Spark Assignment 16.2

1) Limitations of MapReduce.

LIMITATIONS OF MAPREDUCE

1. **MR persists back to disk computing for Map/Reduce functions, thus it is slower in-memory computation.**

2. **MR is more difficult to comprehend and larger in terms of lines of code. Thus, it needs more effort in terms of readability and maintainability.**

3. **MR requires a lot of reading/writing to the hard drive to process data.**

4. **MR is slower while performing Iterative computations.**

5. **MR is restricted to batch processing. It does not provide real time processing.**

6. **MapReduce relies on hard drives, if a process crashes in the middle of execution, it could continue where it left off.**

7. **You cannot perform multiple Mapper and Reducer operations in a single MR program, which is limitation in terms of real time use case scenarios where multiple MAP-REDUCE operations are required in one DAG.**

8. **MR has slower response thereby increasing latency.**

9. **MR cannot cache the intermediate data in-memory for a further requirement which diminishes the performance of Hadoop.**

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

Abstraction in Spark Application is illustrated by RDD i.e. Resilient Distributed Dataset.

RDD is a collection of elements partitioned across the nodes in the cluster that can be operated on in parallel.

RDDs are generated by starting the file in HDFS or an existing Scala collection in a driver program and transforming it.

Each dataset in RDD is divided into logical partitions. On the different node of the cluster, we can compute These partitions.

RDDs can be persisted in memory and can be removed from memory based on the usecase.

RDDs automatically recover from node failures. They exhibit the feature of fault tolerance.

PROMINENT FEATURES OF RDD:

1. **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.
2. **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.
3. **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.
4. **IMMUTABILITY**
   -- RDDS are immutable in nature meaning once we create an RDD we cannot manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.
5. **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.
6. **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.

7. **PARALLEL**
   -- Rdds process the data parallelly over the cluster.
8. **LOCATION STICKINESS**
   -- RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAG Scheduler** places the partitions in such a way that task is close to data as much as possible. Thus, it speeds up computation.
9. **COARSE GRAINER 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.
10. **TYPED**
    -- We can have RDD of various types like: RDD [int], RDD [long], RDD [string].
11. **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.

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

Apache Spark RDD operations are classified into two type:
1.Transformations
2. Actions

TRANSFORMATIONS:

These operations will transform an RDD from one form to another. While applying this operation, a new RDD will be produced.

When a transformation is applied to an RDD, it will generate DAG, Source RDD & Function used for transformation.

It will keep on building DAG using the references until any action operation is called on the last lined up RDD. That is why the transformation in Spark are lazy.

In Transformation operation, one RDD may be dependent on none or more than one RDDs. So eventually it will create DAG from start to end which is referred as lineage in spark. In a nutshell, Lineage leverages the fault tolerance feature of Spark.

RDD lineage, also known as **RDD operator graph** or **RDD dependency graph.** It is a logical execution plan i.e., it is DAG of the entire parent RDDs of RDD.

For e.g. MAP. FILTER, FLATMAP etc.

ACTIONS:
These operations will trigger all the lined up transformation on the base RDD (or in the DAG) and will execute the action operation on the last RDD.

For e.g. COLLECT, COUNT, FIRST, saveAsTestFile etc.

Apache Spark RDD operations can also be classified as below two types:
1. Narrow Operations
2. Wide Operations

NARROW RDD OPERATIONS:

These operations can work on a single partition and map the date of that partition to resulting single partition.
These kind of operations which maps data from one to one partition are referred as Narrow operations.
These operations don't require to distribute the data across the partitions.

For e.g. MAP, FLATMAP, MapPartition, Sample, UNION, FILTER etc.

WIDE RDD OPERATIONS:

These operations may require to map the data across the partitions in new RDD.

These kind of operations which maps data from one to many partitions are referred as Wide operations.

Narrow operations don't require to distribute the data across the partitions. In most of the cases, Wide operations distribute the data across the partitions. Wide operations are considered to be costlier than narrow operations due to data shuffling.

For e.g. groupByKey, DISTINCT, JOIN, INTERSECTION, reduceByKey, CARTESIAN, REPARTITION, COALESCE etc.

==========
Various functions of Transformations and Actions are listed below:

1. **TRANSFORMATION FUNCTION: map(function)**

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.

For e.g.
val mapFile = rdd.**map** (x => (x , 1))

---

2. TRANFORMATION FUNCTION: flatmap ()

With the help of **flatMap ()** function, to each input element, we have many elements in an output RDD. The simplest use of flatMap () is to split each input string into words.

The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

For e.g.

val flatmapFile = rdd.**flatMap**(lines => lines.split(" "))

Above function will split each input string into words delimited with space.

val flatmapFile = rdd.**flatMap**(lines => lines.split(","))

Above function will split each input string into words delimited with comma.

---

3. TRANFORMATION FUNCTION: Filter(function)

Spark RDD **filter ()** function is a narrow operation which returns a new RDD, containing only the elements that meet a predicate.

For e.g.

val mapFile = rdd.**flatMap**(lines => lines.split(" ")).**filter**(value => value=="acadgild")

Above function, flatMap function map line into words and then filter the word “acadgild”.

val cnt = mapFile.**count**()

Above function will count the number of all the strings equal to acadgild.

---

4. TRANFORMATION FUNCTION: MapPartitions(function)

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 partition simultaneously.

MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

---

5. TRANFORMATION FUNCTION: MapPartitionWithIndex (function)

It is like mapPartition; Besides mapPartition it provides *function* with an integer value representing the index of the partition, and the map () is applied on partition index wise one after the other.

---

6. TRANFORMATION FUNCTION: 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 e.g.

val rddUnion = rdd1.union(rdd2). union(rdd3)

Above function will unite three rdd1, rdd2 & rdd3 and produces resultant rdd named as rddUnion.

---

7. TRANFORMATION FUNCTION: 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.

For e.g.

val common = rdd1.**intersection**(rdd2)

---

8. TRANFORMATION FUNCTION: Distinct ()

It returns a new dataset that contains the **distinct** elements of the source dataset. It is helpful to remove duplicate data.

For e.g.

val dist = rdd1.distinct()

---

9. TRANFORMATION FUNCTION: GroupByKey ()

While using groupByKey () on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD.

For e.g.

```
val data = sc. parallelize(Array(('k',5), ('s',3), ('s',4), ('p',7), ('p',5), ('t',8), ('k',6)),3)
val group = data.groupByKey()

group.foreach(println)
```

---

10. TRANFORMATION FUNCTION: ReduceBykey (function, [numTasks])

While using **reduceByKey** on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.

For e.g.

```
val words = Array("one","two","two","four","five","six","six","eight","nine","ten")
val data = sc.parallelize(words).map(w => (w,1)).reduceByKey(_+_)
```

Above function will parallelize the Array of String. It will then map each word with count 1, then reduceByKey will merge the count of values having the similar key.

---

11. TRANFORMATION FUNCTION: SortByKey ()

While applying the **sortByKey () function** on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.

For e.g.

```
val sorted = rdd. sortByKey ()
```

In above code, sortByKey () transformation sort the data RDD into Ascending order of the Key(String).

---

12. TRANFORMATION FUNCTION: Join ()

The join() transformation will join two different RDDs on the basis of Key.

For e.g.

```
val rdd1 = sc.parallelize(Array(('A',1),('b',2),('c',3)))

val rdd2 =sc.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))

val joinoutput = rdd1.join(rdd2)

println(joinoutput.collect().mkString(","))
```

---

13. TRANFORMATION FUNCTION: Coalesce ()

To avoid full shuffling of data we use coalesce () function. In **coalesce ()** we use existing partition so that less data is shuffled. Using this we can cut the number of the partition.

The coalesce will decrease the number of partitions of the source RDD to numPartitions define in coalesce argument.

For e.g.

val result = rdd1.coalesce(2)

In above code, out of total n number of partitions only 2 partitions data will be shuffled.

---

14. ACTION FUNCTION: Count ()

Action operation Count() returns the number of elements in RDD.

For e.g.

Val rdd2 = rdd1.**count**()

---

15. ACTION FUNCTION: Collect ()

This action operation returns our entire RDDs content to driver program.

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

For e.g.

Println(rdd.collect())

---

16. ACTION FUNCTION: Take(n)

The take (n) Action will return an array with the first n elements of the data set defined in the taking argument.

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 e.g.

```
val  ARR = rdd.take(10)

ARR.foreach(println)
```

---

17. ACTION FUNCTION: Top ()

This function returns top n elements from ordered RDD.

For e.g.

```
val rdd1 = sc.textFile("spark_test.txt").rdd
val rdd2 = rdd1.map(line => (line,line.length))
val rdd3 = rdd2.top(3)
rdd3.foreach(println)
```

---

18. ACTION FUNCTION: CountByValue ()

The *countByValue()* action will return a hashmap of (K, Int) pairs with the count of each key.

For e.g.

```
val rdd1 = sc.textFile("spark_test.txt").rdd
val rdd2= rdd1.map(line => (line,line.length)).countByValue()
rdd2.foreach(println)
```

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)}

---

19. ACTION FUNCTION: 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.

For e.g.

```
val rdd1 = sc. parallelize (List (20,32,45,62,8,5))
val sum = rdd1.reduce(_+_)
println(sum)
```

---

20. ACTION FUNCTION: Fold ()

The signature of the fold() is like reduce(). Besides, it takes "zero value" as input, which is used for the initial call on each partition. But, the condition with zero value is that it should be the identity element of that operation. The key difference between fold() and reduce() is that, reduce() throws an exception for empty collection, but fold() is defined for empty collection.
For example, zero is an identity for addition; one is identity element for multiplication. The return type of *fold()* is same as that of the element of RDD we are operating on.
For example,

Val rdd1 = rdd.fold(0)((x, y) => x + y).

Another example :

```
val rdd2 = sc.parallelize(List(("maths", 80),("science", 90)))
val additionalMarks = ("extra", 4)
val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2
("total", add)
}
println(sum)
```

In above code additionalMarks is an initial value. This value will be added to the int value of each record in the source RDD.

---

21. ACTION FUNCTION: Aggregate ()

The aggregate function allows the user to apply two different reduce functions to the RDD.

The first reduce function is applied within each partition to reduce the data within each partition into a single result.

The second reduce function is used to combine the different reduced results of all partitions together to arrive at one final result.

The ability to have two separate reduce functions for intra partition versus across partition reducing adds a lot of flexibility.

For example, the first reduce function can be the max function and the second one can be the sum function. The user also specifies an initial value.

---

22. ACTION FUNCTION: Foreach ()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the *driver*.

The foreach () action runs a function (println) on each element of the dataset group.

For e.g.

```
val rdd1 = sc. parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val rdd3 = rdd2. groupByKey(). collect()
rdd3.foreach(println)
```

---

23. TRANSFORMING FUNCTION: CARTESIAN()

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

---

24. ACTION FUNCTION: FIRST()

While applying it to RDD , it returns the first element of RDD.

```
Val firstelement = rdd.first()
```

---

25. ACTION FUNCTION: saveAsTextFile(path)

This function 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.

---

26. ACTION FUNCTION: saveAsSequenceFile(path) (Java and Scala)

This function writes the elements of the dataset as a Hadoop Sequence File in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also

available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

---

27. ACTION FUNCTION : saveAsObjectFile(path) (Java and Scala)

This function writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().