**ASSIGNMENT-16.2**

1. Pen down the limitations of MapReduce.

Various limitations of Hadoop are discussed below in this section along with their solution-

* **Issue 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.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.

### 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.

### 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.

### 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.

### 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).

### 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.

### 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.

### 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.

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

RDDs are Immutable and partitioned collection of records, which can only be created by *coarse grained operations*such as map, filter, group by etc. By coarse grained operations , it means that the operations are applied on all elements in a datasets. RDDs can only be created by reading data from a stable storage such as HDFS or by transformations on existing RDDs.

In case of we lose some partition of RDD , we can replay the transformation on that partition  in lineage to achieve the same computation, rather than doing data replication across multiple nodes.This characteristic is biggest benefit of RDD , because it saves a lot of efforts in data management and replication and thus achieves faster computations.

**FEATURES :**

* **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.

* **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.

* **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 DAGSchedulerplaces the partitions in such a way that task is close to data as much as possible. Thus speed up computation.

* **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.

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).