partitioned based on the set of sorted range of keys and ordering of keys.
- This involves three steps.

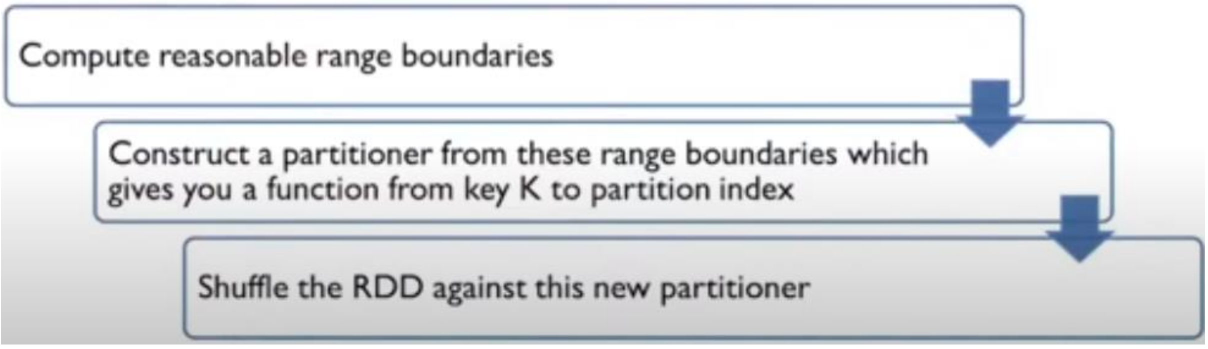


Figure 56. Range partitioner.

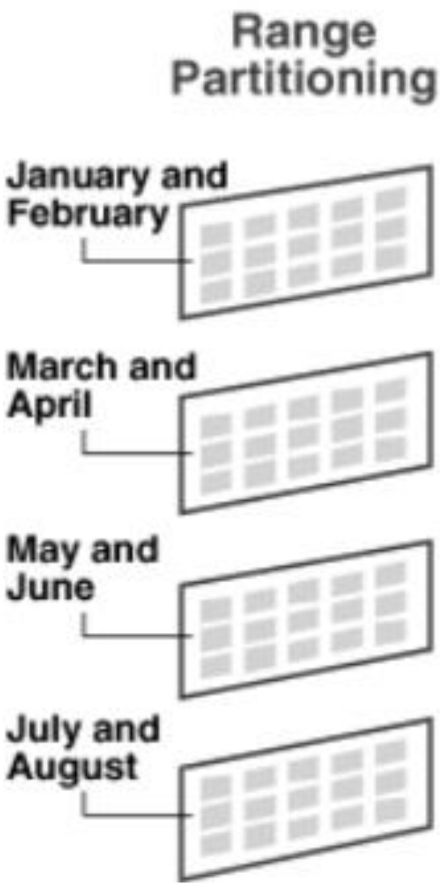


Figure 57. Range partitioning.

Passing functions to spark

- Most of Spark's transformations, and some of its actions, depends on passing in functions that are used by spark to compute data.
- Each of the core languages (Python, Java and Scala) has a slightly different mechanism for passing functions to spark.
- In python, we can pass a function inside another function, similar to python's other functional APIs.
- Although, some other considerations come into play – namely, the function we pass and the data reference in it. Needs to be serializable.
- Spark's API relies heavily on passing functions in the driver program to run on the cluster.

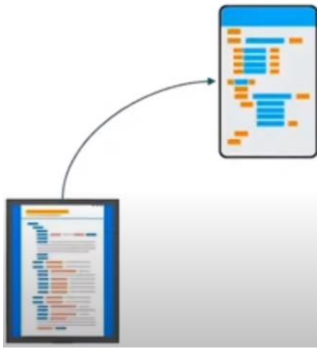


Figure 58. Passing functions to spark.

- Two recommended ways of doing this:
    - Anonymous functions: methos used for short pieces of code.
    - Static methods: Used for a global singleton object.

Anonymous function

- E.G., lambda function that returns a given integer plus 2.
    - Lambda x: x + 2
- On the left of ":" is a list of parameters and on the right is an expression involving the parameters.
- We can name the functions as well.
    - Rdd = sc.parallelize([1,2,3,4,5]
    - Rdd.map(lambda x: x+2).collect()
- Functions may take multiple parameters, or they can take no parameters.

Passing functions to static methods

- Static methods, used for a global singleton object.
- For example, we can define the object MyFunctions and then pass MyFunctions.func1 as follows:
- Object MyFunctions{
    - Def func1(s: string): String ={ … }
- }myRdd.map(MyFunctions.func1)
- It's also possible to pass a reference to a method in a class instance (as opposed to a singleton object), which requires sending the object that contains that class along with the method.
- For ex
- Consider the following class.
- Class MyClass{
    - Def func1(s: String): String = {…}
    - Def doStuff(rdd: RDD[String]): RDD[String] = {rdd.map(func1)}
- }
- Here, if we create a new MyClass instance and call doStuff on it, the map inside reference the func1 method of that MyClass instance, so the whole object needs to be sent to the cluster.
- It's similar to writing rdd.map(x => this.func1(x)).