Session 18: INTRODUCTION TO SPARK
Assignment 1

- Prachi Mohite

Some Brief About Spark

- Apache Spark is a lightning-fast unified analytics engine for big data and machine learning.
- a powerful open-source unified analytics engine built around speed, ease of use, and streaming analytics.
- Engineered from the bottom-up for performance, Spark can be 100x faster than Hadoop for large scale data processing by exploiting in memory computing and other optimizations.
- Spark comes packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing. These standard libraries increase developer productivity and can be seamlessly combined to create complex workflows.

Spark RDD

- **RDD (Resilient Distributed Dataset)** is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is logically partitioned
    - Decomposing the name RDD:
    - Resilient, i.e. fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
    - Distributed, since Data resides on multiple nodes.
    - Dataset represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Solution Approach 1:

We have to create RDD's to perform the above Task. There are different ways of creating RDD's as below

Parallelized collection (parallelizing)

This method is used in the initial stage of learning Spark since it quickly creates our own RDDs in Spark shell and performs operations on them. Parallelizing already existing collection in driver program.

External Datasets (Referencing a dataset)

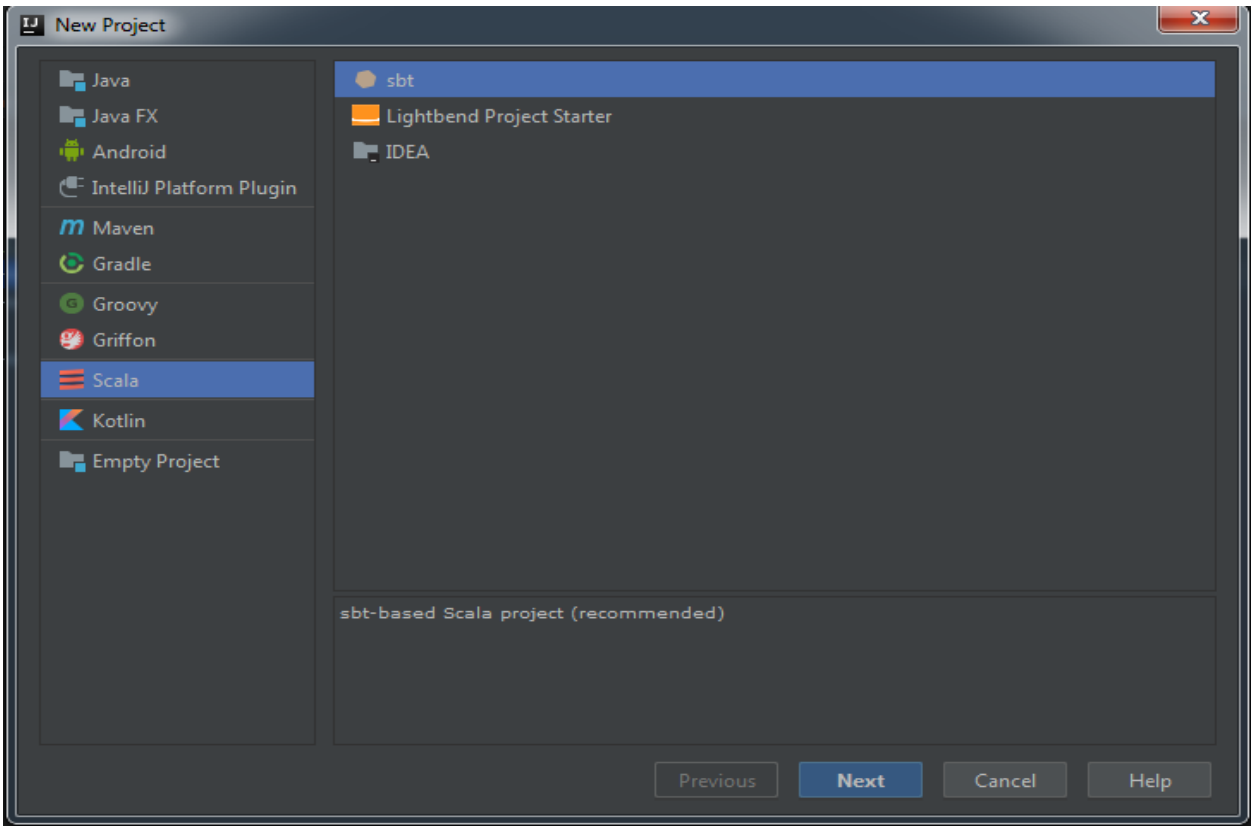
In Spark, distributed dataset can be formed from any data source supported by Hadoop, including the local file system, HDFS, Cassandra, HBase etc. In this, the data is loaded from the external dataset. To create text file RDD, we can use SparkContext's textFile method.

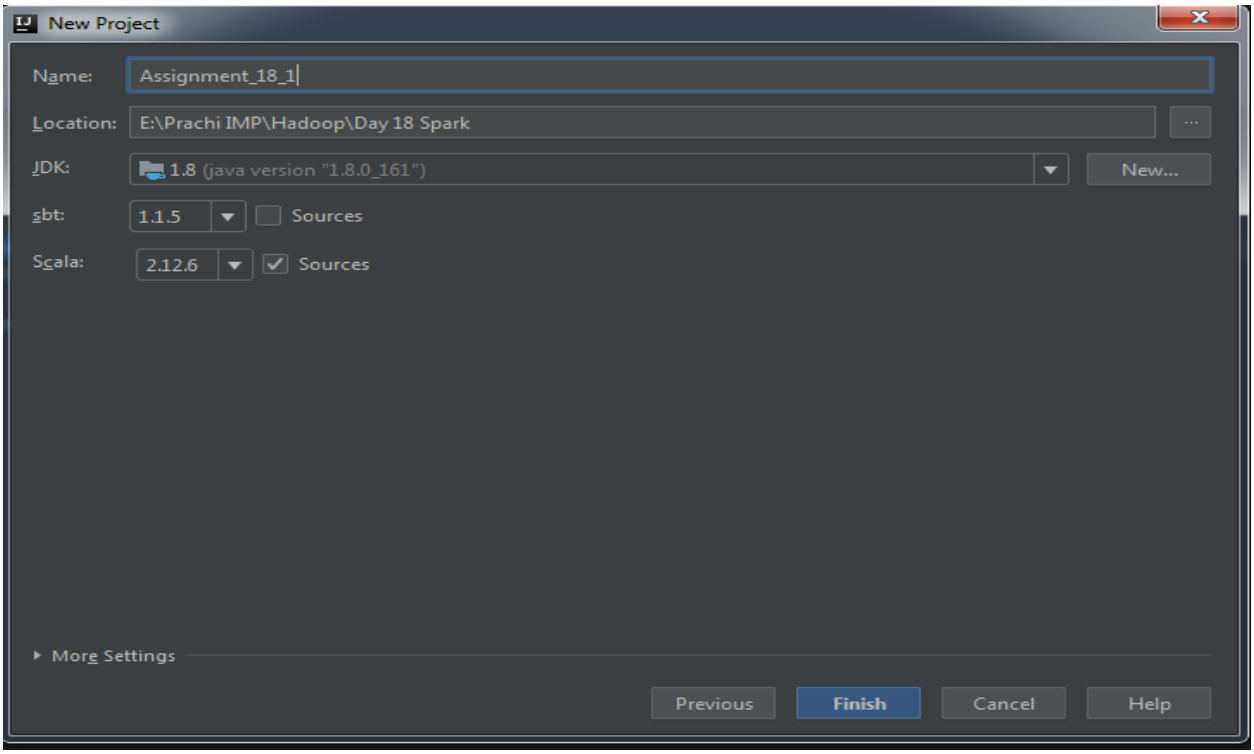
Creating RDD from existing RDD

Transformation mutates one RDD into another RDD, thus transformation is the way to create an RDD from already existing RDD.

In our example we will treat the given list as existing Dataset in our driver code and we will use first method – Using parallelize() method to create RDD.

We have created Project in IntelliJ IDEA by installing scala plugin with sbt projects.


New Project
Java
Java FX
Android
IntelliJ Platform Plugin
Maven
Gradle
Groovy
Griffon
Scala
Kotlin
Empty Project
sbt
Lightbend Project Starter
IDEA
sbt-based Scala project (recommended)
Previous
Next
Cancel
Help


New Project
Name:
Assignment_18_1
Location:
E:\Prachi IMP\Hadoop\Day 18 Spark
JDK:
1.8 (java version "1.8.0_161")
New...
sbt:
1.1.5
Sources
Scala:
2.12.6
Sources
More Settings
Previous
Finish
Cancel
Help

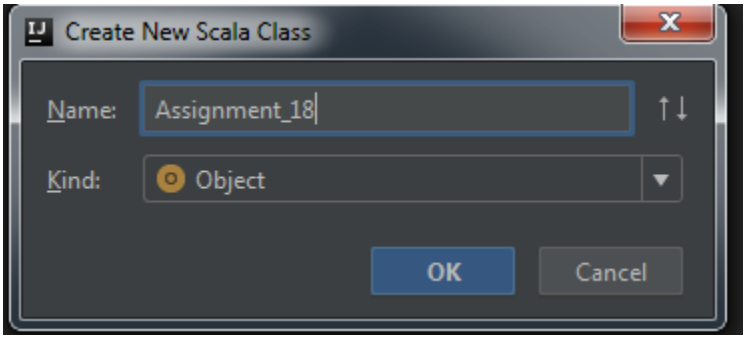
Where required external jar files for Spark are automatically downloaded from Maven Repository by adding below lines to sbt Project

```
name := "Project1"

version := "0.1"

scalaVersion := "2.11.7"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
```

Added new object to project as below



//create spark object in the project

```
    //Create spark object
val spark = SparkSession
  .builder()
  .master( master = "local")
  .appName( name = "Spark Basic Example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()
```

Solution Approach 2

Above can be achived through Spark Shell as well

Make sure Spark shell is running

```
[acadgild@localhost ~]$ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/05/21 17:38:43 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
18/05/21 17:38:44 WARN util.Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.0.4 instead (on interface eth12)
18/05/21 17:38:44 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Spark context Web UI available at http://192.168.0.4:4040
Spark context available as 'sc' (master = local[*], app id = local-1526904531479).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_151)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

- Create RDD by using parallelize method

```
// Craete RDD for the given collection by using the spark context parallelize method

val objList = spark.sparkContext.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
```

On Shell

Task 1

1.1 Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Explanation :

1. Use sum method of RDD to get sum of all elements of a List **OR**
2. We can use reduce action to get sum of the all the elements

Code
Sum Method

```
//Task 1.1 Find the sum Of all the numbers
val sumOfAll = objList.sum()
println("Sum Of all the Numbers from the List: "+ sumOfAll)
```

Reduce Action

```
val sumUsingReduce = objList.reduce(_+_)
println("Sum Of all the Numbers from the List (Reduce): "+ sumUsingReduce)
```

Output

```
18/05/22 11:21:23 INFO DAGScheduler: ResultStage 0 (sum at Assignment_18.scala:20) finished in 0.220
18/05/22 11:21:23 INFO DAGScheduler: Job 0 finished: sum at Assignment_18.scala:20, took 1.163127 s
Sum Of all the Numbers from the List: 55.0
18/05/22 11:21:23 INFO SparkContext: Invoking stop() from shutdown hook
18/05/22 11:21:23 INFO SparkUI: Stopped Spark web UI at http://169.254.26.246:4040
18/05/22 11:21:23 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
```

Using Reduce Method

```
18/05/22 15:58:48 INFO DAGScheduler: ResultStage 1 (reduce at Assignment_18.scala:23) finished in 0.019 s
18/05/22 15:58:48 INFO DAGScheduler: Job 1 finished: reduce at Assignment_18.scala:23, took 0.033345 s
Sum Of all the Numbers from the List (Reduce): 55
18/05/22 15:58:48 INFO SparkContext: Starting job: count at Assignment_18.scala:27
18/05/22 15:58:48 INFO DAGScheduler: Got job 2 (count at Assignment_18.scala:27) with 1 output partitions
18/05/22 15:58:48 INFO DAGScheduler: Final stage: ResultStage 2 (count at Assignment_18.scala:27)
```

1.2 find the total elements in the list

Explanation: Used count action to get count of all elements belonging to list

Code

```
  //Task 1.2 find the total elements in the list

  val totalElementCount = objList.count()
  println("Total element count from the List: "+ totalElementCount)
}
```

Output

```
18/05/22 11:23:27 INFO Executor: Running task 0.0 in stage 1.0 (TID 1)
Total element count from the List: 10
18/05/22 11:23:27 INFO Executor: Finished task 0.0 in stage 1.0 (TID 1). 794 by
18/05/22 11:23:27 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 1) i
```

1.3 calculate the average of the numbers in the list

we have already calculated total and total elements , so average should be total / count

Code

```
//1.3 calculate the average of the numbers in the list
val average = sumOfAll / totalElementCount
println("The average of the numbers in the list: "+ average)
}
```

Output

```
The average of the numbers in the list: 5.5
18/05/22 11:27:32 INFO SparkContext: Invoking stop() from shutdown hook
18/05/22 11:27:32 INFO SparkUI: Stopped Spark web UI at http://169.254.26.246:4040
18/05/22 11:27:32 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
18/05/22 11:27:32 INFO MemoryStore: MemoryStore cleared
```

1.4 find the sum of all the even numbers in the list

Code

```
//Task 1.4 find the sum of all the even numbers in the list
//Iterate through the list and write lambda expression to get even numbers and then add those
val sumOfEven = objList.filter(f=>f%2==0).sum()
println("Sum Of all the Even Numbers from the List: "+ sumOfEven)
}
```

Output

```
18/05/22 11:30:44 INFO TaskSetManager: Starting task 0.0 in stage 2.
18/05/22 11:30:44 INFO Executor: Running task 0.0 in stage 2.0 (TID
18/05/22 11:30:44 INFO Executor: Finished task 0.0 in stage 2.0 (TID
Sum Of all the Even Numbers from the List: 30.0
18/05/22 11:30:44 INFO TaskSetManager: Finished task 0.0 in stage 2.
18/05/22 11:30:44 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose
```

1.5 find the total number of elements in the list divisible by both 5 and 3

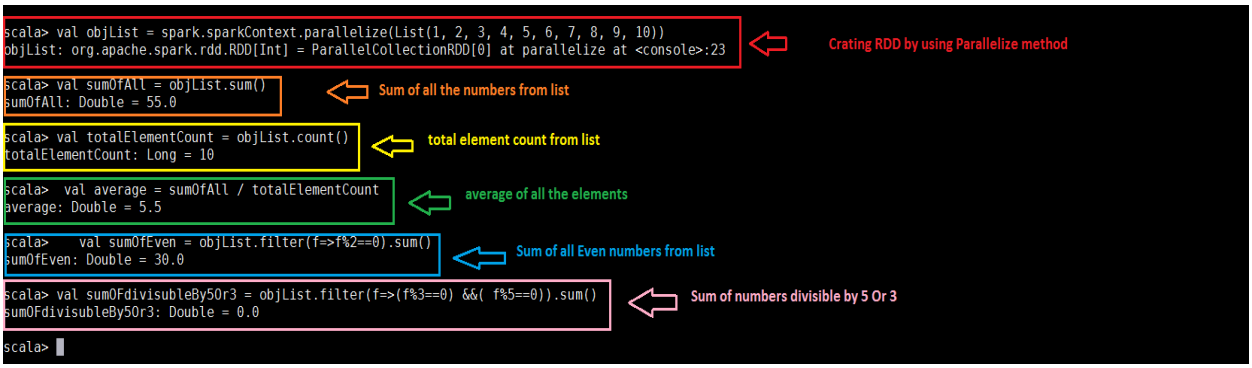
Code

```
//Task 1.5 find the total number of elements in the list divisible by both 5 and 3
//Iterate through the list and write lambda expression to get a number which is divisible by 5 and 3
val sumOFdivisubleBy5Or3 = objList.filter(f=>(f%3==0) &&( f%5==0)).sum()
println("Sum Of all the Numbers divisible by 5 Or 3 from the List: "+ sumOFdivisubleBy5Or3)
}
```

Output

```
18/05/22 11:34:39 INFO TaskSchedulerImpl: Removed TaskSet 3.0, whose tasks have
18/05/22 11:34:39 INFO DAGScheduler: ResultStage 3 (sum at Assignment_18.scala:
18/05/22 11:34:39 INFO DAGScheduler: Job 3 finished: sum at Assignment_18.scala
Sum Of all the Numbers divisible by 5 Or 3 from the List: 0.0
18/05/22 11:34:39 INFO SparkContext: Invoking stop() from shutdown hook
```

The above all 5 tasks can be executed from spark shell as well. Below are the commands and there respective output

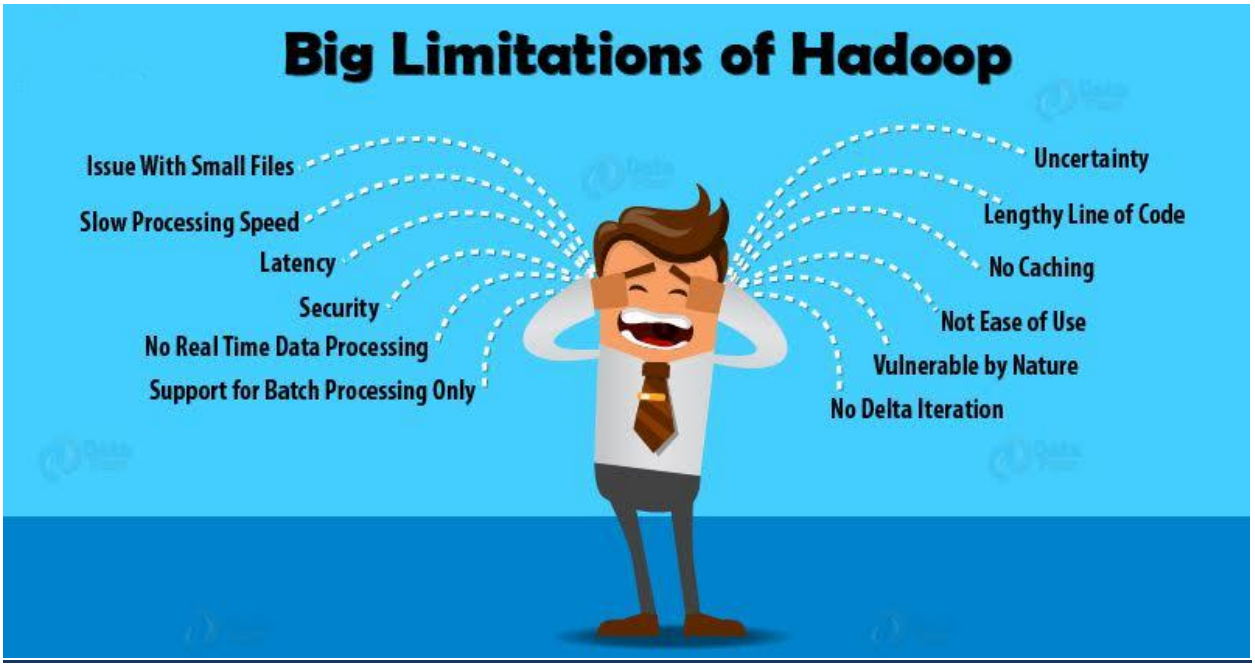


Task 2

2.1 Pen down the limitations of MapReduce.

MapReduce cannot handle:

1. Interactive Processing
2. Real-time (stream) Processing
3. Iterative (delta) Processing
4. In-memory Processing
5. Graph Processing



Below are some details regarding the issues / limitations of Map Reduce

1. **Issues with Small Files**

- **Hadoop** is not suited for small data. **(HDFS) Hadoop distributed file system** lacks the ability to efficiently support the random reading of small files because of its high capacity design

2. **Slow Processing Speed**

- In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: Map and Reduce and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

3. **Support for Batch Processing only**

- Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum.

4. **No Real-time Data Processing**

- Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

5. **No Delta Iteration**

- Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

6. **High Latency**

- In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In MapReduce, Map takes a set of data and converts it into another set of data, where individual element are broken down into key value pair and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

7. **Not Easy to Use**

- In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as hive and pig makes working with MapReduce a little easier for adopters.

8. **No Caching**

- Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

9. **Security**

- Hadoop can be challenging in managing the complex application. If the user doesn't know how to enable platform who is managing the platform, your data could be at huge risk. At storage and network levels, Hadoop is missing encryption, which is a major point of concern. Hadoop supports **Kerberos authentication**, which is hard to manage.

10. **No Abstraction**

- Hadoop does not have any type of abstraction so MapReduce developers need to hand code for each and every operation which makes it very difficult to work.

11. **Vulnerable by Nature**

- Hadoop is entirely written in **java**, a language most widely used, hence java been most heavily exploited by cyber criminals and as a result, implicated in numerous security breaches.

12. Lengthy Line of Code

- Hadoop has 1,20,000 line of code, the number of lines produces the number of bugs and it will take more time to execute the program.

2.2 What is RDD? Explain few features of RDD?

RDD (Resilient Distributed Dataset) is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is **logically partitioned** across many servers so that they can be computed on different nodes of the cluster.

Decomposing the name RDD:

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed**, since Data resides on multiple nodes.
- **Dataset** represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Features Of RDD

1. **In-memory Computation**
   Spark RDDs have a provision of **in-memory computation**. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).
2. **Lazy Evaluations**
   All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.
   Spark computes transformations when an action requires a result for the driver program.
3. **Fault Tolerance**
   Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself.
4. **Immutability**
   Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

5. **Partitioning**
   Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

6. **Persistence**
   Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

7. **Coarse-grained Operations**
   We apply coarse-grained transformations to RDD, i.e. the operation applies to the whole dataset not on an individual element in the data set of RDD.

8. **Parallel**
   RDD, process the data in parallel over the cluster.

9. **Location-Stickiness**
   RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The DAG Scheduler places the partitions in such a way that task is close to data as much as possible. Thus speed up the computation.

10. **Typed**
    We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

11. **No limitation**
    we can have any number of RDD. there is no limit to its number, but depends on the size of disk and memory.

2.3 List down few Spark RDD operations and explain each of them.

RDD in Apache Spark supports two types of operations:

- Transformation
- Actions

Transformations

Spark RDD Transformations are functions that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey() etc.

Transformations are lazy operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

- **Narrow Transformations**

  It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

- **Wide Transformations**

  It is the result of groupByKey() and reduceByKey() like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.

Actions

An Action in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

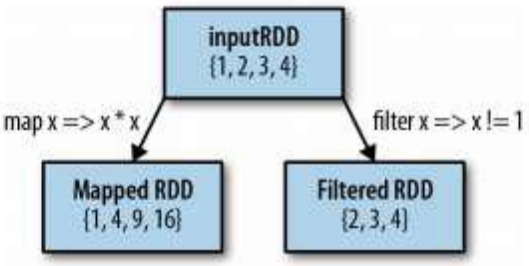
1. map(func)

The map function iterates over every line in RDD and split into new RDD.
Using map() transformation we take in any function, and that function is applied to every element of RDD. In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.

For Eg: we need to square all the values in the list
val input = sc.parallelize(List(1, 2, 3, 4))

val result = input.map(x => x * x)

println(result.collect().mkString(","))

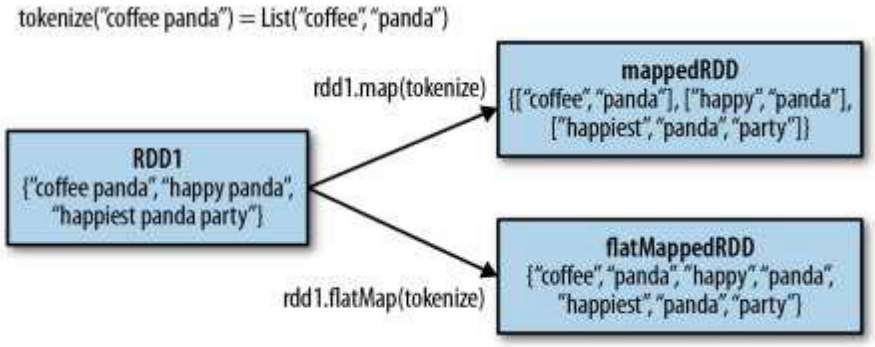


2. Flatmap()

With the help of flatMap() function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words. Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

flatMap() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val flatmapFile = data.flatMap(lines => lines.split(" "))
flatmapFile.foreach(println)
```



3. filter(func)

Spark RDD filter() function returns a new RDD, containing only the elements that meet a predicate. It does not shuffle data from one partition to many partitions.

Filter() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())
```

Note – In above code, flatMap function map line into words and then count the word "Spark" using count() Action after filtering lines containing "Spark" from mapFile.

4. mapPartitions(func)

The MapPartition converts each partition of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

5. mapPartitionWithIndex()

It is like mapPartition; Besides mapPartition it provides function with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.

Function name	Purpose	Example	Result
union()	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract()	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), … (3,5)}

6. union(dataset)

With the union() function, we get the elements of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.
Union() example:
val rdd1 =
spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))
val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
val rddUnion = rdd1.union(rdd2).union(rdd3)
rddUnion.foreach(Println)
Note – In above code union() operation will return a new dataset that contains the union of the elements in the source dataset (rdd1) and the argument (rdd2 & rdd3).

7. intersection(other-dataset)

With the intersection() function, we get only the common element of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Intersection() example:
val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014,
(16,"feb",2014)))

```
val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
val comman = rdd1.intersection(rdd2)
comman.foreach(Println)
```

Note – The intersection() operation return a new RDD. It contains the intersection of elements in the rdd1 & rdd2.

8. distinct()

It returns a new dataset that contains the distinct elements of the source dataset. It is helpful to remove duplicate data.

Distinct() example:

```
val rdd1 =
park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2
014)))
val result = rdd1.distinct()
println(result.collect().mkString(", "))
```

Note – In the above example, the distinct function will remove the duplicate record i.e. (3,'”nov”,2014).

9. groupByKey()

When we use groupByKey() on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD. In this transformation, lots of unnecessary data get to transfer over the network.
Spark provides the provision to save data to disk when there is more data shuffled onto a single executor machine than can fit in memory.

groupByKey() example:

```
val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)
```

Note – The groupByKey() will group the integers on the basis of same key(alphabet). After that collect() action will return all the elements of the dataset as an Array.

10. reduceByKey(func, [numTasks])

When we use reduceByKey on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.

reduceByKey() example:

```
val x = sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),("a", 1), ("b", 1),("b", 1),("b", 1), ("b", 1)))
val y = x.reduceByKey((key, value) => (key + value))
y.collect()
```

Note – The above code will parallelize the Array of String. It will then map each letter with count 1, then reduceByKey will merge the count of values having the similar key.

11. sortByKey()

When we apply the sortByKey() function on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.

sortByKey() example:

```
val data = spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82), ("computer",65), ("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)
```

Note – In above code, sortByKey() transformation sort the data RDD into Ascending order of the Key(String).

12. join()

The Join is database term. It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.

Join() example:

```
val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))
```

RDD Action

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

---

1 count()

Action count() returns the number of elements in RDD.

Count() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())
```

Note – In above code flatMap() function maps line into words and count the word "Spark" using count() Action after filtering lines containing "Spark" from mapFile.

2 collect()

The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result. Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

Collect() example:

```
val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))
```

Note – join() transformation in above code will join two RDDs on the basis of same key(alphabet). After that collect() action will return all the elements to the dataset as an Array.

3 countByValue()

The countByValue() returns, many times each element occur in RDD.

countByValue() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val result= data.map(line => (line,line.length)).countByValue()
result.foreach(println)
```

Note – The countByValue() action will return a (K, Int) pairs with the count of each key.

Function name	Purpose	Example	Result
`take(num)`	Return num elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top num elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return num elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return num elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., sum).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

4. take(n)

The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.

Take() example:

```
val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
val twoRec = result.take(2)
twoRec.foreach(println)
```

Note – The take(2) Action will return an array with the first n elements of the data set defined in the taking argument.

5. top()

If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top() use default ordering of data.

Top() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
val res = mapFile.top(3)
res.foreach(println)
```

Note – map() operation will map each line with its length. And top(3) will return 3 records from mapFile with default ordering.

6. reduce()

The reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.

Reduce() example:

```
val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
val sum = rdd1.reduce(_+_)
println(sum)
```

Note – The reduce() action in above code will add the elements of the source RDD.

7. fold()

The signature of the fold() is like reduce(). Besides, it takes “zero value” as input, which is used for the initial call on each partition. But, the condition with zero value is that it should be the identity element of that operation. The key difference between fold() and reduce() is that, reduce() throws an exception for empty collection, but fold() is defined for empty collection.

For example, zero is an identity for addition; one is identity element for multiplication. The return type of fold() is same as that of the element of RDD we are operating on.
For example, rdd.fold(0)((x, y) => x + y).

Fold() example:

```
val rdd1 = spark.sparkContext.parallelize(List(("maths", 80),("science", 90)))
val additionalMarks = ("extra", 4)
val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2
("total", add)
}
println(sum)
```

Note – In above code additionalMarks is an initial value. This value will be added to the int value of each record in the source RDD.

8. aggregate()

It gives us the flexibility to get data type different from the input type.
The aggregate() takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

9. foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the driver. In this case, foreach() function is useful. For example, inserting a record into the database.

Foreach() example:

```
val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)
```

Note – The foreach() action run a function (println) on each element of the dataset group