Problem Statement

1. Pen down the limitations of MapReduce.

Solution-

There are certain cases where Map Reduce is not a suitable choice. Map Reduce has been designed to handle large scale of data and one should understand that we can't expect a Map Reduce job to give the result inn couple of milliseconds. Below are some limitations of Map Reduce where it is not suggested to use Map Reduce-

- Real-time processing.
- It's not always very easy to implement each and everything as a Map Reduce program.
- When your intermediate processes need to talk to each other(jobs run in isolation).
- When your processing requires lot of data to be shuffled over the network.
- When you need to handle streaming data. MR is best suited to batch process huge amounts of data which you already have with you.
- When you can get the desired result with a standalone system. It's obviously less painful to configure and manage a standalone system as compared to a distributed system.
- When you have OLTP needs. MR is not suitable for a large number of short on-line transactions.
- Processing graphs.
- Iterations - when you need to process data again and again.
- When map phase generate too many keys. Then sorting takes for ever.
- Map Reduce processing is tightly coupled processing, so if the computation of a value depends on previously computed values, then Map Reduce cannot be used. For example- Fibonacci series where each value is summation of the previous two values. i.e., f(k+2) = f(k+1) + f(k).
- **Map Reduce** 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.
- 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** 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.
- Map Reduce 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.
- MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

But above listed points should not be considered as disadvantages. These are the situations where Map Reduce is not a good choice.

Problem Statement

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

Solution- RDD stands for **Resilient Distributed Datasets**. RDD is basically is a fault-tolerant collection of elements that can be operated on in parallel. It 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.

In a typical Spark program, one or more RDDs are loaded as input and through a series of transformations are turned into a set of target RDDs, which have an action performed on them (such as computing a result or writing them to persistent storage).

Now each of its term has significance. For example the term “resilient” in “Resilient Distributed Dataset” refers to the fact that Spark can automatically reconstruct a lost partition by re-computing it from the RDDs that it was computed from. Since it operates on data which is distributed in a cluster itself that is why the term- distributed and dataset.

There are three ways of creating RDDs:-

1. From an in-memory collection of objects (known as parallelizing a collection). It is useful for doing CPU-intensive computations on small amounts of input data in parallel. For example, the following runs separate computations on the numbers from 1 to 10.
   val params = sc.parallelize(1 to 10)

2. Using a dataset from external storage (such as HDFS) which means create an RDD by creating a reference to an external dataset.
   Ex- **val text: RDD[String] = sc.textFile(inputPath)**
   The path may be any Hadoop filesystem path, such as a file on the local filesystem or on HDFS. Internally, Spark uses TextInputFormat from the old MapReduce API to read the file. This means that the file-splitting behavior is the same as in Hadoop itself, so in the case of HDFS there is one Spark partition per HDFS block.

3. The third way of creating an RDD is by transforming an existing RDD.

Now as far as operations with RDD is concerned, spark has two categories of operations on RDDs-
a. Transformations- A transformation generates a new RDD from an existing one. Transformations do not have any immediate effect–they are lazy, in the sense that they don’t perform any work until an action is performed on the transformed RDD.
b. Actions- An action triggers a computation on an RDD and does something with the results–either returning them to the user, or saving them to external storage. Actions have an immediate effect.
For example-

```
val text = sc.textFile(inputPath)
val lower: RDD[String] = text.map(_.toLowerCase())
lower.foreach(println(_))
```

The map() method is a transformation, which Spark represents internally as a function (toLowerCase()) to be called at some later time on each element in the input RDD (text). The function is not actually called until the foreach() method (which is an action) is invoked and Spark runs a job to read the input file and call toLowerCase() on each line in it, before writing the result to the console. One way of telling if an operation is a transformation or an action is by looking at its return type: if the return type is RDD, then it’s a transformation; otherwise, it’s an action.

Problem Statement

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

Solution- RDDs support two types of operations, transformations and actions. Transformations are operations on RDDs that return a new RDD, such as map and filter. Actions are operations that return a result back to the driver program or write it to storage, and kick off a computation, such as count and first.

Transformations-

Transformations are operations on RDDs that return a new RDD. Transformed RDDs are computed lazily, only when you use them in an action. Many transformations are element-wise, that is they work on one element at a time, but this is not true for all transformations.
Some of the examples are-

1. **Flatmap-** In below example we are first creating a RDD to load the test file then in next line we are transforming it by creating a new RDD by using flatmap the text file. Flatmap basically will transform the strings present in file into a collection of N collections then flatten these into a single RDD of results-

```
val logFile = "hdfs://master.backtobazics.com:9000/user/root/sample.txt"
val lineRDD = sc.textFile(logFile) //Transformation 1 -> DAG created
                                    //{DAG: Start -> [sc.textFile(logFile)]}

val wordRDD = lineRDD.flatMap(_.split(" "))  //Transformation 2 -> wordRDD DAG updated
                                              //{DAG: Start -> [sc.textFile(logFile)]
                                             //          -> [lineRDD.flatMap(_.split(" "))]}
```

2. **Filter-** Now in below example the RDD **filteredWordRDD** is getting created after transforming the RDD **wordRDD.filter.** Here we are basically filtering out the above created RDD based on the string containing –"**the"**

```
val filteredWordRDD = wordRDD.filter(_.equalsIgnoreCase("the"))
                               //Transformation 3 -> filteredWordRDD DAG updated
                               //{DAG: Start -> [sc.textFile(logFile)]
                               //      -> [lineRDD.flatMap(_.split(" "))]
                               //      -> [wordRDD.filter(_.equalsIgnoreCase("the"))]}
```

3. **Map()-Map()** operation applies to each element of **RDD** and it returns the result as new RDD. In the Map, operation developer can define his own custom business logic.
For example-

```
val data = spark.read.textFile("INPUT-PATH").rdd
val newData = data.map (line => line.toUpperCase() )
```

Here in above example we are creating a RDD data to read the txt file and then the RDD newData will convert each and every record of RDD to upper case.

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-

```
val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))
val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
val rddUnion = rdd1.union(rdd2).union(rdd3)
rddUnion.foreach(Println)
```

In above example union() operation will return a new dataset that contains the union of the elements in the source dataset (rdd1) and the argument (rdd2 & rdd3).

5. **Distinct()-**It returns a new dataset that contains the **distinct** elements of the source dataset. It is helpful to remove duplicate data.
   For example-
   ```
   val rdd1 = park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
   val result = rdd1.distinct()
   println(result.collect().mkString(", "))
   ```
   In the above example, the distinct function will remove the duplicate record i.e. (3,'”nov”,2014).

6. **Join()-**The **Join** is database term. It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.
   For example-
   ```
   val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
   val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
   val result = data.join(data2)
   println(result.collect().mkString(","))
   ```
   In above example the join() transformation will join two different RDDs on the basis of Key.

Actions

Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

Some basic action operations are-

1. **Count()-** Action **count()** returns the number of elements in RDD.
   For example-
   ```
   val data = spark.read.textFile("spark_test.txt").rdd
   val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
   println(mapFile.count())
   ```

   In above example *flatMap()* function maps line into words and count the word “Spark” using *count()* Action after filtering lines containing “Spark” from mapFile.

2. **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.
   For example-
   ```
   val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
   val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
   val result = data.join(data2)
   println(result.collect().mkString(","))
   ```

   join() transformation in above code will join two RDDs on the basis of same key(alphabet). After that *collect()* action will return all the elements to the dataset as an Array.

3. **Top()-** Action *top()* use default ordering of data.
   For example-
   ```
   val data = spark.read.textFile("spark_test.txt").rdd
   ```

```
val mapFile = data.map(line => (line,line.length))
val res = mapFile.top(3)
res.foreach(println)
```

map() operation will map each line with its length. And top(3) will return 3 records from mapFile with default ordering.

4. **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*. In this case, **foreach()** function is useful.

   For example-

   ```
   val data =
   spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
   val group = data.groupByKey().collect()
   group.foreach(println)
   ```

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

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

   For example-

   ```
   val data = spark.read.textFile("spark_test.txt").rdd
   val result= data.map(line => (line,line.length)).countByValue()
   result.foreach(println)
   ```

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