1) Pen down the limitations of MapReduce

1. Processing speed

In Hadoop, 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 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 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

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

RDD stands **Resilient Distributed Dataset**. RDDs are the fundamental abstraction of Apache Spark. It is an immutable distributed collection of the dataset. Each dataset in RDD is divided into logical partitions. On the different node of the cluster, we can compute These partitions. RDDs are a read-only partitioned collection of record. we can create RDD in three ways:

- **Parallelizing** already existing collection in driver program.
- **Referencing a dataset** in an external storage system (e.g. HDFS, Hbase, shared file system).
- Creating RDD **from already existing RDDs**.

3.1. 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. refer this comprehensive guide to Learn Spark in-memory computation in detail.

3.2. 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. Follow this guide to learn Spark lazy evaluation in great detail.

3.3. 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. Learn Fault tolerance is Spark in detail.

3.4. Immutability

- RDDS are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve

consistency through immutability.

- 3.5. 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. Follow this guide for the detailed study of RDD persistence in Spark.

- 3.6. 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.

- 3.7. Parallel
- Rdd, process the data parallelly over the cluster.

- 3.8. 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. Follow this guide to learn What is DAG?

- 3.9. Coarse-grained 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.

- 3.10. Typed
- We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

- 3.11. 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.

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.

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.

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.

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.

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.

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.

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.

distinct()

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

groupByKey()

When we use **groupByKey()** on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD. In this transformation, lots of unnecessary data get to transfer over the network.

reduceByKey(func, [numTasks])

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

sortByKey()

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

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.
The boon of using keyed data is that we can combine the data together. The join() operation combines two data sets

on the basis of the key.

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. Suppose, we have four nodes and we want only two nodes. Then the data of extra nodes will be kept onto nodes which we kept.