ASSIGNMENT 16.2

Problem Statement:

1. Pen down the limitations of MapReduce.
2. What is RDD? Explain few features of RDD?
3. List down few Spark RDD operations and explain each of them.

Solution:

1. Limitations of MapReduce:
   - **MapReduce is not suitable for processing of small amount of data.** It has been designed to work on data of higher volumes typically in Giga Bytes or Terra Bytes and it lacks the ability to process smaller data files of size in few Mega Bytes.
   - **MapReduce significantly takes more time to process data.** MapReduce makes use of distributed and parallel algorithm having two important stages: Map and Reduce. It needs a lot of time to achieve these tasks which leads to latency in obtaining results. In MapReduce environment, a dataset is read from the disk storage, say HDFS and not stored in memory of processing framework. Disk read is always time consuming as it requires lookups to disk very often. **It does not make use of caching** concept which is helpful to eliminate delay in fetching huge amount of data.
   - **MapReduce does not support real time data processing.** It is only designed for batch data processing. It takes static data files of high volume, processes it at once and returns results. It cannot process data coming in a streamed form in real time which is a main reason nowadays for organizations to not choose MapReduce for data processing.
   - **MapReduce lacks strong security features to protect data.** Apart from the standard Kerberos authentication scheme which is difficult to handle, there are no encryption schemes provided by MapReduce to secure customer’s data. Hence it relies on traditional user level file permissions and access control lists (ACLs).
   - **Coding in MapReduce is not user-friendly.** MapReduce does not support interactive mode and developers need to hard code every operation which makes it more painful to work with.

- **MapReduce is tightly coupled as far as programming language is concerned.** MapReduce is written in Java with no support to other languages. Hence we mostly see a lengthy code to achieve the smallest of operations on given dataset. People who don't have exposure to Java programming find it hard to adapt and produce solutions.

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

RDD – Resilient Distributed Dataset: It is the basic unit of abstraction in Spark which is a collection of immutable objects computed over distributed nodes of a cluster. An RDD is a partitioned collection of records which can be created by applying transformation operations on existing RDD's such as map, filter, etc. The first RDD in any Spark program can be created in two ways: either by reading a text file from given file system path or by parallelizing an existing collection of data elements.

Features of RDD:

- **Resilient:** RDD's in Spark are created over a set of transformations and these are recorded in a graphical notation called 'lineage graph'. With the help of this graph we can re-compute missing or damaged partitions which might happen due to node failure.
- **Distributed:** RDD's are distributed in nature. It means they are computed by parallelizing any operation on source data over a collection of nodes in the cluster which makes its creation faster.
- **Dataset:** It is a collection of data that is partitioned with primitive values or a pair of values such as tuples or maps.
- **Lazily evaluated:** Transformations makes no sense without actions. It means transformations are not computed until there exist an action on those transformed units.
- **Immutable:** The objects inside an RDD are only readable and can be used to create another RDD by applying a transformation such as map(), etc.
- **In-memory computation:** As soon as we load a text file or a collection into Spark context, all operations related to RDD's are carried out in Spark's execution memory to increase access speed.

- **Cacheable:** This is a unique feature which allows an RDD to be cached to a persistent storage. This will be helpful when user wants perform different operations on the same RDD. It avoids re-computation of same RDD and saves time.
- **Parallel computation:** Operations on an RDD are computed over a distributed collection of nodes based on the partitions specified.
- **Location stickiness:** An RDD can define preferences on its placement to compute partitions as close to the records as possible.
- **Partitioned:** Records inside an RDD are split into logical partitions and are distributes across nodes of a cluster.
- **Typed:** RDD elements are slightly different in terms of data types compared to primitive ones. For example, Int in RDD[Int], String in RDD[String], (Int, String) in RDD[(Int, String)], etc.

3. Operations on RDD's:

Here is a list of common operations on Spark RDD's:

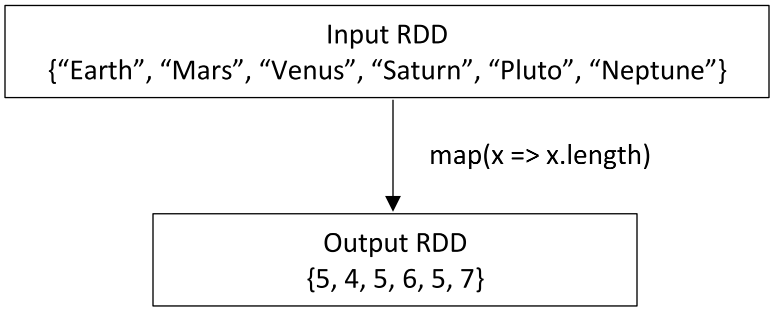
- map()
- flatMap()
- filter()
- distinct()
- collect()
- count()
- take()
- reduce()
- foreach()

Let's try to understand each of them with an example.

- **map():** The map() operation takes a function as input and applies it to each element in the RDD with the result of function being a new value of each element in the resulting RDD. The datatype of resulting need not be same as that of original RDD. If we have an RDD of strings in the form of RDD[String], we find length of each string in this RDD by applying a map() on it. The resulting RDD type will be an integer in the form of RDD[Int].

Example:

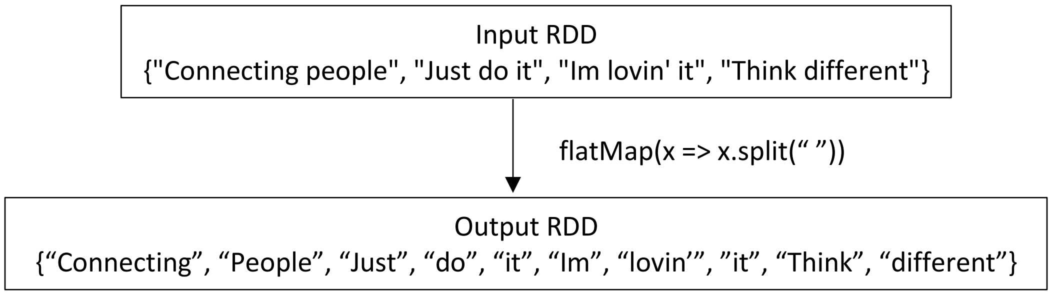
```
val sampleList = List("Earth", "Mars", "Venus", "Saturn", "Pluto", "Neptune")
val parallelized_rdd = sc.parallelize(sampleList)
val mapped_rdd = parallelized_rdd.map(x => x.length)
```



- **flatMap():** It is applied on each element of an RDD just like map(), but instead of returning a single element, flatMap() returns one or more elements for each input element. A common use case of flatMap() is to split up an input string by separator character.

  Example:

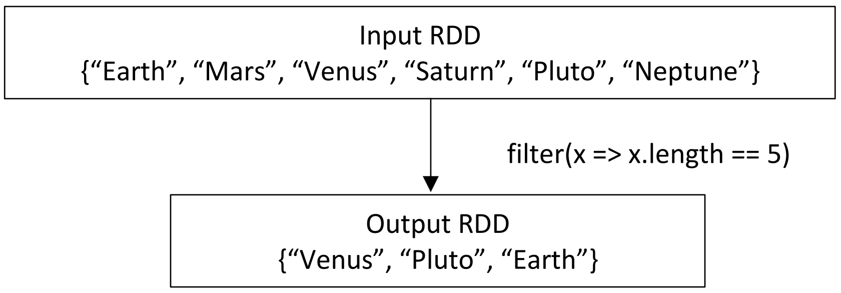
```
val sampleList = List("Connecting people", "Just do it", "Im lovin' it", "Think different")
val parallelized_rdd = sc.parallelize(sampleList)
val flatmap_rdd = parallelized_rdd.flatMap(x => x.split(" "))
```


Input RDD
{"Connecting people", "Just do it", "Im lovin' it", "Think different"}
flatMap(x => x.split(“ ”))
Output RDD
{“Connecting”, “People”, “Just”, “do”, “it”, “Im”, “lovin’”, ”it”, “Think”, “different”}

- **filter():** It takes a function and returns an RDD consisting of only those elements that satisfies the condition passed to filter().

  Example:

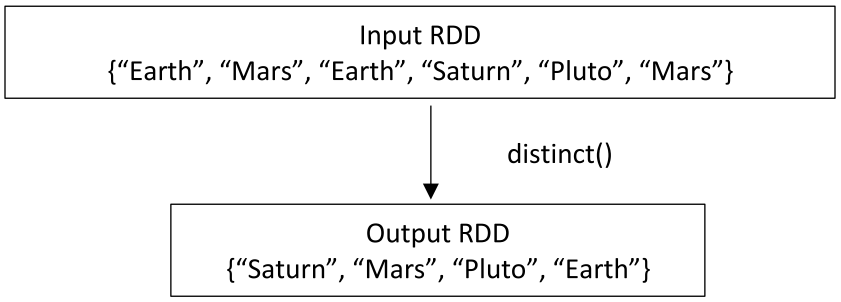
```
val sampleList = List("Earth", "Mars", "Venus", "Saturn", "Pluto", "Neptune")
val parallelized_rdd = sc.parallelize(sampleList)
val filtered_rdd = parallelized_rdd.filter(x => x.length == 5)
```


Input RDD
{“Earth”, “Mars”, “Venus”, “Saturn”, “Pluto”, “Neptune”}
filter(x => x.length == 5)
Output RDD
{“Venus”, “Pluto”, “Earth”}

- **distinct():** dictinct() produces a new RDD with only distinct elements from original RDD.

Example:

```
val sampleList = List("Earth", "Mars", "Earth", "Saturn", "Pluto", "Mars")
val parallelized_rdd = sc.parallelize(sampleList)
val distinct_rdd = parallelized_rdd.distinct()
```


Input RDD
{“Earth”, “Mars”, “Earth”, “Saturn”, “Pluto”, “Mars”}
distinct()
Output RDD
{“Saturn”, “Mars”, “Pluto”, “Earth”}

- **collect():** it returns all the contents of an RDD. It has a constraint that all data should fit in memory of a single machine which has to be then copied to the driver. This operation is mostly useful in unit tests so as to check whether entire content fits in the memory.

Example:

```
val sampleList = List("Earth", "Mars", "Earth", "Saturn", "Pluto", "Mars")
val parallelized_rdd = sc.parallelize(sampleList)
val collect_rdd = parallelized_rdd.collect()
```

- **count():** As the name indicates, it returns the count of elements in the input RDD.

Example:

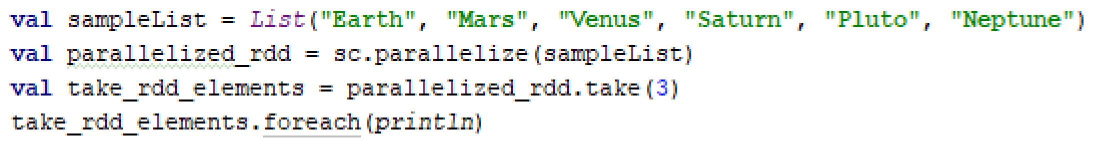
```
val sampleList = List("Earth", "Mars", "Earth", "Saturn", "Pluto", "Mars")
val parallelized_rdd = sc.parallelize(sampleList)
val count_rdd_elements = parallelized_rdd.count()
println("Count of elements in the input RDD: " + count_rdd_elements)
```

Output:

```
Count of elements in the input RDD: 6
```

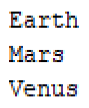
- **take(n):** It returns n elements from an input RDD**.** It also attempts to reduce the number of partitions it accesses.

Example:



```
val sampleList = List("Earth", "Mars", "Venus", "Saturn", "Pluto", "Neptune")
val parallelized_rdd = sc.parallelize(sampleList)
val take_rdd_elements = parallelized_rdd.take(3)
take_rdd_elements.foreach(println)
```

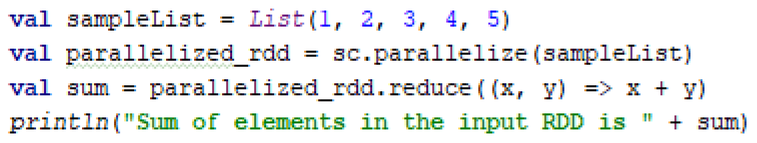
Output:



```
Earth
Mars
Venus
```

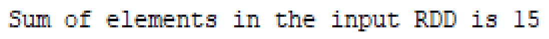
- **reduce():** It is the most common action on RDDs which takes a function as input that operates on two elements of the type in an RDD and returns a new element of the same type. Here is an example the computes the sum of elements in an RDD.

Example:



```
val sampleList = List(1, 2, 3, 4, 5)
val parallelized_rdd = sc.parallelize(sampleList)
val sum = parallelized_rdd.reduce((x, y) => x + y)
println("Sum of elements in the input RDD is " + sum)
```

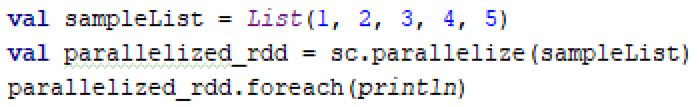
Output:



```
Sum of elements in the input RDD is 15
```

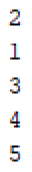
- **foreach():** This takes a function as input and applies computation specified in that function on each element of input RDD.

Example:



```
val sampleList = List(1, 2, 3, 4, 5)
val parallelized_rdd = sc.parallelize(sampleList)
parallelized_rdd.foreach(println)
```

Output:



```
2
1
3
4
5
```

In the above example, I have supplied println function as argument to foreach() to print each element of given RDD.