**Assignment 16.2**

1. Pen down the limitations of MapReduce.

**Real-time** processing.

It's not **always** very easy to implement each and everything as a MR program.

When your intermediate processes need to talk to each other(jobs run in isolation).

When your processing requires lot of data to be **shuffled** over the network.

When you need to handle streaming data. MR is best suited to **batch process** huge amounts of data

which you already have with you.

When you can get the desired result with a standalone system. It's obviously less painful to configure and manage a standalone system as compared to a distributed system.

When you have **OLTP** needs. MR is not suitable for a large number of short on-line transactions.

**3.1. Issue with Small Files**

**Hadoop** is not suited for small data. [**(HDFS)** **Hadoop distributed file system**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) lacks the ability to efficiently support the random reading of small files because of its high capacity design.

Small files are the major problem in HDFS. A small file is significantly smaller than the[**HDFS block**](http://data-flair.training/blogs/data-blocks-hdfs-hadoop-distributed-file-system/)size (default 128MB). If we are storing these huge numbers of small files, HDFS can’t handle these lots of files, as HDFS was designed to work properly with a small number of large files for storing large data sets rather than a large number of small files. If there are too many small files, then the **NameNode** will be overloaded since it stores the namespace of HDFS.

**Solution-**

* Solution to deal with small file issue is simple merge the small files to create bigger files and then copy bigger files to HDFS.
* **HAR files** (Hadoop Archives) were introduced to reduce the problem of lots files putting pressure on the namenode’s memory. By building a layered filesystem on the top of HDFS, HAR files works. Using Hadoop archive command, HAR files are created, which runs a **[MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/)** job to pack the files being archived into a small number of HDFS files. Reading through files in a HAR is not more efficient than reading through files in HDFS. Since each HAR file access requires two index files read as well the data file to read, this makes it slower.
* **Sequence files**work very well in practice to overcome the ‘small file problem’, in which we use the filename as the key and the file contents as the value. By writing a program for files (100 KB), we can put them into a single Sequence file and then we can process them in a streaming fashion operating on the Sequence file. MapReduce can break Sequence file into chunks and operate on each chunk independently because Sequence file is splittable.
* Storing files in **[HBase](http://data-flair.training/blogs/hbase-tutorial-beginners-guide/)**is a very common design pattern to overcome small file problem with HDFS. We are not actually storing millions of small files into HBase, rather adding the binary content of the file to a cell.

Refer this guide to [learn HDFS data read-write operations in detail.](http://data-flair.training/blogs/hadoop-hdfs-data-read-and-write-operations/)

**3.2. 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**](http://data-flair.training/blogs/mapper-in-hadoop-mapreduce/) and [**Reduce**](http://data-flair.training/blogs/reducer-in-hadoop-mapreduce/)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.

Refer this guide to[learn how data flows in Hadoop MapReduce.](http://data-flair.training/blogs/hadoop-mapreduce-flow-how-data-flows-in-mapreduce/)

**Solution-**

Spark has overcome this issue, by in-memory processing of data. [**In-memory processing**](http://data-flair.training/blogs/apache-spark-in-memory-computing/) is faster as no time is spent in moving the data/processes in and out of the disk. Spark is 100 times faster than MapReduce as it processes everything in memory. Flink is also used, as it processes faster than spark because of its streaming architecture and Flink may be instructed to process only the parts of the data that have actually changed, thus significantly increases the performance of the job.

Refer this guide to [learn MapReduce job optimization and performance tuning techniques.](http://data-flair.training/blogs/mapreduce-job-optimization-performance-tuning-techniques/)

**3.3. 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**](http://data-flair.training/blogs/install-hadoop-2-x-ubuntu-hadoop-multi-node-cluster/) to the maximum.

**Solution-**

Spark improves the performance, but [**Spark stream processing**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/)is not as much efficient as Flink as it uses micro-batch processing. Flink improves the overall performance as it provides single run-time for the streaming as well as batch processing. Flink uses native closed loop iteration operators which make [**machine learning**](http://data-flair.training/blogs/machine-learning-tutorial/)and graph processing faster.

**3.4. 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.

**Solution-**

* **Apache Spark** supports stream processing. Stream processing involves continuous input and output of data. It emphasizes on the velocity of the data, and data is processed within a small period of time. Learn more about [Spark Streaming APIs](http://data-flair.training/blogs/apache-spark-streaming-transformation-operations/).
* **Apache Flink** provides single run-time for the streaming as well as batch processing, so one common run-time is utilized for data streaming application and batch processing application. Flink is a stream processing system that is able to process row after row in real time.

**3.5. 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).

**Solution-**

Apache Spark can be used to overcome this issue, as it accesses data from RAM instead of disk, which dramatically improves the performance of iterative algorithms that access the same dataset repeatedly. Spark iterates its data in batches. For iterative processing in Spark, each iteration has to be scheduled and executed separately.

**3.6. 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**](http://data-flair.training/blogs/key-value-pairs-hadoop-mapreduce/) 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.

**Solution-**

Spark is used to reduce this issue, Apache spark is yet another batch system but it is relatively faster since it caches much of the input data on memory by [**RDD(Resilient Distributed Dataset)**](http://data-flair.training/blogs/rdd-in-apache-spark/)and keeps intermediate data in memory itself. Flink’s data streaming achieves low latency and high throughput.

Refer this guide to [learn how to create RDD in Apache Spark](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/).

**3.7. 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**](http://data-flair.training/blogs/apache-hive-tutorial-introductory-guide/)and[**pig**](http://data-flair.training/blogs/apache-pig-tutorial-introduction-guide/)makes working with MapReduce a little easier for adopters.

**Solution-**

While Spark can be used for such issue, Spark has interactive mode so that developers and users alike can have intermediate feedback for queries and other action. Spark is easy to program as it has tons of high-level operators. Flink can also be easily used as it also has high-level operators.

Refer this guide to[learn Apache Spark RDD Transformations and Actions API](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

**3.8. 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.

HDFS **supports access control lists** (ACLs) and a traditional file permissions model. However, third party vendors have enabled an organization to leverage**Active Directory Kerberos** and**LDAP** for authentication.

**Solution-**

Spark provides security bonus. If we run spark in HDFS, it can use HDFS ACLs and file-level permissions. Additionally, Spark can run on [**YARN**](http://data-flair.training/blogs/hadoop-yarn-tutorial/) giving it the capability of using Kerberos authentication.

**3.9. 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.

**Solution-**

To overcome this, Spark is used in which for batch we have RDD abstraction. Flink has Dataset abstraction.

**3.10. 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.

**3.11. 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.

**Solution-**

Spark and Flink can overcome this, as Spark and Flink cache data in memory for further iterations which enhance the overall performance.

**3.12. 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.

**Solution-**

Although Spark and Flink are written in[**scala**](http://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/)and java but they are implemented in Scala, so the number of line of code is lesser than Hadoop. So it will also take less time to execute the program.

To learn Scala [get Best Scala books to become a master in Scala](http://data-flair.training/blogs/best-scala-books-list/).

**3.13. Uncertainty**

Hadoop only ensures that data job is complete, but it’s unable to guarantee when the job will be complete

1. What is RDD? Explain few features of RDD?

## Apache Spark RDD

[**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) stands **Resilient Distributed Dataset**. RDDs are the fundamental abstraction of Apache Spark. It is an immutable distributed collection of the dataset. Each dataset in RDD is divided into logical partitions. On the different node of the cluster, we can compute These partitions. RDDs are a read-only partitioned collection of record. we can[create RDD in three ways](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/):

* **Parallelizing** already existing collection in driver program.
* **Referencing a dataset** in an external storage system (e.g. [HDFS](http://data-flair.training/blogs/apache-hadoop-hdfs-introduction-tutorial/), [Hbase](http://data-flair.training/blogs/hbase-tutorial-beginners-guide/), shared file system).
* Creating RDD **from already existing RDDs**.

There are two operations in RDD namely [transformation and Action](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

## Sparkling Features of Spark RDD

There are several advantages of using RDD. Some of them are-

**In-memory computation**

The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes. refer this comprehensive guide to [Learn Spark in-memory computation](http://data-flair.training/blogs/apache-spark-in-memory-computing/) in detail.

### Lazy Evaluation

The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do. Follow this guide to [learn Spark lazy evaluation in great detail](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/).

### Fault Tolerance

Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data.

### Immutability

RDDS are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

**Persistence**

We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function

### Partitioning

RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

### Parallel

Rdd, process the data parallelly over the cluster.

### Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAGScheduler**places the partitions in such a way that task is close to data as much as possible. Thus speed up computation. Follow this guide to

### Coarse-grained Operation

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

### Typed

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

**No limitation**

we can have any number of RDD. there is no limit to its number. the limit depends on the size of disk and memory.

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

### 3.1. 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 example, in RDD {1, 2, 3, 4, 5} if we apply “rdd.map(x=>x+2)” we will get the result as (3, 4, 5, 6, 7).

**Map() example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import org.apache.spark.SparkContext  import org.apache.spark.SparkConf  import org.apache.spark.sql.SparkSession  object  mapTest{  def main(args: Array[String]) = {  val spark = SparkSession.builder.appName("mapExample").master("local").getOrCreate()  val data = spark.read.textFile("spark\_test.txt").rdd  val mapFile = data.map(line => (line,line.length))  mapFile.foreach(println)  }  } |

**spark\_test.txt"**

hello...user! this file is created to check the operations of spark. [How to create RDD](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/)?, and how can we apply functions on that RDD partitions?. All this will be done through spark programming which is done with the help of scala language support…

* ***Note –***In above code, map() function map each line of the file with its length.

### 3.2. 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()](http://data-flair.training/blogs/map-vs-flatmap-operation-in-apache-spark/) is map() returns only one element, while flatMap() can return a list of elements.

**flatMap() example:**

|  |  |
| --- | --- |
| 1  2  3 | val data = spark.read.textFile("spark\_test.txt").rdd  val flatmapFile = data.flatMap(lines => lines.split(" "))  flatmapFile.foreach(println) |

* **Note –**In above code, flatMap() function splits each line when space occurs.

### 3.3. filter(func)

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

[](https://data-flair.training/blogs/category/quiz/spark-quiz/)

For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

**Filter() example:**

|  |  |
| --- | --- |
| 1  2  3 | 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.

### 3.4. 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.

### 3.5. mapPartitionWithIndex()

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

### 3.6. 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.

For example, the elements of **RDD1** are (Spark, Spark,[**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), **[Flink](http://data-flair.training/blogs/apache-flink-tutorial-comprehensive-guide/)**) and that of**RDD2** are ([**Big data**](http://data-flair.training/blogs/why-learn-big-data-use-cases/), Spark, Flink) so the resultant ***rdd1.union(rdd2)*** will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

**Union() example:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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).

### 3.7. 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.

Consider an example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant **rdd1.intersection(rdd2)**will have elements (spark).

**Intersection() example:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

### 3.8. distinct()

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

For example, if RDD has elements (Spark, Spark, Hadoop, Flink),then ***rdd.distinct()***will give elements (Spark, Hadoop, Flink).

**Distinct() example:**

|  |  |
| --- | --- |
| 1  2  3 | val rdd1 = park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))  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).

### 3.9. 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. Follow this link to [learn about RDD Caching and Persistence mechanism](http://data-flair.training/blogs/apache-spark-rdd-persistence-caching/) in detail.

**groupByKey() example:**

|  |  |
| --- | --- |
| 1  2  3 | 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.

### 3.10. 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:**

|  |  |
| --- | --- |
| 1  2  3 | val words = Array("one","two","two","four","five","six","six","eight","nine","ten")  val data = spark.sparkContext.parallelize(words).map(w => (w,1)).reduceByKey(\_+\_)  data.foreach(println) |

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

### 3.11. 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:**

|  |  |
| --- | --- |
| 1  2  3 | 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).

### 3.12. 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.

The boon of using keyed data is that we can combine the data together. The join() operation combines two data sets on the basis of the key.

**Join() example:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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*** –  The join() transformation will join two different RDDs on the basis of Key.

### 3.13. coalesce()

[](https://data-flair.training/blogs/category/interview-questions/spark-interview-questions/)

To avoid full shuffling of data we use coalesce() function. In **coalesce()** we use existing partition so that less data is shuffled. Using this we can cut the number of the partition. Suppose, we have four nodes and we want only two nodes. Then the data of extra nodes will be kept onto nodes which we kept.

**Coalesce() example:**

|  |  |
| --- | --- |
| 1  2  3 | val rdd1 = spark.sparkContext.parallelize(Array("jan","feb","mar","april","may","jun"),3)  val result = rdd1.coalesce(2)  result.foreach(println) |

* **Note –** The coalesce will decrease the number of partitions of the source RDD to numPartitions define in coalesce argument.

## 4. RDD Action

**Transformations** [**create RDDs**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

### 4.1. count()

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

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

**Count() example:**

|  |  |
| --- | --- |
| 1  2  3 | 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.

### 4.2. 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:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

### 4.3. 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.

For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

**Take() example:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

### 4.4. 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:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

### 4.5. countByValue()

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

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.countByValue()”  will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

**countByValue() example:**

|  |  |
| --- | --- |
| 1  2  3 | 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 hashmap of (K, Int) pairs with the count of each key.

### 4.6. 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:**

|  |  |
| --- | --- |
| 1  2  3 | 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.

### 4.7. 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:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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.

### 4.8. 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.

### 4.9. 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:**

|  |  |
| --- | --- |
| 1  2  3 | 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