**Assignment 16.2:**

1. **Pen down the limitations of MapReduce.**

- MapReduce is based on disk based computing, suitable for single pass computations and not iterative computations

- It needs a sequence of MapReduce jobs to run iterative tasks

**1. Processing speed**

In [Hadoop](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), 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.

**2. Data processing**

Hadoop [MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) 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.

**3. 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](https://goo.gl/VKRPf6) 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

RDDs are Immutable and partitioned collection of records, which can only be created by coarse grained operations such as map, filter, and 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.

Since RDDs are created over a set of transformations, it logs those transformations, rather than actual data. Graph of transformations to produce one RDD is called as Lineage Graph.

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 of RDD:**

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).

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.

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.

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. 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.

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).

7. Coarse-grained Operations

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

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.

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

1. MAP:

Return a new RDD by applying a function to each element of this RDD

val x = sc.parallelize(Array("b", "a", "c"))

val y = x.map(z => (z,1))

println(x.collect().mkString(", "))

println(y.collect().mkString(", "))

After applying MAP:-

X: ['b', 'a', 'c']

Y: [('b', 1), ('a', 1), ('c', 1)]

2. FILTER:

Return a new RDD containing only the elements that satisfy a predicate

val x = sc.parallelize(Array(1,2,3))

val y = x.filter(n => n%2 == 1)

println(x.collect().mkString(", "))

println(y.collect().mkString(", "))

After applying FILTER:-

X: [1, 2, 3]

Y: [1 , 3]

3. FLATMAP:

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

val x = sc.parallelize(Array(1,2,3))

val y = x.flatMap(n => Array(n, n\*100, 42))

println(x.collect().mkString(", "))

println(y.collect().mkString(", "))

After applying FLATMAP:-

X: [1, 2, 3]

Y: [1 , 100, 42, 2, 200, 42, 3, 300, 42]

4. GROUPBY:

Group the data in the original RDD. Create pairs where the key is the output of a user function, and the value is all items for which the function yields this key.

val x = sc.parallelize(Array("John", "Fred", "Anna", "James"))

val y = x.groupBy(w => w.charAt(0))

println(y.collect().mkString(", "))

5. JOIN:

Return a new RDD containing all pairs of elements having the same key in the original RDDs union(otherRDD, numPartitions=None)

val x = sc.parallelize(Array(("a", 1), ("b", 2)))

val y = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))

val z = x.join(y)

println(z.collect().mkString(", "))

After applying JOIN:-

X: [("a", 1), ("b", 2)]

Y: [("a", 3), ("a", 4), ("b", 5)]

Z: [('a', (1, 3)), ('a', (1, 4)), ('b', (2, 5))]

6. PARTITIONBY:

Return a new RDD with the specified number of partitions, placing original items into the partition returned by a user supplied function.

partitionBy(numPartitions, partitioner=portable\_hash)

import org.apache.spark.Partitioner

val x = sc.parallelize(Array(('J',"James"),('F',"Fred"),

('A',"Anna"),('J',"John")), 3)

val y = x.partitionBy(new Partitioner() {

val numPartitions = 2

def getPartition(k:Any) = {

if (k.asInstanceOf[Char] < 'H') 0 else 1

}

})

val yOut = y.glom().collect()

After PARTITIONBY:-

X: Array(Array((A,Anna), (F,Fred)), Array((J,John), (J,James)))

Y: Array(Array((F,Fred), (A,Anna)), Array((J,John), (J,James)))

7. ZIP:

Return a new RDD containing pairs whose key is the item in the original RDD, and whose value is that item’s corresponding element (same partition, same index) in a second RDD

zip(otherRDD)

val x = sc.parallelize(Array(1,2,3))

val y = x.map(n=>n\*n)

val z = x.zip(y)

println(z.collect().mkString(", "))

After applying ZIP:-

X: [1, 2, 3]

Y: [1, 4, 9]

Z: [(1, 1), (2, 4), (3, 9)]

8. COLLECT:

Return all items in the RDD to the driver in a single list

collect()

val x = sc.parallelize(Array(1,2,3), 2)

val y = x.collect()

val xOut = x.glom().collect()

println(y)

After applying collect:-

X: [[1], [2, 3]]

Y: [1, 2, 3]

9. REDUCE:

Aggregate all the elements of the RDD by applying a user function pairwise to elements and partial results, and returns a result to the driver

reduce(f)

val x = sc.parallelize(Array(1,2,3,4))

val y = x.reduce((a,b) => a+b)

println(x.collect.mkString(", "))

println(y)

After applying REDUCE:-

X: [1, 2, 3, 4]

Y: 10

10. AGGREGATE:

Aggregate all the elements of the RDD by:

- applying a user function to combine elements with user-supplied objects,

- then combining those user-defined results via a second user function,

- and finally returning a result to the driver.

def seqOp = (data:(Array[Int], Int), item:Int) =>

(data.\_1 :+ item, data.\_2 + item)

def combOp = (d1:(Array[Int], Int), d2:(Array[Int], Int)) =>

(d1.\_1.union(d2.\_1), d1.\_2 + d2.\_2)

val x = sc.parallelize(Array(1,2,3,4))

val y = x.aggregate((Array[Int](), 0))(seqOp, combOp)

println(y)

After applying AGGREGATE:-

X: [1, 2, 3, 4]

Y: (Array(3, 1, 2, 4),10)