

Prepared By : T Raveendra Reddy



Source: Azure Data Bricks & www.databricks.com

Spark Architecture & Components



Apache Spark is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics.

Spark SQL +
DataFrames

Streaming

MLlib
*Machine
Learning*

GraphX
*Graph
Computation*

Spark Core API

R

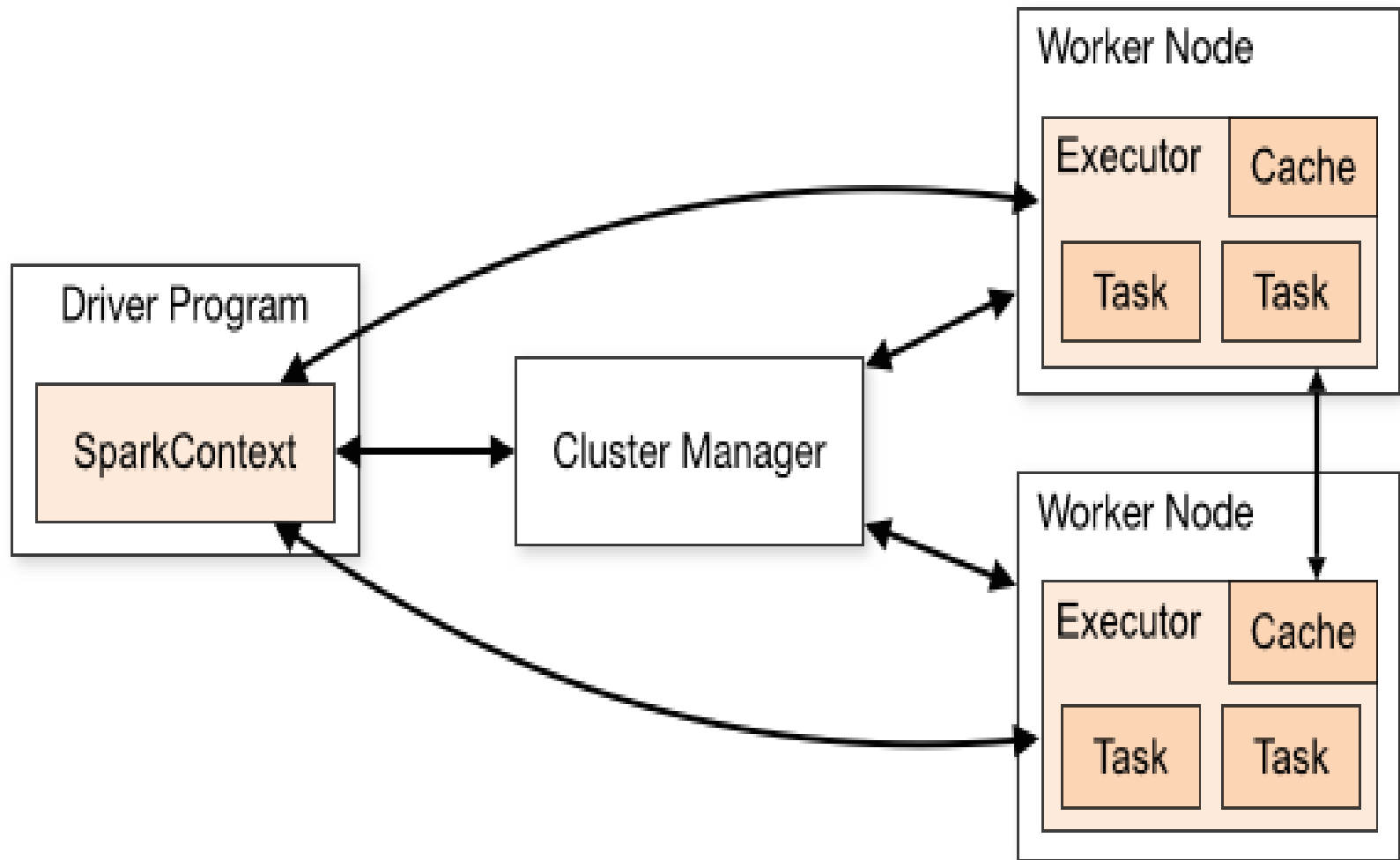
SQL

Python

Scala

Java

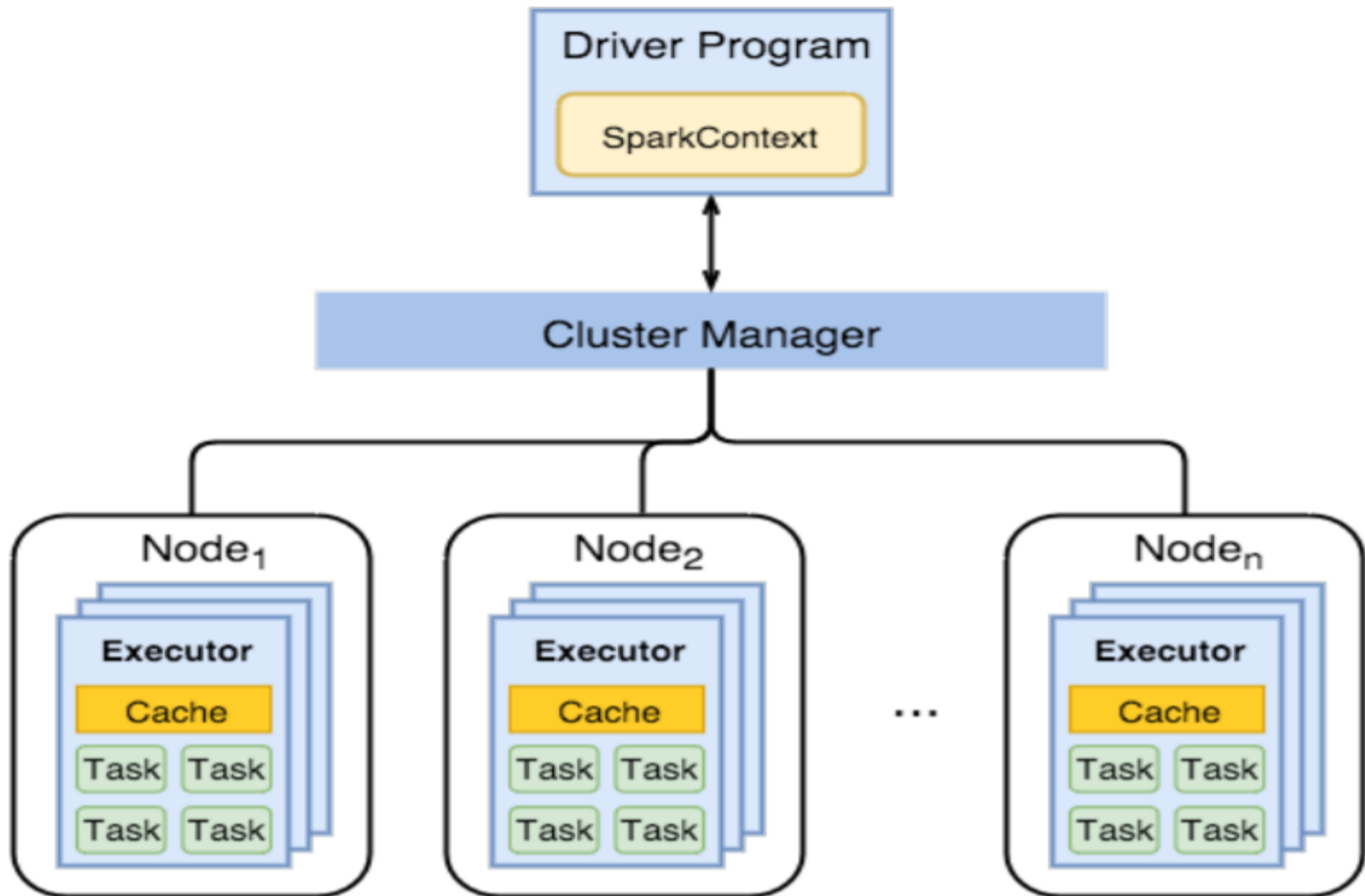
SPARK EXECUTION



SPARK EXECUTION

- Job: A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- Stages: Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- Tasks: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- DAG: DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- Executor: The process responsible for executing a task.
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs

SPARK EXECUTION



Spark API's

1) Resilient Distributed Dataset (RDD) => (Spark1.0)

An RDD stands for Resilient Distributed Datasets. It is Read-only partition collection of records. RDD is the fundamental data structure of Spark. It allows a programmer to perform in-memory computations on large clusters in a fault-tolerant manner. Thus, speed up the task

RDD Limitations: Handling structured data -> Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

2) DataFrame => (Spark1.3)

DataFrame data organized into named columns. For example a table in a relational database. It is an immutable distributed collection of data. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction.

Dataframe Limitations: Compile-time type safety

3) Dataset => (Spark1.6)

It is an extension to Dataframe API, the latest abstraction which tries to provide best of both RDD and Dataframe. Datasets API provides compile time safety which was not available in Dataframes.

DataSet Provides best of both RDD and Dataframe

RDD (functional programming, type safe),

DataFrame (relational model, Query optimazation , Tungsten execution, sorting and shuffling)

The Dataset API is available in Scala and Java

Initializing Pyspark

▶ `pip install pyspark`

```
Requirement already satisfied: pyspark in c:\users\raveendra\anaconda3\envs\spark\lib\site-packages  
Requirement already satisfied: py4j==0.10.9 in c:\users\raveendra\anaconda3\envs\spark\lib\site-packages  
Note: you may need to restart the kernel to use updated packages.
```

▶ `pip install findspark`

```
Requirement already satisfied: findspark in c:\users\raveendra\anaconda3\envs\spark\lib\site-packages  
Note: you may need to restart the kernel to use updated packages.
```

▶ *#### Importing findspark and intializing findspark...*

▶ `import findspark`
`findspark.init()`
#findspark.init(SPARK_HOME)

Creating Pyspark Session in local system

▶ `#### Creating Spark Session`

▶ `from pyspark.sql import SparkSession
#spark = SparkSession.builder\
#.master("local".format(2))\
#.appName("Localspark")\
#.getOrCreate()

n_cpu = 2
spark = SparkSession.builder.appName('localSpark')\
#.master('local[{}]'.format(n_cpu))\
#.config("spark.driver.memory", "1g")\
#.config('spark.executor.memory', '2g')\
#.config('spark.executor.cores', '3')\
#.config('spark.cores.max', '3')\
#.getOrCreate()`

▶ `spark`

`]: SparkSession - in-memory
SparkContext`

Creating SparkContext in local System

```
▶ from pyspark import SparkContext  
  
sc = SparkContext.getOrCreate()  
  
sc
```

]: **SparkContext**

[Spark UI](#)

Version

v3.0.1

Master

local[2]

AppName

localSpark

Stopping SparkSession & SparkContext using STOP() method

```
▶ #stop spark session  
#spark.stop()  
# stop SparkContext  
#sc.stop()
```

Get All Spark Config Parameter Values

```
► #Get all configuration  
sc.getConf().getAll()
```

```
.5]: [('spark.driver.memory', '4g'),  
      ('spark.executor.memory', '4g'),  
      ('spark.executor.id', 'driver'),  
      ('spark.executor.cores', '4'),  
      ('spark.cores.max', '4'),  
      ('spark.driver.host', 'VICKY'),  
      ('spark.app.id', 'local-1603445567945'),  
      ('spark.app.name', 'Spark Updated Conf'),  
      ('spark.rdd.compress', 'True'),  
      ('spark.serializer.objectStreamReset', '100'),  
      ('spark.submit.pyFiles', ''),  
      ('spark.submit.deployMode', 'client'),  
      ('spark.ui.showConsoleProgress', 'true'),  
      ('spark.master', 'local[2]'),  
      ('spark.driver.port', '50729')]
```

Setting SparkContext Configuration Parameters Values

SET Configuration parameters in spark

Cmd 11

[illegible]

RDD (Resilient Distributed Dataset)

Terminologies

- RDD stands for Resilient Distributed Dataset, these are the elements that run and operate on multiple nodes to do parallel processing on a cluster.

RDDs are...

- immutable
- fault tolerant / automatic recovery
- can apply multiple ops on RDDs

RDD operation are...

- Transformation
- Action

Basic Operations (Ops)

- `count()`: Number of elements in the RDD is returned.
- `collect()`: All the elements in the RDD are returned.
- `foreach(f)`: input callable, and returns only those elements which meet the condition of the function inside `foreach`.
- `filter(f)`: input callable, and returns new RDDs containing the elements which satisfy the given callable
- `map(f, preservesPartitioning = False)`: A new RDD is returned by applying a function to each element in the RDD
- `reduce(f)`: After performing the specified commutative and associative binary operation, the element in the RDD is returned.
- `join(other, numPartitions = None)`: It returns RDD with a pair of elements with the matching keys and all the values for that particular key.
- `cache()`: Persist this RDD with the default storage level (`MEMORY_ONLY`). You can also check if the RDD is cached or not

Spark Operations



TRANSFORMATIONS

An illustration showing the stages of a butterfly's life cycle: a green caterpillar, a chrysalis, and two adult butterflies, positioned above the word 'TRANSFORMATIONS'.

Spark Operations =

+



ACTIONS

Transformations and Actions



Transformations <i>(lazy)</i>	Actions
select	show
distinct	count
groupBy	collect
sum	save
orderBy	
filter	
limit	



= easy



= medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS



- reduce
- collect
- aggregate
- fold
- first
- take
- foreach
- top
- treeAggregate
- treeReduce
- foreachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

 = easy

 = medium

Essential Core & Intermediate PairRDD Operations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure

- partitionBy

ACTIONS

- keys
- values

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact





vs

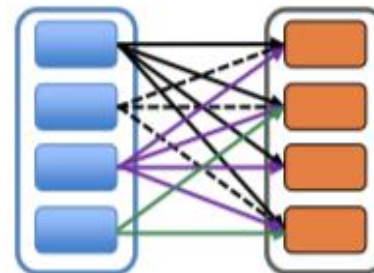
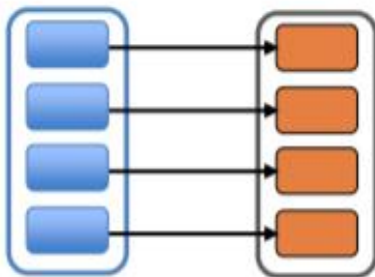


narrow

wide

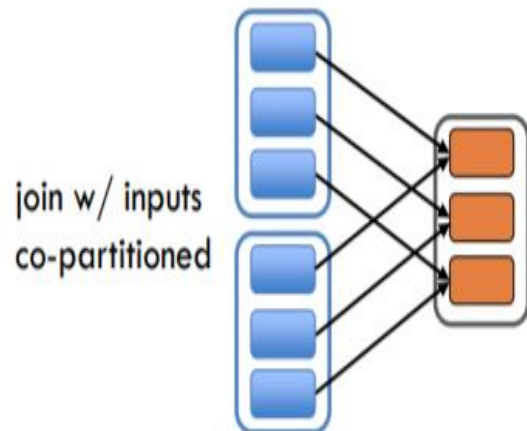
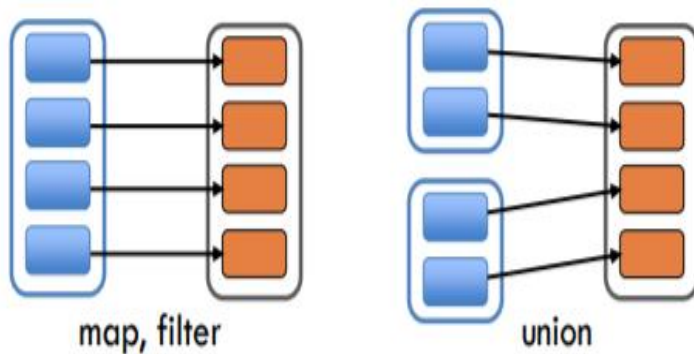
*each partition of the parent RDD is used by
at most one partition of the child RDD*

*multiple child RDD partitions may depend
on a single parent RDD partition*



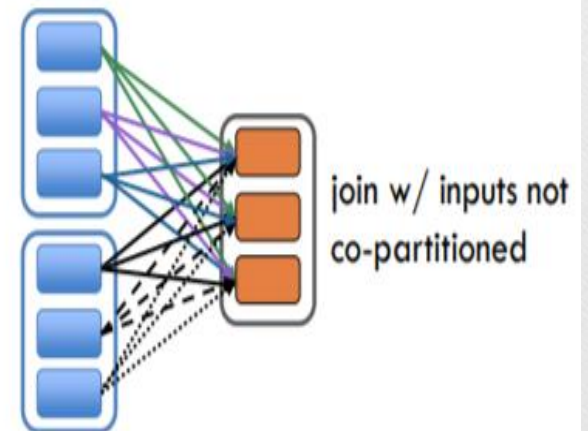
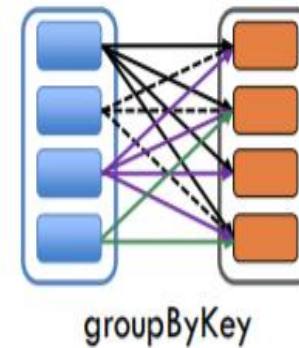
narrow

*each partition of the parent RDD is used by
at most one partition of the child RDD*



wide

*multiple child RDD partitions may depend
on a single parent RDD partition*



An RDD can be created 2 ways

- Parallelize a collection

```
# Parallelize in Python
wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
```

- Take an existing in-memory collection and pass it to SparkContext's parallelize method
- Not generally used outside of prototyping and testing since it requires entire dataset in memory on one machine

- Read from File

```
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

- There are other methods to read data from HDFS, C*, S3, HBase, etc.

Creating an RDD

- Create RDDs from Python collections (lists)

```
>>> data = [1, 2, 3, 4, 5]
```

```
>>> data
```

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

```
>>> rDD = sc.parallelize(data, 4)
```

```
>>> rDD
```

```
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with `sc.parallelize()`

- Spark only records how to create the RDD with four partitions



Creating RDD using SparkContext

- Applying MAP Transformation in RDD.
- MAP(func)
- Return a new distributed dataset formed by passing each element of the source through a function func.

Cmd 2

```
1 x=sc.parallelize([1,2,3])
2 y=x.map(lambda z: z*2 )
```

Command took 0.07 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 5:18:24 PM on datacluster

Cmd 3

```
1 y.collect()
```

► (1) Spark Jobs

Out[13]: [2, 4, 6]

Lambda Function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Cmd 2

```
1 x = lambda a : a + 10
2 print(x(5))
3 print(x(10))
```

Cmd 3

```
1 x = lambda a, b : a * b
2 print(x(5, 6))
```

Creating RDD using SparkContext

- Applying MAP Transformation in RDD.
- MAP(func)
- Return a new distributed dataset formed by passing each element of the source through a function func.

Cmd 37

```
1 x=sc.parallelize([1,2,3,4,5,6,7])
2 y=x.map(lambda a: a+10 )
3 print(x.collect())
4 print(y.collect())
```

► (2) Spark Jobs

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

[11, 12, 13, 14, 15, 16, 17]

Command took 4.06 seconds -- by pysparktelugu@gmail.com at 12/15/2020, 6:57:25 AM on datacluster

Cmd 38

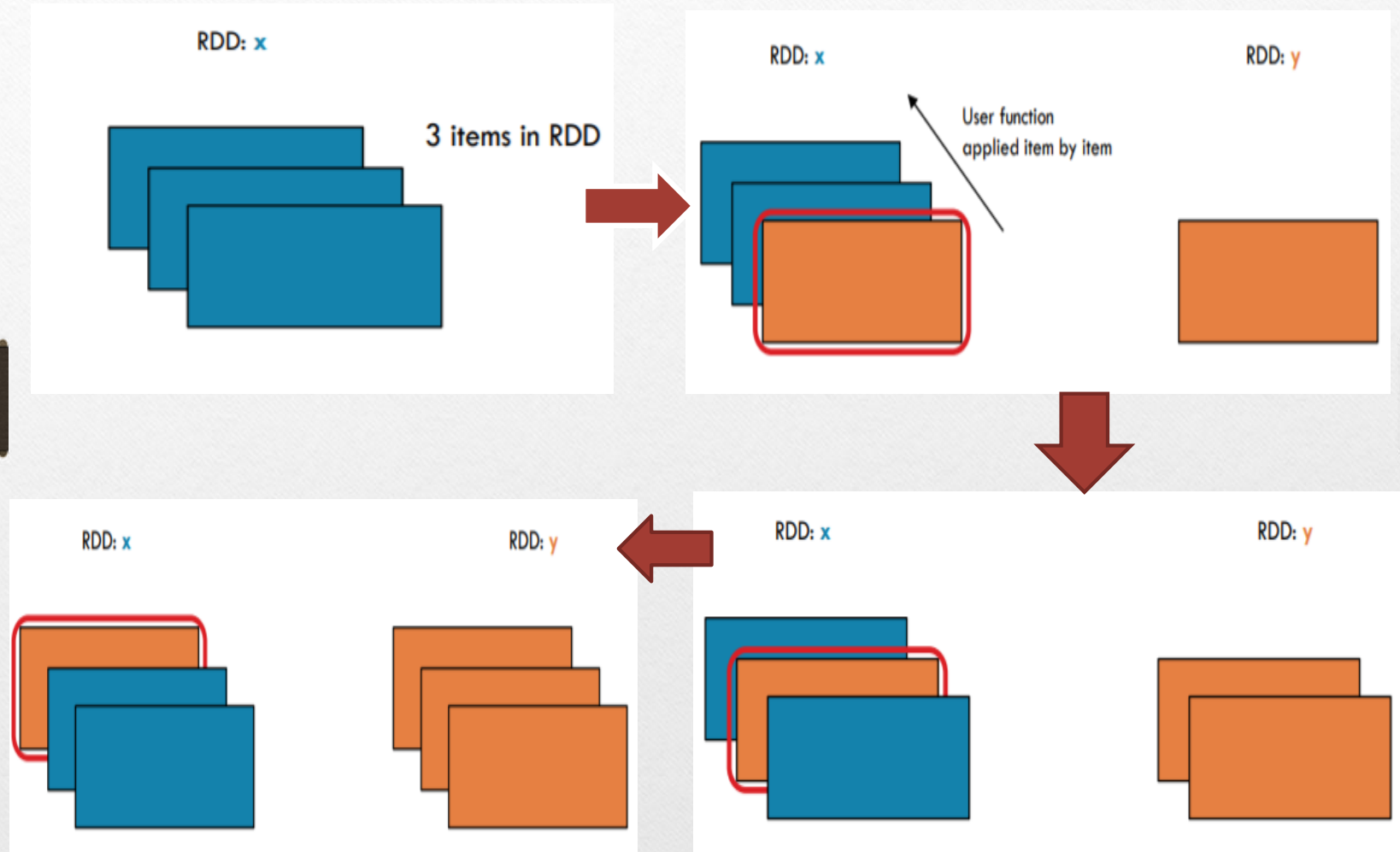
```
1 print("X RDD Values :",x.collect())
2 print("Y RDD Values after applying map and lambda transformation",y.collect())
```

► (2) Spark Jobs

X RDD Values : [1, 2, 3, 4, 5, 6, 7]

Y RDD Values after applying map and lambda transformation [11, 12, 13, 14, 15, 16, 17]

MAP Transformation



Filter Transformation in RDD

filter(func) Transformation

- Return a new dataset formed by selecting those elements of the source on which func returns true.

Cmd 5

```
1 x = sc.parallelize([1,2,3])
2 y = x.filter(lambda x: x%2 == 1) #keep odd values
3 print('RDD X Values Before Filter : ',x.collect())
4 print('RDD Y Values After aplying Filter in X RDD : ',y.collect())
```

► (2) Spark Jobs

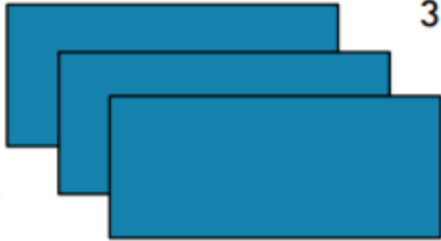
RDD X Values Before Filter : [1, 2, 3]

RDD Y Values After aplying Filter in X RDD : [1, 3]

Command took 0.34 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 5:32:43 PM on datacluster

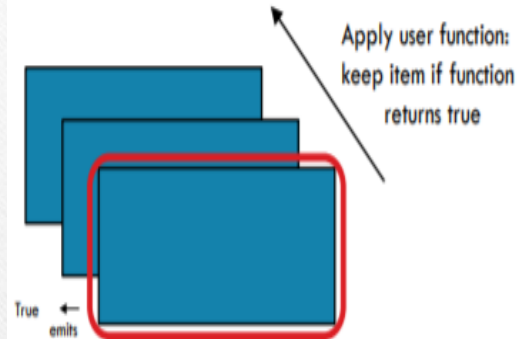
RDD: x

3 items in RDD



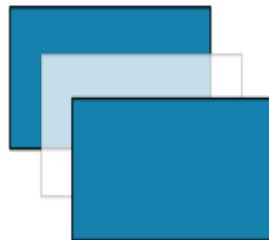
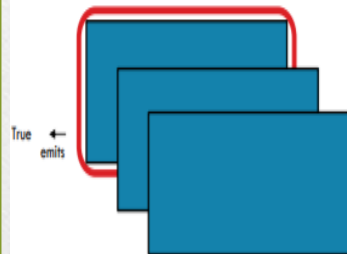
RDD: x

RDD: y



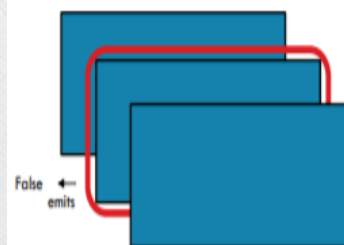
RDD: x

RDD: y



RDD: x

RDD: y



flatMap Transformation in RDD

flatMap(func) Transformation

- Similar to map, but each input item can be mapped to 0 or more output items
- (so func should return a Seq rather than a single item).

Cmd 7

```
1 x = sc.parallelize([1,2,3])
2 y = x.flatMap(lambda x: (x, x*100, 66))
3 print(x.collect())
4 print(y.collect())
```

► (2) Spark Jobs

[1, 2, 3]

[1, 100, 66, 2, 200, 66, 3, 300, 66]

Command took 0.30 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 5:42:40 PM on da

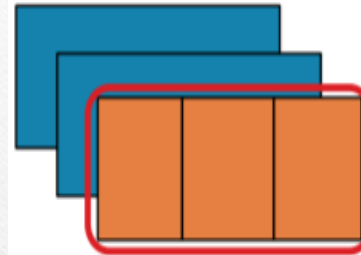
RDD: x



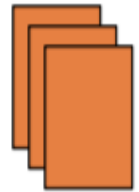
3 items in RDD



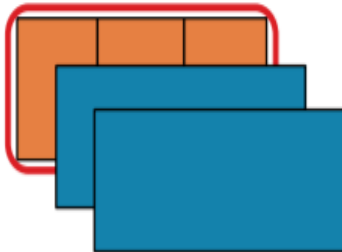
RDD: x



RDD: y



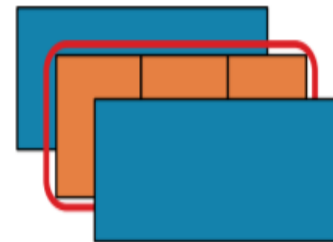
RDD: x



RDD: y



RDD: x



RDD: y



groupBy() Transformation in RDD

groupBy(func) Transformation in RDD

- Group the data in the original RDD. Create pairs where the key is the output of
- a user function, and the value is all items for which the function yields this key

Cmd 9

```
1 x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
2 y = x.groupBy(lambda w: w[0])
3 print(x.collect())
4 print([(k, list(v)) for (k, v) in y.collect()])
```

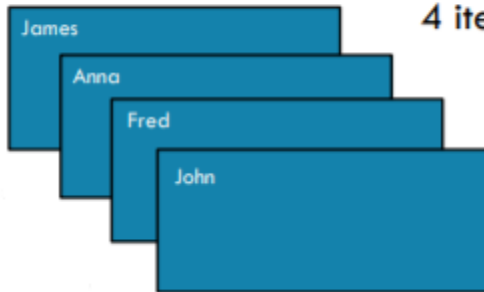
► (2) Spark Jobs

['John', 'Fred', 'Anna', 'James']

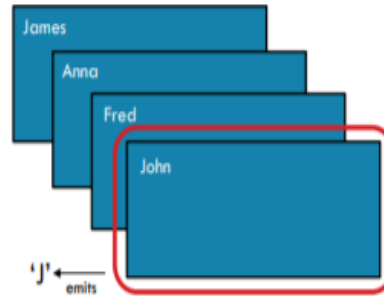
[('J', ['John', 'James']), ('F', ['Fred']), ('A', ['Anna'])]

Command took 0.78 seconds -- by pysparktelugu@gmail.com at 10/23/2020, 6:15:30 PM on datac

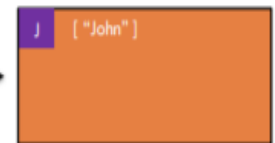
RDD: x



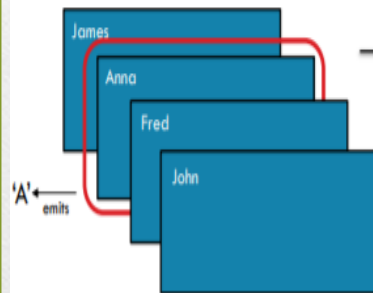
RDD: x



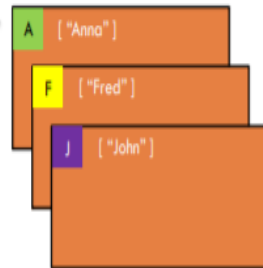
RDD: y



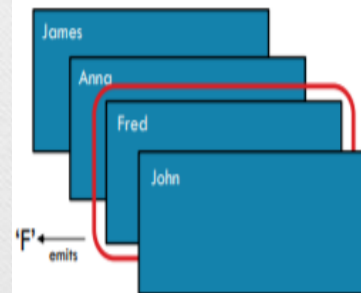
RDD: x



RDD: y

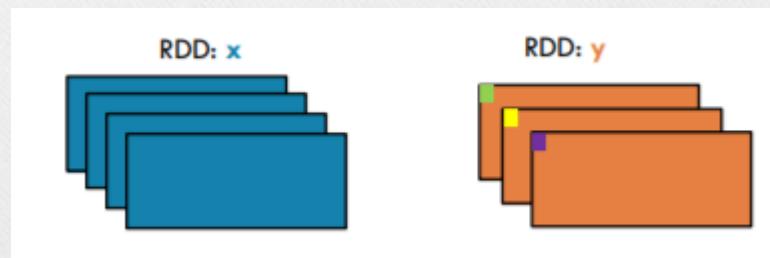
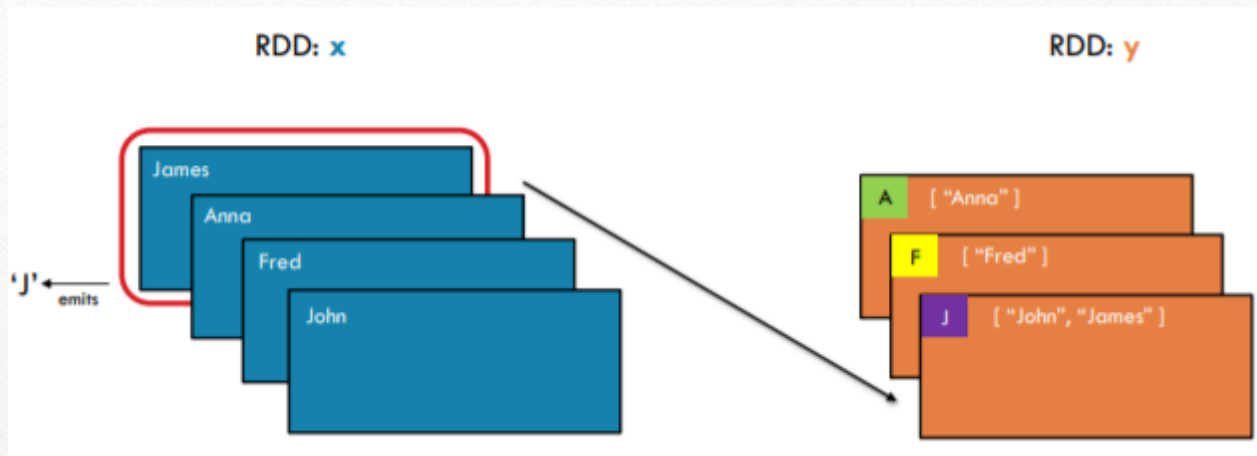


RDD: x



RDD: y





Sample Transformation

* Return a sampled subset of this RDD.

```
1 x = sc.parallelize([1,2, 3, 4, 5])
2 y = x.sample(False, 0.4, 15)
3 print(x.collect())
4 print(y.collect())
```

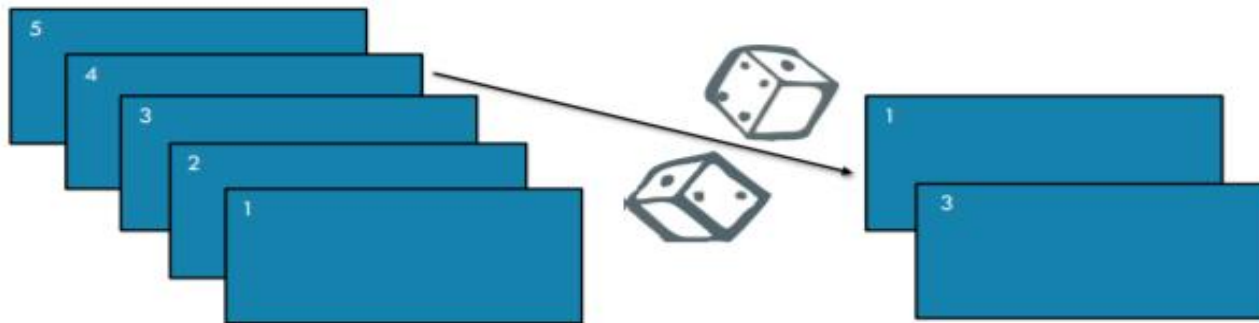
► (2) Spark Jobs

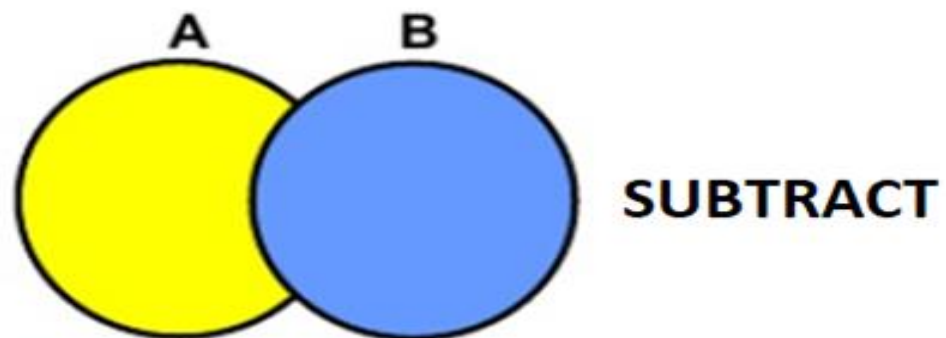
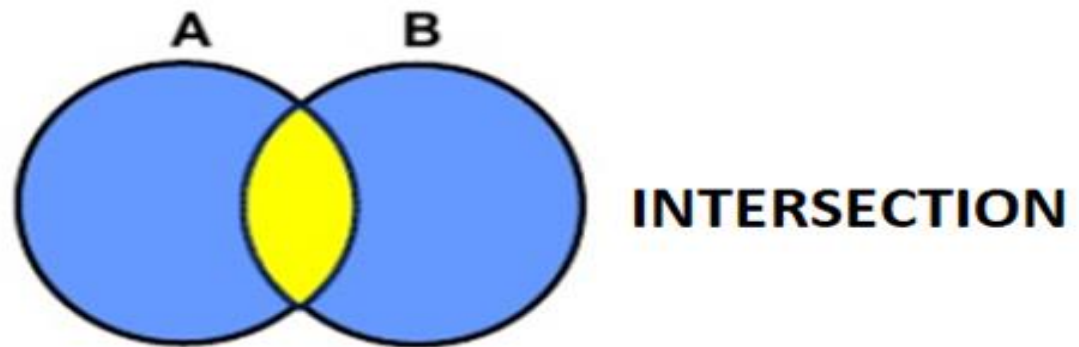
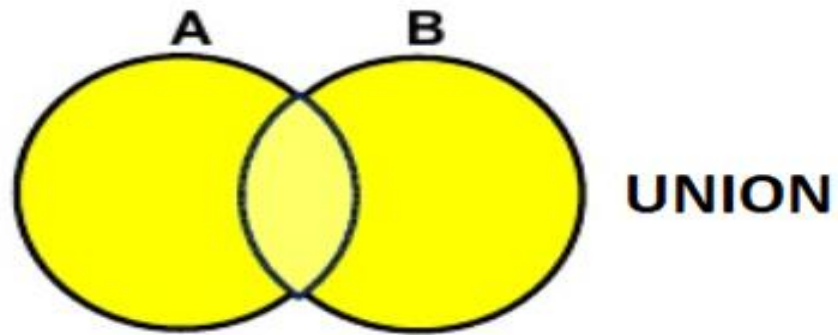
[1, 2, 3, 4, 5]

[1, 3]

RDD: x

RDD: y





Union(DataSet) Transformation

- Return a new dataset that contains the union of the elements in the source dataset and the argument.
- `glom()` Return an RDD created by coalescing all elements within each partition into an array

Cmd 41

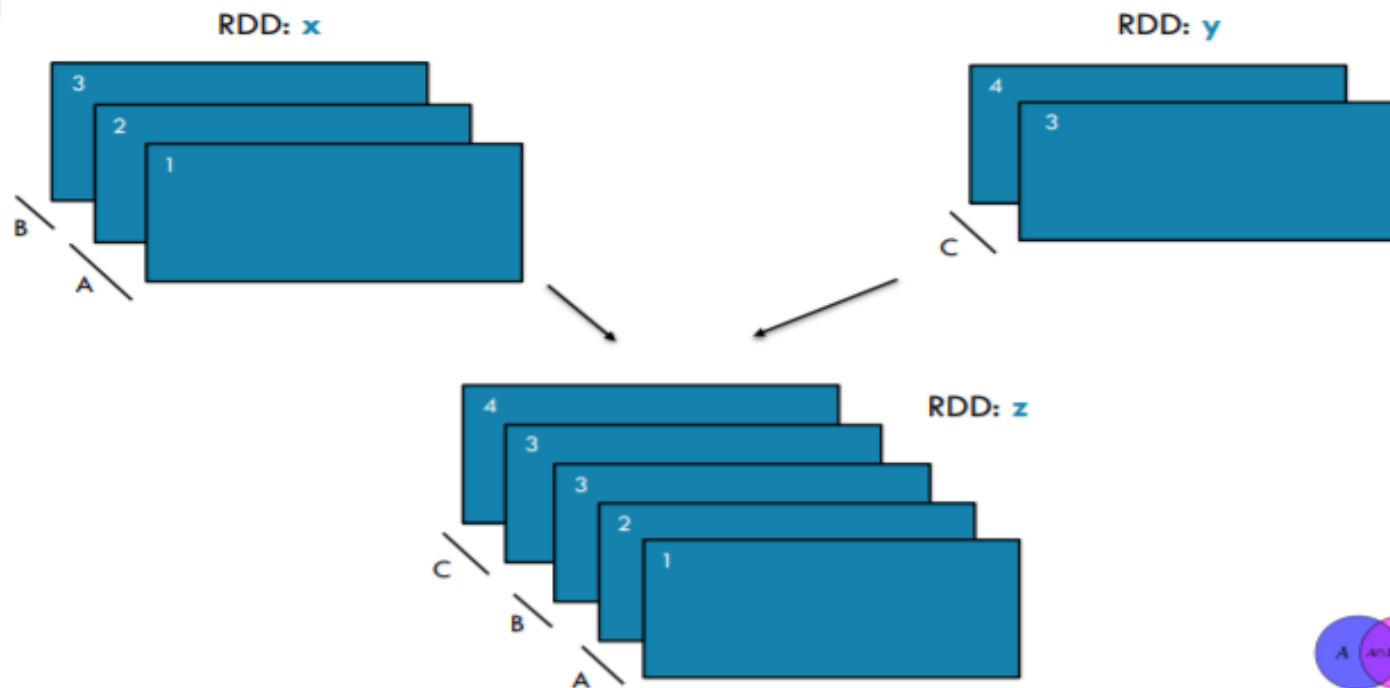
```
1 x = sc.parallelize([1,2,3], 2)
2 y = sc.parallelize([3,4], 1)
3 z = x.union(y)
4 print(z.collect())
5 print(z.glom().collect())
```

► (2) Spark Jobs

[1, 2, 3, 3, 4]

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

UNION



```
1 x = sc.parallelize([1,2,3], 2)
2 y = sc.parallelize([3,4], 1)
3 z = x.union(y)
4 print(z.glom().collect())
```

► (1) Spark Jobs

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

intersection(otherDataset)

- Return a new RDD that contains the intersection of elements in the source dataset and the argument.

Cmd 43

```
1 x = sc.parallelize([1,2,3,4,5], 2)
2 y = sc.parallelize([3,4,5,6,7], 1)
3 z = x.intersection(y)
4 print(z.collect())
5 print(z.glom().collect())
```

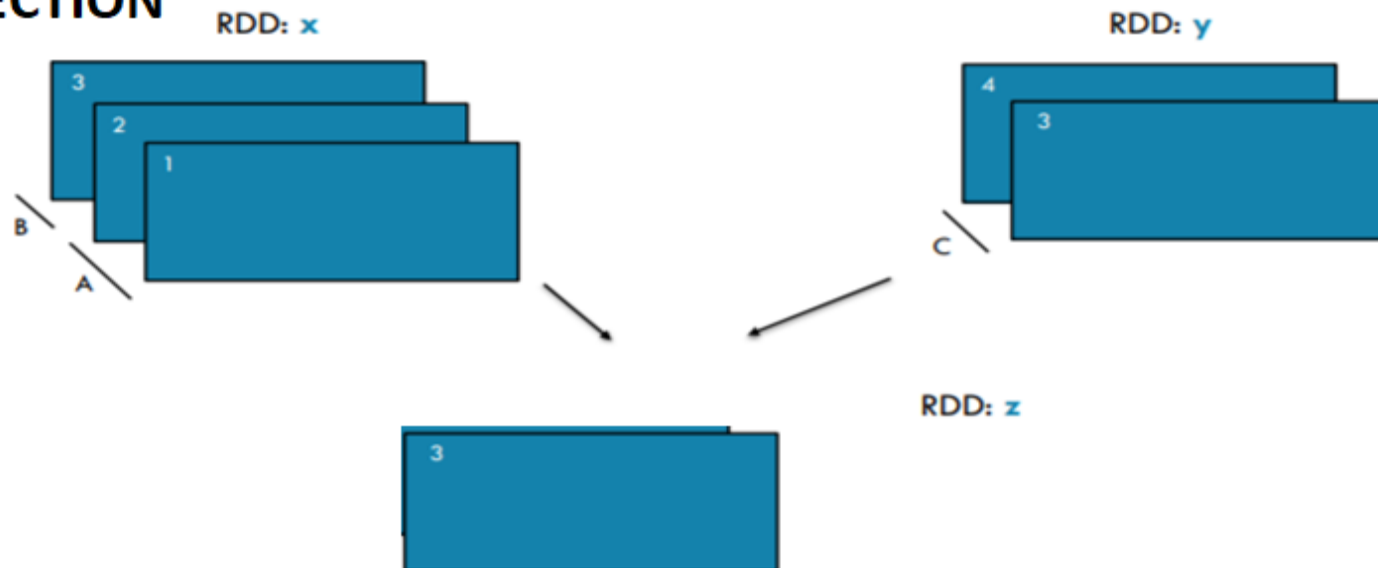
► (2) Spark Jobs

[3, 4, 5]

[[3], [4], [5]]

Command took 1.18 seconds -- by pysparktelugu@gmail.com at 10/25/2020, 2:13:55 PM on datacluster

INTERSECTION

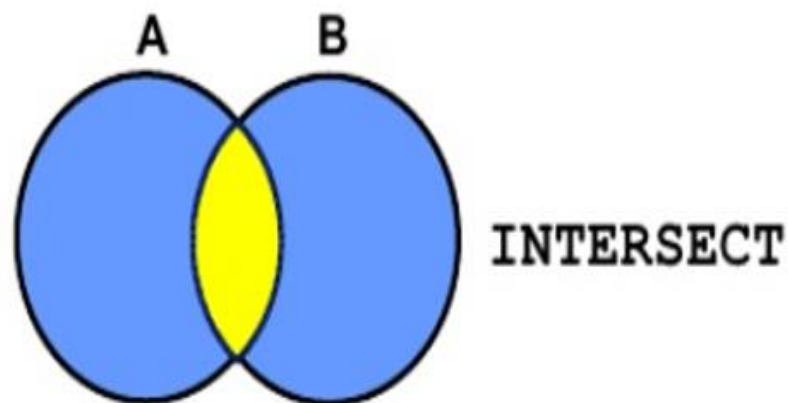


```
1 x = sc.parallelize([1,2,3], 2)
2 y = sc.parallelize([3,4], 1)
3 z = x.intersection(y)
4 print(z.collect())
5 print(z.glom().collect())
```

► (2) Spark Jobs

[3]

[[3], [], []]



subtract(otherdataset) Transformation

- It returns an RDD that has only value present in the first RDD and not in second RDD.
- its returns if first RDD is having any duplicates. its wont remove any duplicate

Cmd 46

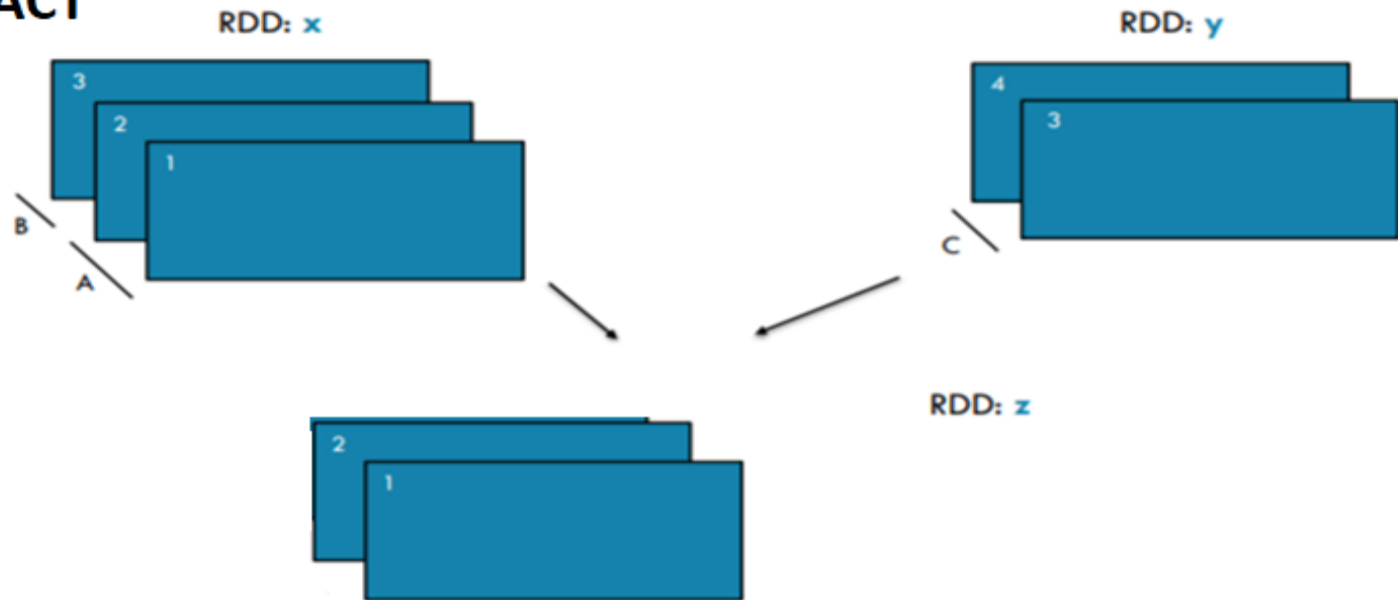
```
1 x = sc.parallelize([1,2,2,3,3,4,5,7,8], 2)
2 y = sc.parallelize([3,4,3,4,5,6], 1)
3 z = x.subtract(y)
4 print(z.collect())
5 print(z.glom().collect())
```

► (2) Spark Jobs

[1, 7, 2, 2, 8]

[[], [1, 7], [2, 2, 8]]

SUBTRACT

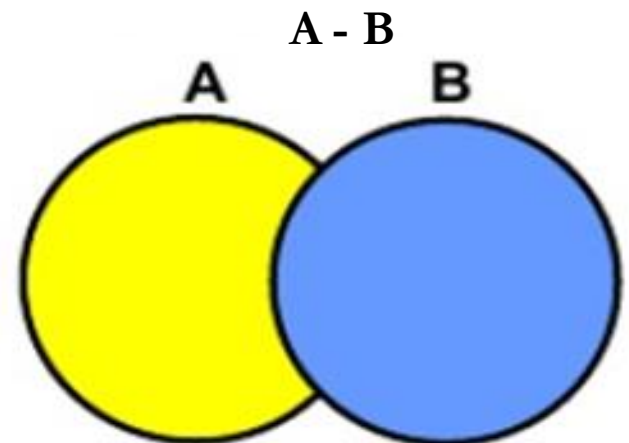


```
1 x = sc.parallelize([1,2,3], 2)
2 y = sc.parallelize([3,4], 1)
3 z = x.subtract(y)
4 print(z.collect())
5 print(z.glom().collect())
```

► (2) Spark Jobs

[1, 2]

[[], [1], [2]]

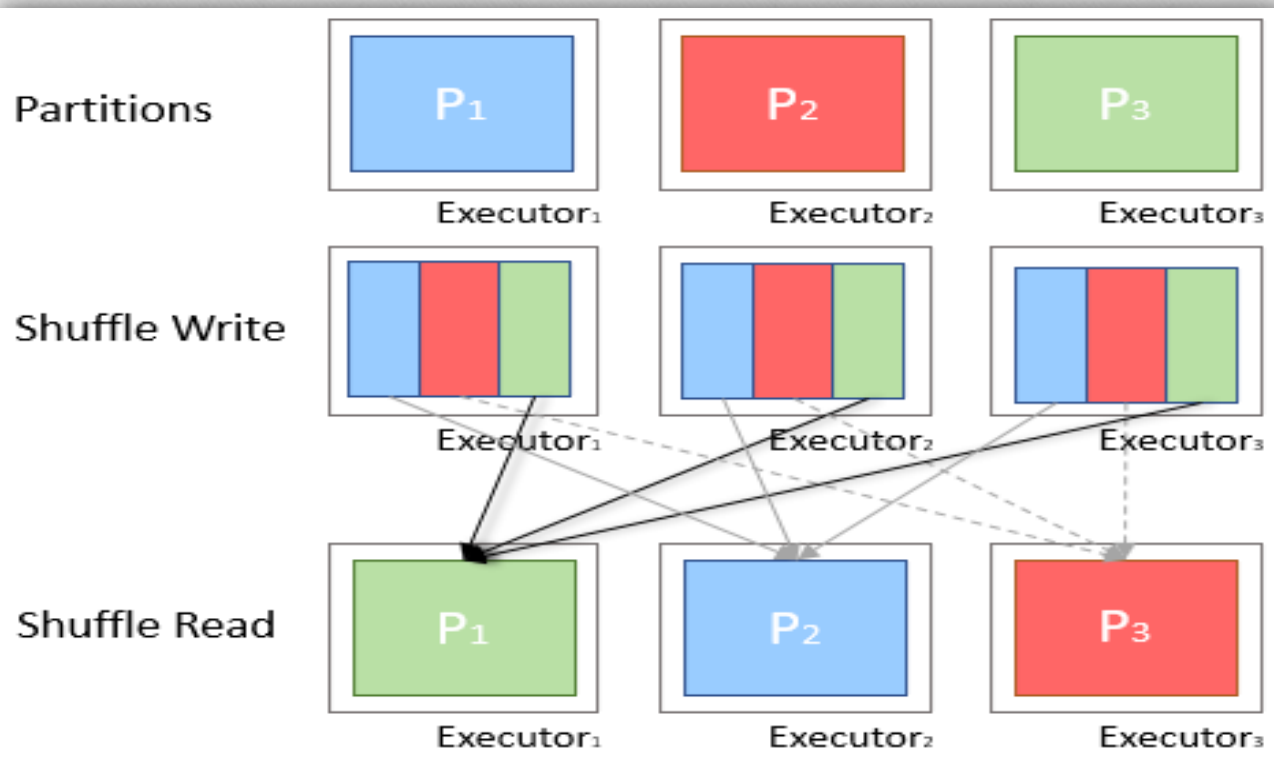


What is Shuffling?

A shuffle occurs when data is rearranged between partitions. This is required when a transformation requires information from other partitions, such as summing all the values in a column. Spark will gather the required data from each partition and combine it into a new partition, likely on a different executor. During a shuffle, data is written to disk and transferred across the network, halting Spark's ability to do processing in-memory and causing a performance bottleneck. Consequently we want to try to reduce the number of shuffles being done or reduce the amount of data being shuffled.

Shuffle read: Total shuffle bytes and records read, includes both data read locally and data read from remote executors

Shuffle write: Bytes and records written to memory(disk) in order to be read by a shuffle in a future stage



Distinct Transformation

- Return a new RDD containing distinct items from the original RDD (omitting all duplicates)

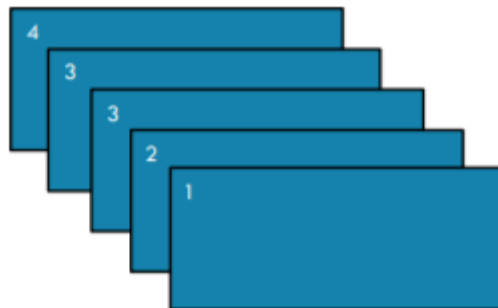
Cmd 48

```
1 x = sc.parallelize([1,2,3,3,4])
2 y = x.distinct()
3 print(y.collect())
4 |
```

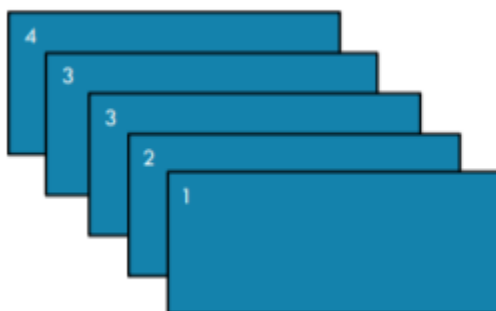
► (1) Spark Jobs

[1, 2, 3, 4]

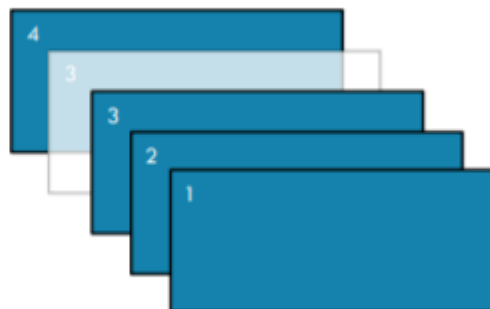
RDD: x



RDD: x



RDD: y



Coalesce Transformation

- Return a new RDD which is reduced to a smaller number of partitions

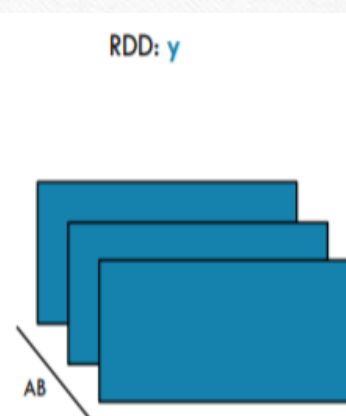
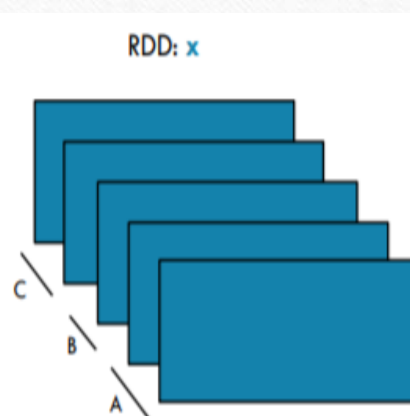
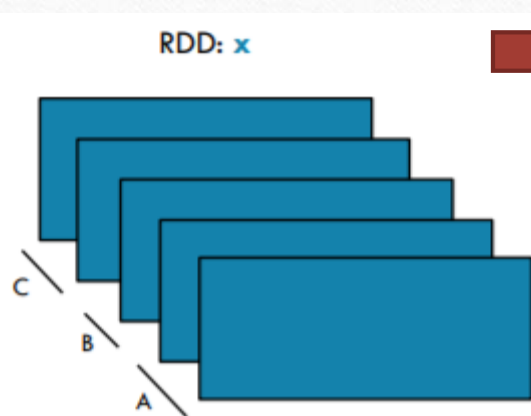
Cmd 52

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

► (2) Spark Jobs

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

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

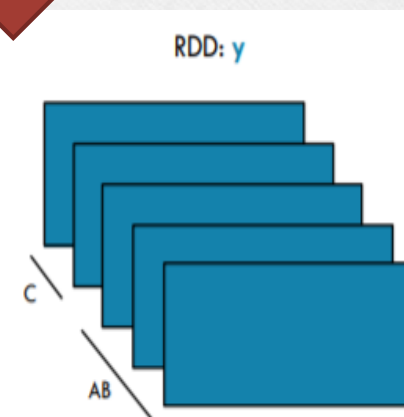
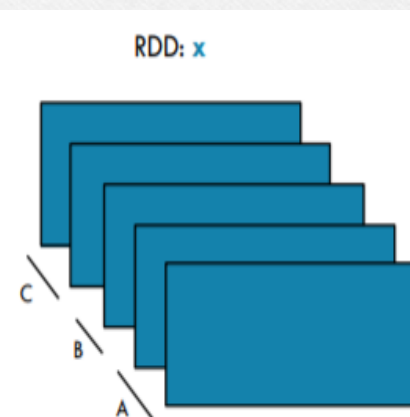


```
1 x = sc.parallelize([1, 2, 3, 4, 5], 3)
2 y = x.coalesce(2)
3 print(x.glom().collect())
4 print(y.glom().collect())
```

► (2) Spark Jobs

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

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



repartition(numPartitions) Transformation

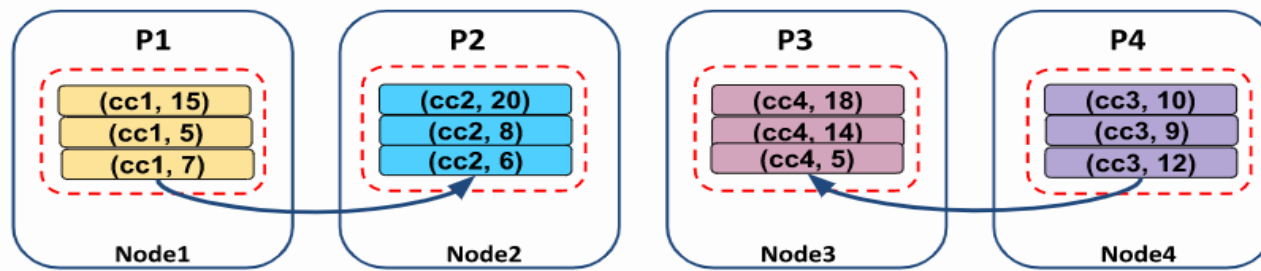
- Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them.
- This always shuffles all data over the network. Repartition by column We can also repartition by columns.
- syntax: `repartition(numPartitions, *cols)`

Cmd 54

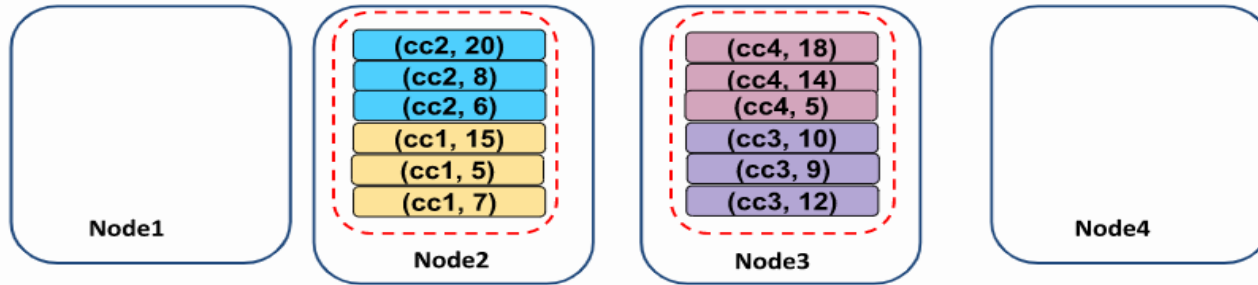
```
1 x = sc.parallelize([1, 2, 3, 4, 5,6,7,8,9,10,11,12,13,14,15], 3)
2 y = x.repartition(6)
3 print(x.glom().collect())
4 print(y.glom().collect())
5 print(y.getNumPartitions())
```

► (2) Spark Jobs

```
[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
[[], [1, 2, 3, 4, 5], [], [], [11, 12, 13, 14, 15], [6, 7, 8, 9, 10]]
6
```

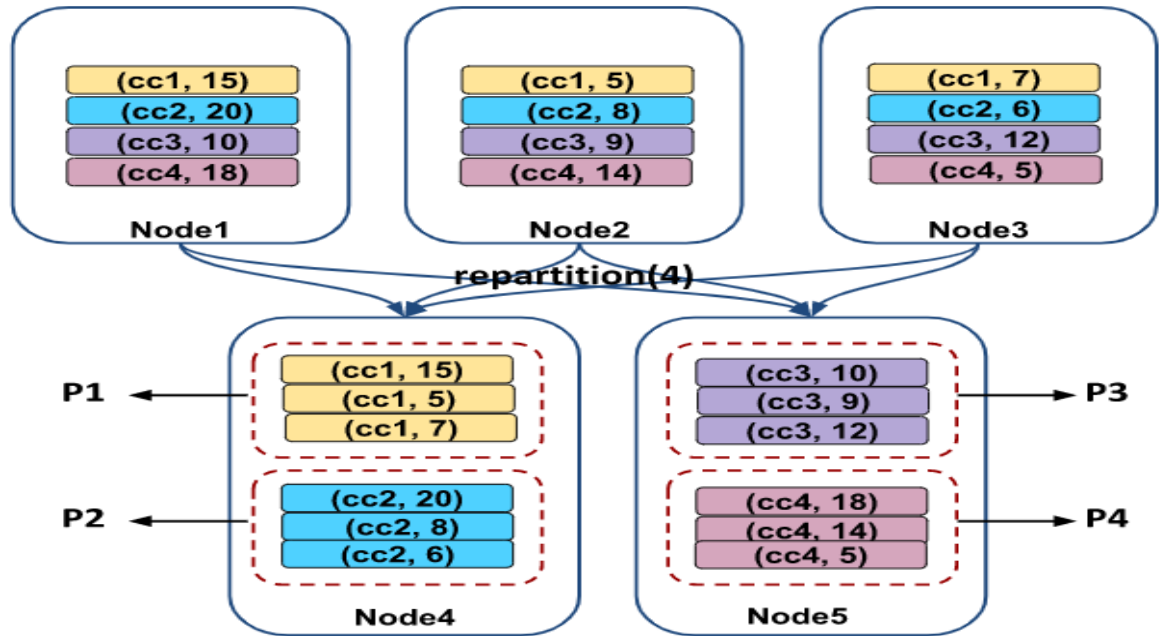


coalesce(2)



COALESCE() Just adjust the data in existing partitions

REPARTITION() will delete existing partitions And it will recreate all partitions



PartitionBy Transformation

- Return a new RDD with the specified number of partitions,
- placing original items into the partition returned by a user supplied function

Cmd 56



```
1 x = sc.parallelize([('J','James'),('F','Fred'),('A','Anna'),('J','John']], 3)
2 y = x.partitionBy(2, lambda w: 0 if w[0] < 'H' else 1)
3 print (x.glom().collect())
4 print (y.glom().collect())
5 print('X RDD No.OF Partitones : ',x.getNumPartitions())
6 print('Y RDD No.OF Partitones : ',y.getNumPartitions())
```

► (2) Spark Jobs

```
[('J', 'James')], [('F', 'Fred')], [('A', 'Anna'), ('J', 'John')]]
```

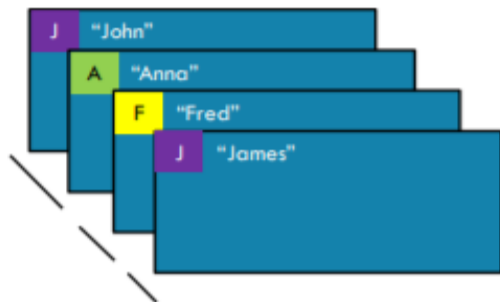
```
[('F', 'Fred'), ('A', 'Anna')], [('J', 'James'), ('J', 'John')]]
```

```
X RDD No.OF Partitones : 3
```

```
Y RDD No.OF Partitones : 2
```

Command took 0.41 seconds -- by pysparktelugu@gmail.com at 10/25/2020, 2:50:31 PM on datacluster

RDD: x



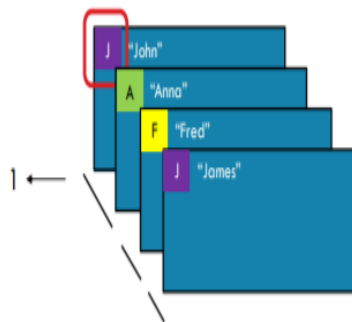
RDD: x



RDD: y



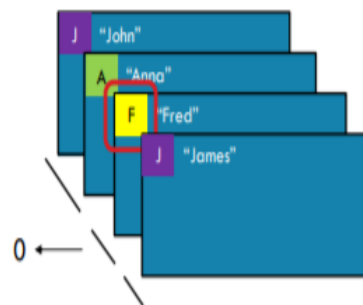
RDD: x



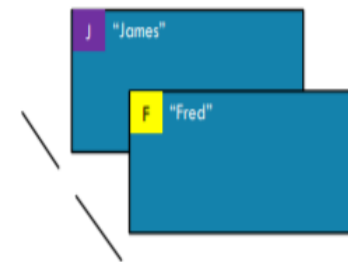
RDD: y



RDD: x



RDD: y



ZIP Transformation

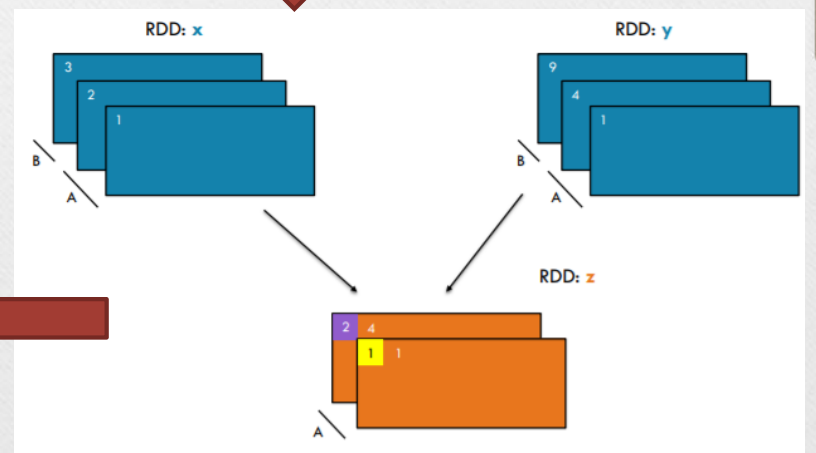
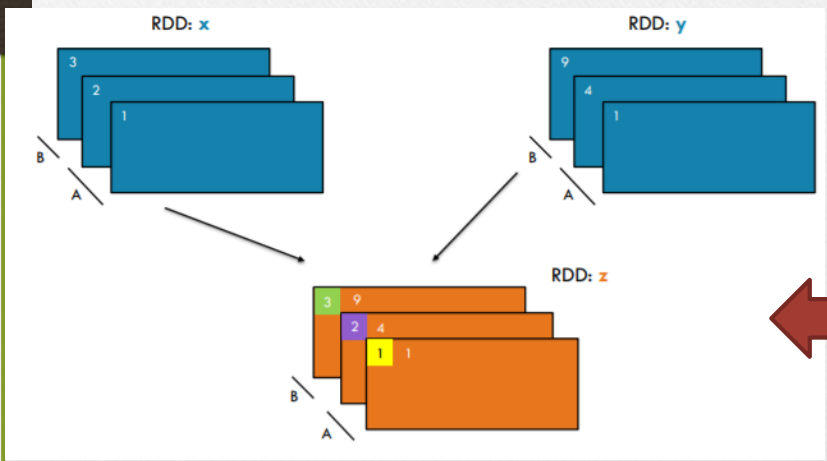
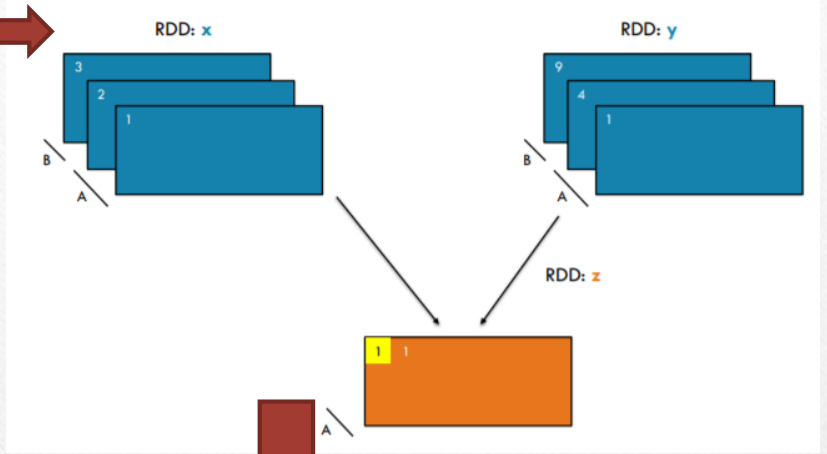
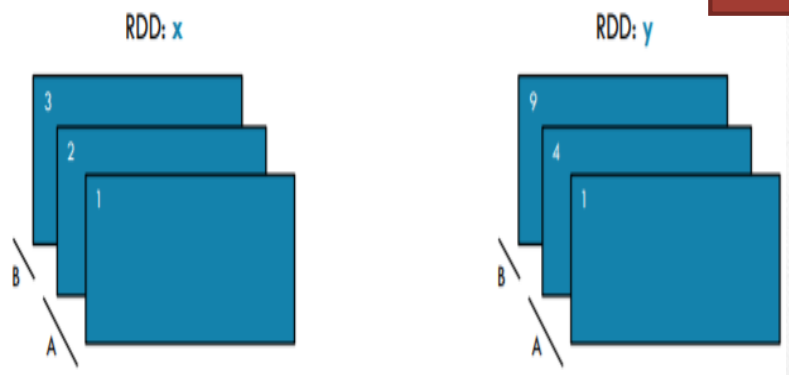
- Return a new RDD containing pairs whose key is the item in the original RDD, and whose
- value is that item's corresponding element (same partition, same index) in a second RDD

Cmd 58

```
1 x = sc.parallelize([1, 2, 3])
2 y = x.map(lambda n:n*n)
3 z = x.zip(y)
4 print(z.collect())
```

► (1) Spark Jobs

[(1, 1), (2, 4), (3, 9)]



groupByKey Transformation in RDD

* When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.

* Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance.

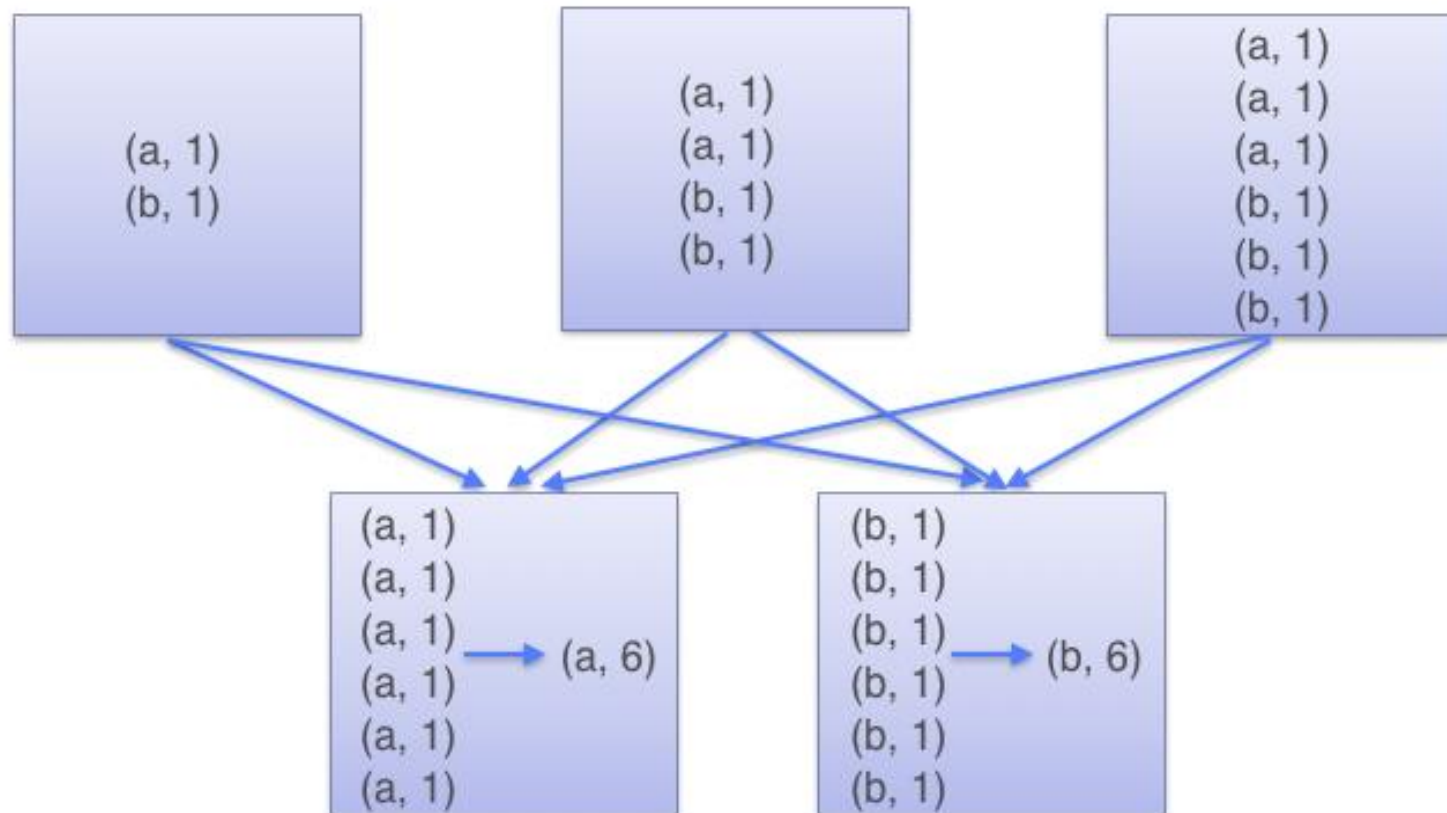
* Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.

```
1 x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
2 y = x.groupByKey()
3 print(x.collect())
4 print(list((j[0], list(j[1])) for j in y.collect()))
```

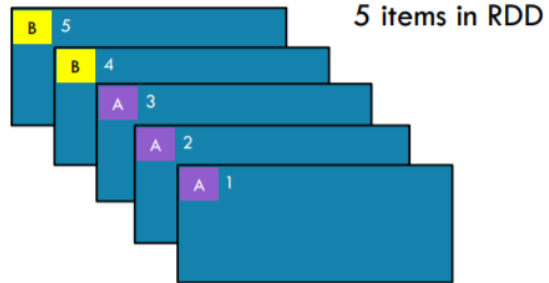
► (2) Spark Jobs

```
[('B', 5), ('B', 4), ('A', 3), ('A', 2), ('A', 1)]
[('B', [5, 4]), ('A', [3, 2, 1])]
```

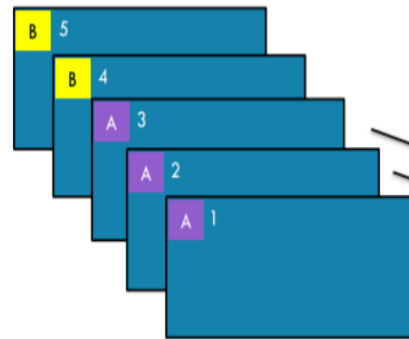
GroupByKey



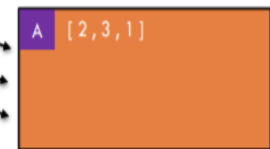
Pair RDD: x



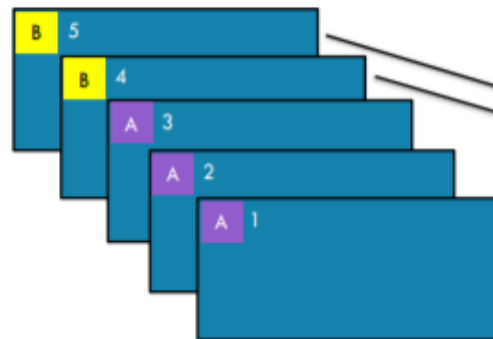
Pair RDD: x



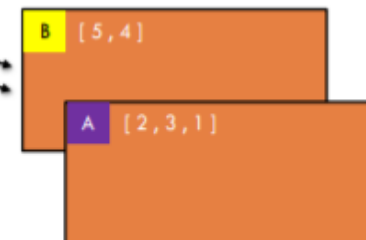
RDD: y



Pair RDD: x



RDD: y



reduceByKey(func, [numPartitions])

- When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
- NOTE: Note If you are grouping using (groupbykey) in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will provide much better performance.

Cmd 2



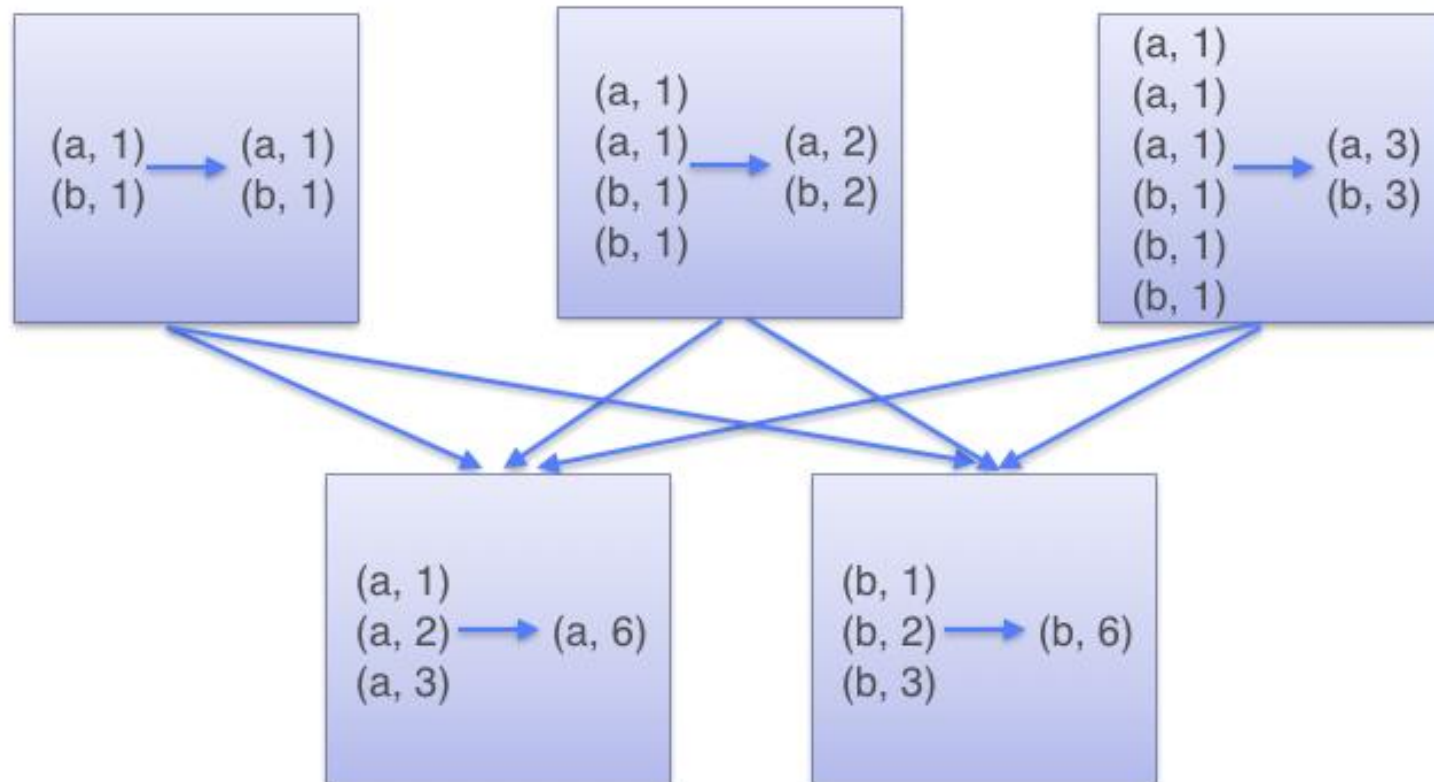
```
1 wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
2 wordsRDD = sc.parallelize(wordsList, 4)
3 wordCountsCollected = (wordsRDD
4                         .map(lambda w: (w, 1))
5                         .reduceByKey(lambda x,y: x+y)
6                         .collect())
7 print(wordCountsCollected)
```

► (1) Spark Jobs

```
[('cat', 2), ('elephant', 1), ('rat', 2)]
```

Command took 0.77 seconds -- by pysparktelugu@gmail.com at 10/26/2020, 10:39:25 PM on datacluster

ReduceByKey



ACTIONS

Cmd 70

reduce(func) Action

- Aggregate the elements of the dataset using a function func (which takes two arguments and returns one).
- The function should be commutative and associative so that it can be computed correctly in parallel.

Cmd 71

```
1 # reduce numbers 1 to 10 by adding them up
2 x = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
3 y = x.reduce(lambda a,b: a+b)
4 print(x.collect())
5 print(y)
6
```

► (2) Spark Jobs

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

55

Command took 0.57 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 7:03:25 PM on datacluster

count()

- Return the number of elements in the dataset.

Cmd 73

```
1 x = sc.parallelize([1,2,3,4])
2 x.count()
```

► (1) Spark Jobs

Out[72]: 4

Command took 0.18 seconds -- by pysparktelugu@gmail.com at 10/27/202

Cmd 74

first() Action

- Return the first element of the dataset (similar to take(1)).

Cmd 75

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

► (2) Spark Jobs

Out[73]: 1

takeSample(withReplacement, num, [seed])

- Return an array with a random sample of num elements of the dataset, with or without replacement,
- optionally pre-specifying a random number generator seed.
- Return a fixed-size sampled subset of this RDD
- withReplacement whether sampling is done with replacement
- num size of the returned sample
- seed seed for the random number generator
- returns sample of specified size in an array

Cmd 79

```
1 rdd = sc.parallelize(range(0, 10))
2 print(rdd.takeSample(True, 20, 1))
3 print(rdd.takeSample(False, 20, 1))
4
5 print(rdd.takeSample(False, 5, 2))
6
7 print(rdd.takeSample(False, 8, 5))
8
```

► (8) Spark Jobs

```
[3, 9, 3, 1, 0, 0, 6, 8, 3, 6, 9, 9, 2, 1, 0, 1, 2, 2, 4, 9]
[6, 8, 9, 7, 5, 3, 0, 4, 1, 2]
[5, 9, 3, 4, 6]
[2, 3, 1, 0, 8, 7, 6, 5]
```

take(n) Action

- Return an array with the first n elements of the dataset.

Cmd 79

```
1 x = sc.parallelize([1,2,3,4])
2 x.take(2)
```

► (2) Spark Jobs

Out[82]: [1, 2]

Command took 0.35 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 7:06:58 PM on datacluster

Cmd 80

countByValue(self)

- Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

Cmd 81

```
1 x = sc.parallelize([1, 2, 1, 2, 2])
2 y = x.countByValue().items()
3 print(y)
```

► (1) Spark Jobs

dict_items([(1, 2), (2, 3)])

isEmpty()

- Returns true if and only if the RDD contains no elements at all.
- note:: an RDD may be empty even when it has at least 1 partition

Cmd 83

```
1 x = sc.parallelize(range(10))
2 print(x.isEmpty())
3 y = sc.parallelize(range(0))
4 print(y.isEmpty())
```

► (4) Spark Jobs

False

True

Command took 0.41 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 6:01:30 PM on data

Cmd 84

keys()

- Return an RDD with the keys of each tuple.

Cmd 85

```
1 x = sc.parallelize([(1, 2), (3, 4)])
2 y = x.keys()
3 print(y.collect())
```

► (1) Spark Jobs

[1, 3]

saveAsTextFile(path, compressionCodecClass=None)

- Save the RDD to the filesystem indicated in the path

Cmd 87

```
1 dbutils.fs.rm("dbfs:/tmp/test_data/", True)
2 x = sc.parallelize([2,4,1])
3 x.saveAsTextFile("dbfs:/tmp/test_data/saveAs")
4 y = sc.textFile("dbfs:/tmp/test_data/saveAs")
5 print(y.collect())
```

► (2) Spark Jobs

['2', '4', '1']

saveAsPickleFile(self, path, batchSize=10)

- Save this RDD as a SequenceFile of serialized objects. The serializer
- used is :class: `pyspark.serializers.PickleSerializer` , default batch size is 10.

Cmd 89

```
1 dbutils.fs.rm("dbfs:/tmp/test_data/picklefile/",True)
2 x = sc.parallelize([2,4,1])
3 x.saveAsPickleFile("dbfs:/tmp/test_data/picklefile")
4 y = sc.pickleFile("dbfs:/tmp/test_data/picklefile")
5 print(y.collect())
6
7
```

► (2) Spark Jobs

[2, 4, 1]

Command took 2.36 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 6:01:30 PM on datacluster

STDEV()

- Return the standard deviation of the items in the RDD

Cmd 92

```
1 x = sc.parallelize([2,4,1])
2 y = x.stdev()
3 print(x.collect())
4 print(y)
```

► (2) Spark Jobs

[2, 4, 1]

1.247219128924647

MIN()

- Return the MIN value of the items in the RDD

Cmd 94

```
1 x = sc.parallelize([2,4,1,3,5,6,7,-5,-8])
2 y = x.min()
3 print(x.collect())
4 print(y)
```

► (2) Spark Jobs

[2, 4, 1, 3, 5, 6, 7, -5, -8]

-8

MAX()

- REturn the MAX Value of the items in the RDD

Cmd 96

```
1 x = sc.parallelize([2,4,1])
2 y = x.max()
3 print(x.collect())
4 print(y)
```

► (2) Spark Jobs

[2, 4, 1]

4

MEAN()

- Return the mean of the items in the RDD

Cmd 98

```
1 x = sc.parallelize([2,4,1])
2 y = x.mean()
3 print(x.collect())
4 print(y)
```

► (2) Spark Jobs

[2, 4, 1]

2.3333333333333335

Command took 0.20 seconds -- by pysparktelugu@gmail.com at 10/27/2020,

SUM()

- Return the Sum of the items in the RDD

Cmd 100

```
1 x = sc.parallelize([2,4,1])
2 y = x.sum()
3 print(x.collect())
4 print(y)
```

► (2) Spark Jobs

[2, 4, 1]

7

Stats()

- Return a StatCounter object that captures the mean, variance and count of the RDD's elements in one operation.

Cmd 101

```
1 list_rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
2 list_rdd.stats()
```

► (1) Spark Jobs

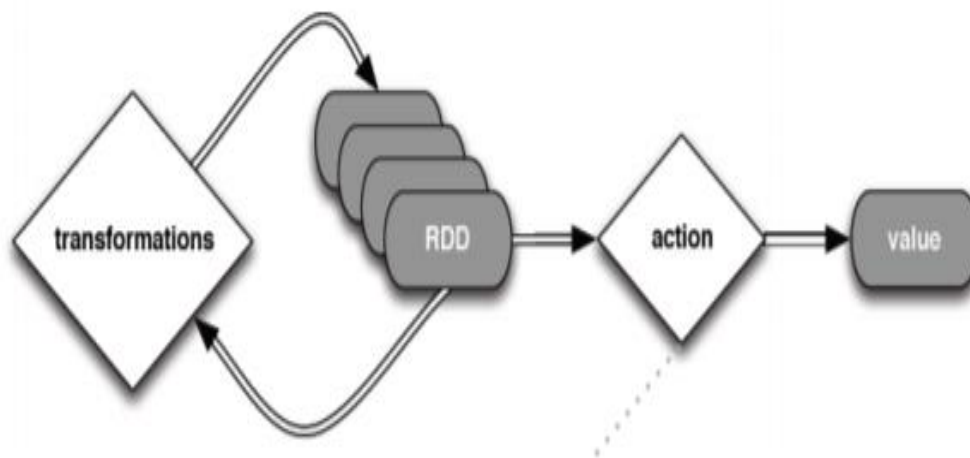
Out[90]: (count: 15, mean: 8.0, stdev: 4.320493798938574, max: 15.0, min: 1.0)

Command took 0.51 seconds -- by pysparktelugu@gmail.com at 10/27/2020, 7:36:50 PM on datacluster

Spark Core

RDD – Resilient Distributed Dataset

- A primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel.
- Two Types
 - Parallelized Scala collections
 - Hadoop datasets
- Transformations and Actions can be performed on RDDs.




Transformations

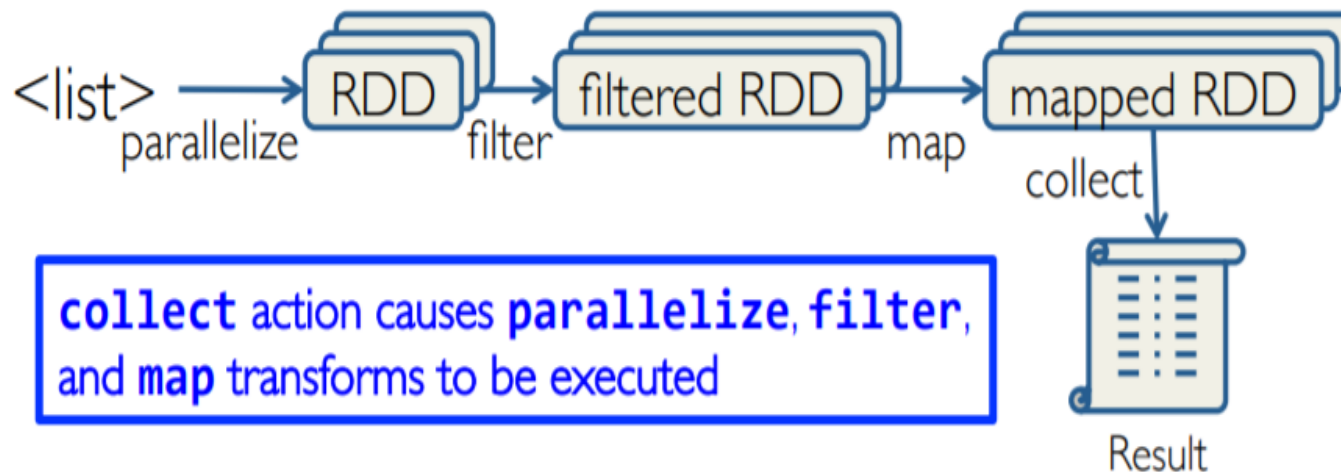
- Operate on an RDD and return a new RDD.
- Are Lazily Evaluated

Actions

- Return a value after running a computation on a RDD.
- The DAG is evaluated only when an action takes place.

Working with RDDs

- Create an RDD from a data source:  <list>
- Apply transformations to an RDD: map filter
- Apply actions to an RDD: collect count



Creating an RDD

- Create RDDs from Python collections (lists)

```
>>> data = [1, 2, 3, 4, 5]
```

```
>>> data
```

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

```
>>> rDD = sc.parallelize(data, 4)
```

```
>>> rDD
```

```
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

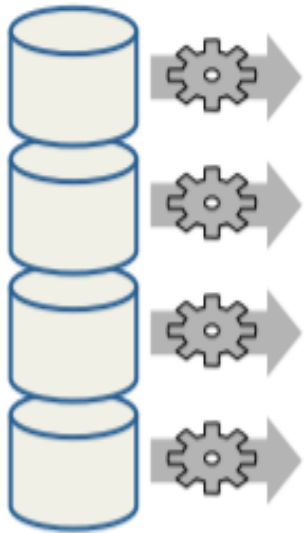
No computation occurs with `sc.parallelize()`

- Spark only records how to create the RDD with four partitions



Creating an RDD from a File

```
distFile = sc.textFile("...", 4)
```



- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now

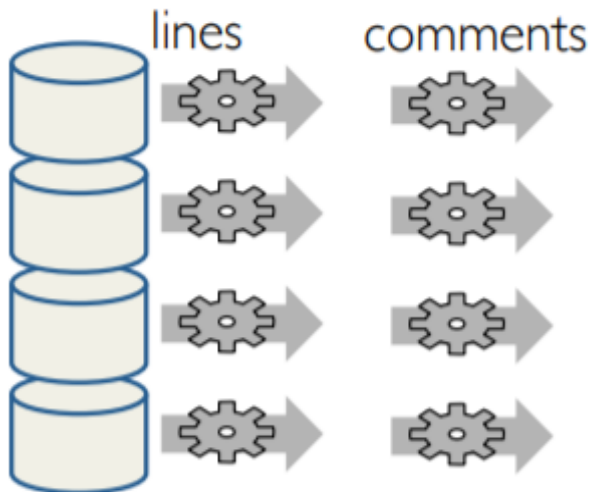
Spark Transformations

- Create new datasets from an existing one
- Use *lazy evaluation*: results not computed right away – instead Spark remembers set of transformations applied to base dataset
 - » Spark optimizes the required calculations
 - » Spark recovers from failures and slow workers
- Think of this as a recipe for creating result

Transforming an RDD

```
lines = sc.textFile("...", 4)
```

```
comments = lines.filter(isComment)
```

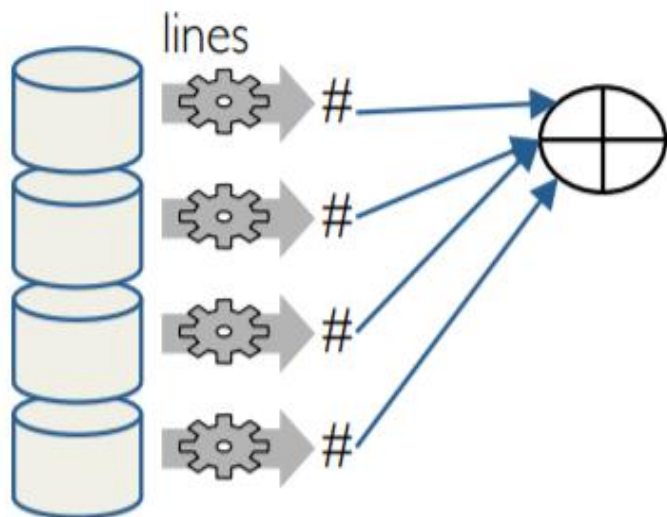


Lazy evaluation means
nothing executes –
Spark saves recipe for
transforming source

Spark Programming Model

```
lines = sc.textFile("...", 4)
```

```
print lines.count()
```

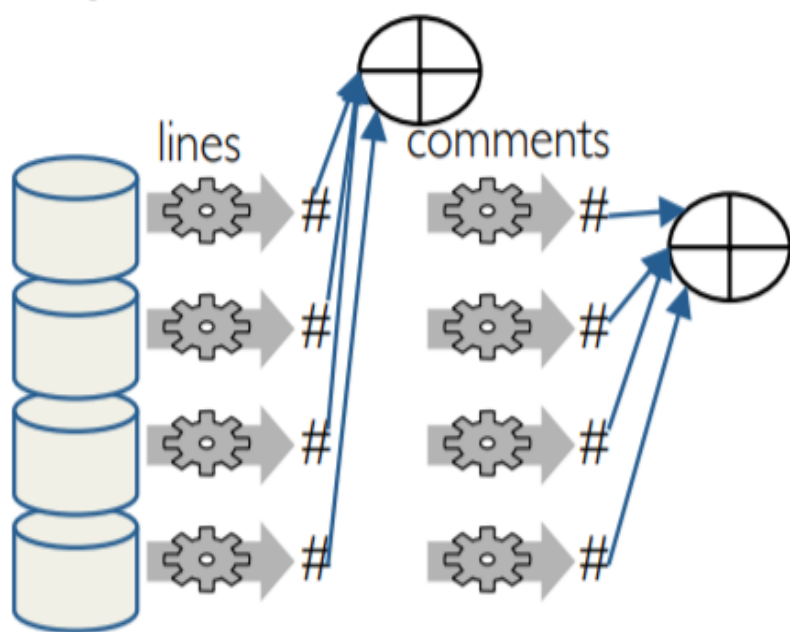


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```



Spark recomputes **lines**:

- read data (again)
- sum within partitions
- combine sums in driver

Spark Program Lifecycle

1. Create RDDs from external data or parallelize a collection in your driver program
2. Lazily transform them into new RDDs
3. **cache()** some RDDs for reuse
4. Perform actions to execute parallel computation and produce results

Spark Caching & Persistence

Spark caching can be used to pull data sets into a cluster-wide in-memory cache. This is very useful for accessing repeated data, such as querying a small “hot” dataset or when running an iterative algorithm

There are two ways to persist RDDs in Spark:

1. `cache()`
2. `persist()`

There are some advantages of RDD caching and persistence mechanism in spark.

- Time efficient
- Cost efficient
- Lesser the execution time.

STORAGE TYPES:

- 1) `MEMORY_ONLY`
- 2) `MEMORY_AND_DISK`
- 3) `DISK_ONLY`
- 4) `MEMORY_ONLY_SER`
- 5) `MEMORY_AND_DISK_SER`

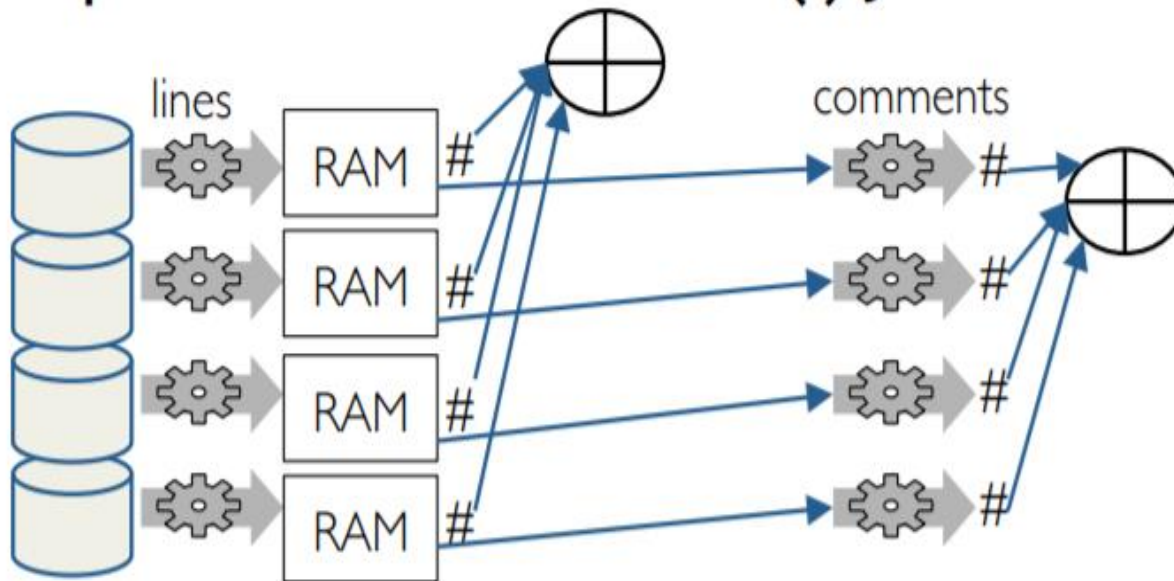
Caching RDDs

```
lines = sc.textFile("...", 4)
```

```
lines.cache() # save, don't recompute!
```

```
comments = lines.filter(isComment)
```

```
print lines.count(), comments.count()
```



Spark Caching & Persistence

RDD Unpersist

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (**LRU**) fashion.

If you would like to manually remove an RDD instead of waiting for it to fall out of the cache,
use the **RDD.unpersist()** method.

Differences Between RDD & DataFrame & DataSet

RDD Features:-

- 1) Distributed collection
- 2) Immutable
- 3) Fault tolerant
- 4) Lazy evaluations
- 5) Functional transformations => Transformations and Actions
- 6) Data processing formats => structured as well as unstructured data
- 7) Programming Languages supported => Java, Scala, Python and R.

DataFrame Features:

- 1) Distributed collection of Row Object
- 2) Data Processing
- 3) Optimization using catalyst optimizer
- 4) Hive Compatibility
- 5) Tungsten
- 6) Programming Languages supported => Java, Scala, Python and R.

DataSet Features:

- 1) Provides best of both RDD and Dataframe
- 2) Encoders
- 3) Programming Languages supported => Java, Scala
- 4) Type Safety

Disadvantages Of RDD & DATAFRAME & DATASET

Disadvantages of RDDs

If you choose to work with RDD you will have to optimize each and every RDD. In addition, unlike Datasets and DataFrames, RDDs don't infer the schema of the data ingested therefore you will have to specify it.

Disadvantages of DataFrames

The main drawback of DataFrame API is that it does not support compile time safety, as a result, the user is limited in case the structure of the data is not known.

Disadvantages of DataSets

The main disadvantage of datasets is that they require typecasting into strings.

Creating RDDs

There are many ways to create RDD objects:

1. From list or arrays defined within the program
2. By reading from normal files
3. Reading from Hadoop HDFS
4. From the output of Hive queries
5. From the output of normal databases queries

Help() function: The `help()` function is used to display the documentation string and also facilitates you to see the help related to modules, keywords, attributes, etc.

Dir() function: The `dir()` function is used to display the defined methods.

All The Best ☺

Thank you

