**1. Introduction**

This assignment will help you to consolidate the concepts learnt in the session.

**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.

N/A

**3. Output**

**1)** **limitations of MapReduce**:

**a) Performance**

One of the main limitations of MapReduce is that it persists the full dataset to HDFS after running each job. This is very expensive, because it incurs both three times (for replication) the size of the dataset in disk I/O and a similar amount of network I/O. Spark takes a more holistic view of a pipeline of operations. When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage.

Hadoop Spark has been said to execute batch processing jobs near about 10 to 100 times faster than the [Hadoop MapReduce](https://www.dezyre.com/Big-Data-and-Hadoop/19" \t "_blank" \o "Learn efficient big data processing in hadoop mapreduce) framework just by merely by cutting down on the number of reads and writes to the disc.

In case of MapReduce there are these Map and Reduce tasks subsequent to which there is a synchronization barrier and one needs to preserve the data to the disc. This feature of MapReduce framework was developed with the intent that in case of failure the jobs can be recovered but the drawback to this is that, it does not leverage the memory of the Hadoop cluster to the maximum ,with Hadoop Spark the concept of RDDs (Resilient Distributed Datasets) lets you save data on memory and preserve it to the disc if and only if it is required and as well it does not have any kind of synchronization barriers that possibly could slow down the process. Thus the general execution engine of Spark is much faster than Hadoop MapReduce with the use of memory.

**b)Caching**

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

**c)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.

**d)Ease of Use**

MapReduce doesn’t have an interactive model. Although, Hive & Pig includes command line interfaces, the performance of these systems is still dependent on MapReduce. MapReduce is great for batch processing.

Spark is easy to use compared to MapReduce. Very easy. Even a simple logic or algorithm could take 100′s of lines of code in MapReduce; with Spark the same logic can be written using few lines of code. This leads to a crucial factor called versatility. Many advanced algorithms of machine learning or graph problems, which were impossible in MapReduce, can now be done in Spark. This is driving Spark adoption very highly.

**2)** **What is RDD? Explain few features of RDD?**

An RDD in Spark is simply an automatic immutable distributed collection of objects. Each RDD is split into multiple such objects which are called partitions (), which may be processed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user defined classes.

There are following two types of operations that can be done on RDDs

**Transformations**

It either loads the data (in terms of the partition reference) or constructs a new RDD from the previous one

It does not give you any return value.

**Actions**

Actions, on the other hand, compute a result based on an RDD, and either

return it to the driver program or save it to an external storage system (e.g.,

HDFS)

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.

**few features of RDD:**

--------------------------------

**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 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.

**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.

**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.

**Persistence**

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

**Coarse-grained Operations**

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

**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.

**3) List down few Spark RDD operations and explain each of them**

**RDD Transformations**

Transformations are lazy operations on a RDD that create one or many new RDDs, e.g. map, filter, reduceByKey, join, cogroup, randomSplit.

In other words, transformations are functions that take a RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are mmutable and hence cannot be modified), but always produce one or more new RDDs by applying the computations they represent.

By applying transformations you incrementally build a RDD lineage with all the parent RDDs of the final RDD(s).

Transformations are lazy, i.e. are not executed immediately. Only after calling an **action** are transformations executed.

After executing a transformation, the result RDD(s) will always be different from their parents and can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap, union, cartesian) or the same size (e.g. map).

**Transformation**: Transformation refers to the operation applied on a RDD to create new RDD. Filter, groupBy and map are the examples of transformations.

Returns a new distributed dataset, formed by passing each element of the source through a function func.

2.**filter(func)**

Returns a new dataset formed by selecting those elements of the source on which func returns true.

val x = sc.parallelize(1 to 10)

Or with partition

val x = sc.parallelize(1 to 10, 2)

val y = x.filter(num => num%2==0)

y.collect();

**flatMap(func)**

Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).

val x = sc.parallelize(List("spark rdd example", "sample example"))

val y = x.flatMap(x => x.split(" "))

Map

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

4

**mapPartitions(func)**

**mapPartitions()** can be used as an alternative to map() & foreach(). mapPartitions() is called once for each Partition unlike map() & foreach()which is called for each element in the RDD. The main advantage being that, we can do initialization on Per-Partition basis instead of per-element basis(as done by map() & foreach())

Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> ⇒ Iterator<U> when running on an RDD of type T.

**mapPartitionsWithIndex(func)**

Similar to map Partitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) ⇒ Iterator<U> when running on an RDD of type T.

scala> val rdd1 = sc.parallelize(

List( "yellow", "red", "blue", "cyan", "black" ), 3)

rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[10] at parallelize at :21

scala>

scala> val mapped = rdd1.mapPartitionsWithIndex{

(index, iterator) => { println("Called in Partition -> " + index)

val myList = iterator.toList myList.map(x => x + " -> " + index).iterator } }

mapped: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at mapPartitionsWithIndex at :23

scala>

| mapped.collect()

Called in Partition -> 1

Called in Partition -> 2

Called in Partition -> 0

res7: Array[String] = Array(yellow -> 0, red -> 1, blue -> 1, cyan -> 2, black -> 2)

sample(withReplacement, fraction, seed)

Sample a fraction of the data, with or without replacement, using a given random number generator seed.

**union(otherDataset)**

Returns a new dataset that contains the union of the elements in the source dataset and the argument.

val rdd1 = sc.parallelize(List("lion", "tiger", "tiger", "peacock", "horse"))

rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[33] at parallelize at :21

scala> val rdd2 = sc.parallelize(List("lion", "tiger"))

rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[34] at parallelize at :21

**intersection(otherDataset)**

Returns a new RDD that contains the intersection of elements in the source dataset and the argument.

rdd1.intersection(rdd2).collect();

res24: Array[String] = Array(lion, tiger)distinct([numTasks])

Returns a new dataset that contains the distinct elements of the source dataset.

**groupByKey([numTasks])**

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.

Note − If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.

**reduceByKey(func, [numTasks])**

When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V, V) ⇒ V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

**sortByKey([ascending], [numTasks])**

When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.

val y = x.sortByKey()

y.collect()

**join(otherDataset, [numTasks])**

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

val salesprofit = sc.parallelize(Array(("Cadbury's", 3.5),("Nestle", 2.8),("Mars", 2.5), ("Thorton's", 2.2)));

val salesyear = sc.parallelize(Array(("Cadbury's", 2015),("Nestle", 2014),("Mars", 2014), ("Thorton's", 2013)));

val join = salesprofit.join(salesyear);

join.collect();

**cogroup(otherDataset, [numTasks**])

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called group With.

val grouped = rdd1.cogroup(rdd2)

**cartesian(otherDataset)**

When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

rdd1.cartesian(rdd2).collect();

res28: Array[(String, String)] = Array((lion,lion), (lion,tiger), (tiger,lion), (tiger,tiger), (tiger,lion), (tiger,tiger), (peacock,lion), (peacock,tiger), (horse,lion), (horse,tiger))

**Actions**: Actions refer to an operation which also applies on RDD, that instructs Spark to perform computation and send the result back to driver. This is an example of action.

**reduce(func)**

Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

val x = sc.parallelize(1 to 10, 2)

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

**collect()**

Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

filteredRDD.collect()

**count()**

Returns the number of elements in the dataset.

scala> **val data = sc.parallelize(1 to 10)**

scala> **data.count**

**first()**

Returns the first element of the dataset (similar to take (1)).

scala> val lines = sc.textFile("data.txt")

scala> lines.count()

scala> lines.first()

**saveAsTextFile(path)**

Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file.

filteredRDD.saveAsTextFile("./result")

**foreach(func)**

foreach() is an action. Unlike other actions, foreach do not return any value. It simply operates on all the elements in the RDD. foreach() can be used in situations, where we do not want to return any result, but want to initiate a computation.

val testData=Array(1,2,3)

val inputrdd = sc.parallelize(testData)

inputrdd.foreach{ x => {println(x)}}