ASSIGNMENT 16.2

Que 1 : Pen down the limitations of MapReduce.

Ans : There are following limitations of MapReduce:

1. Since MapReduce is suitable only for batch processing jobs, implementing interactive jobs and models becomes impossible.

2. Applications that involve precomputation on the dataset brings down the advantages of MapReduce.

3. Implementing iterative map reduce jobs is expensive due to the huge space consumption by each job.

4. Problems that cannot be trivially partitionable or recombinable becomes a candid limitation of MapReduce problem solving. For instance, Travelling Salesman problem.

5. Due to the fixed cost incurred by each MapReduce job submitted, application that requires low latency time or random access to a large set of data is infeasible.

6. Also, tasks that has a dependency on each other cannot be parallelized, which is not possible through MapReduce.

Que 2 :What is RDD? Explain few features of RDD?

Ans Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs − **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

5.1. 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).

5.2. 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. Follow this guide for the deep study of **Spark Lazy Evaluation**.

5.3. 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. Follow this guide for the deep study of **RDD Fault Tolerance**.

5.4. 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.

5.5. 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.

5.6. Persistence

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

5.7. Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

5.8. 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.

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

Ans : Apache Spark RDD supports two types of Operations-

- Transformations
- Actions
  - **Spark Transformation** is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we

apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

- Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as **RDD operator graph** or **RDD dependency graph.** It is a logical execution plan i.e., it is Directed Acyclic Graph (**DAG**) of the entire parent RDDs of RDD.

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)
```

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.

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.

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**, **Flink**) and that of**RDD2** are (**Big data**, Spark, Flink) so the resultant ***rdd1.union(rdd2)*** will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

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)
```

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:

```
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)
```

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:

```
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(", "))
```