**ASSIGNMENT-16.2**

1. Pen down the limitations of MapReduce.
2. Processing speed

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. MapReduce algorithm contains two important tasks: Map and Reduce and, MapReduce require lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce.

1. Data processing

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

1. 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.

4. Ease of 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 add one such as hive and pig, make working with MapReduce a little easier for adopters.

5. Caching

In Hadoop, MapReduce cannot cache the intermediate data in-memory for a further requirement which diminishes the performance of hadoop

6. 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

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

RDD stands for “Resilient Distributed Dataset”. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node 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.

Hence, each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. It posses self-recovery in the case of failure.

**FEATURES :**

* **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).

* **Lazy Evaluation**

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.

* **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.

* **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.

* **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**

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.

* **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.

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

There are various functions in RDD transformation. Let us see RDD transformation with examples:

* **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).

### 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.

* **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.

**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.

### 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).

* **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).

* **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).