**Assignment 16.2:**

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

It’s based on disk based computing.

- Suitable for single pass computations - not iterative computations.

- Needs a sequence of MR jobs to run iterative tasks.

- Needs integration with several other frameworks/tools to solve bigdata

usecases.

1. Apache Storm for stream data processing
2. Apache Mahout for machine learning

**Processing Speed**

In Hadoop, with a parallel and distributed algorithm, MapReduce processes large data

sets. MapReduce requires a lot of time to perform “map and reduce” tasks thereby

increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed. Moreover, MapReduce reads and writes from disk as a result it slows down the processing speed.

**Solution:**

Spark has overcome this issue, by in-memory processing of data. In-memory processing is

faster as no time is spent in moving the data/processes in and out of the disk. Spark runs

applications in Hadoop clusters up to 100x faster in memory and 10x faster on disk. Flink

is also used, as it processes faster than spark because of its streaming architecture and Flink may be instructed to process only the parts of the data that have actually changed, thus significantly increases the performance of the job.

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

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.

It is the primary abstraction in Spark and is the core of Apache Spark.

One could compare RDDs to collections in Scala, i.e. a RDD is computed on

many JVMs while a Scala collection lives on a single JVM.

**Features of RDD**

* **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures
* **Distributed** with data residing on multiple nodes in a cluster.
* **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects

**Additional traits of RDD:-**

* **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
* **Immutable or Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
* **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
* **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
* **IParallel**, i.e. process data in parallel.
* **Typed** — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].
* **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
* **Location-Stickiness** — RDD can define placement preferences to compute partitions (as close to the records as possible).

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

RDD Supports Two Kinds of Operations

• **Actions** - operations that trigger computation and return values

• **Transformations** - lazy operations that return another RDD

**Transformations example:-**

1. **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).

1. **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.

1. **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.

1. **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.

1. **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.

**Action example:-**

1. **count()**

Action count() returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

1. **collect()**

The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.

Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

1. **take(n)**

The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.

For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

1. **top()**

If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top() use default ordering of data.

1. **countByValue()**

The countByValue() returns, many times each element occur in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.countByValue()” will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

1. **reduce()**

The reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.