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

-Hadoop MapReduce requires every problem to be broken down into a sequence of map and reduce jobs. It is hard to express non-trivial algorithms with just map and reduce.

-A MapReduce-based data processing pipeline consists of a sequence of jobs, where each job reads data from disk, processes it, and writes the results to disk. Thus, a complex data processing application implemented with MapReduce may read data from and write data to disk several times. This increases the I/O Latency which impacts the overall job execution time

-Hadoop MapReduce creates a DAG with exactly two predefined stages—Map and Reduce—for every job.

A complex data processing algorithm implemented with MapReduce may need to be split into multiple jobs,

which are executed in sequence. This design prevents Hadoop MapReduce from doing any optimization.

-Hadoop MapReduce is designed just for batch processing. Therefore, a developer using MapReduce has to use different frameworks for stream processing and graph computing.

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

RDD represents a collection of partitioned data elements that can be operated on in parallel. It is the primary data abstraction mechanism in spark. It is defined as an abstract class in spark library.

Features of RDD:

**Immutable**

An RDD is an immutable data structure. Once created, it cannot be modified in-place. Basically, an

operation that modifies an RDD returns a new RDD

**Partitioned**

Data represented by an RDD is split into partitions. These partitions are generally distributed across a cluster

of nodes.

**Fault Tolerant**

RDD automatically handles node failures. When a node fails, and partitions stored on that node

become inaccessible, Spark reconstructs the lost RDD partitions on another node. Spark stores lineage

information for each RDD. Using this lineage information, it can recover parts of an RDD or even an entire

RDD in the event of node failures.

**Interface**

RDD is an interface for processing data. It is defined as an abstract class

in the Spark library. RDD provides a uniform interface for processing data from a variety of data sources,

Such as HDFS, HBase, Cassandra, and others. The same interface can also be used to process data stored in

Memory across a cluster of nodes.

Spark provides concrete implementation classes for representing different data sources. Examples

of concrete RDD implementation classes include HadoopRDD, ParallelCollectionRDD, JdbcRDD, and

CassandraRDD. They all support the base RDD interface.

**Strongly Typed**

The RDD class definition has a type parameter. This allows an RDD to represent data of different types. It is

a distributed collection of homogenous elements, which can be of type Integer, Long, Float, String, or a

custom type defined by an application developer. Thus, an application always works with an RDD of some

type. It can be an RDD of Integer, Long, Float, Double, String, or a custom type.

**In Memory**

The RDD classprovides the API for enabling in-memory cluster computing. Spark allows RDDs to be cached or persisted in memory.

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

**Creating an RDD**

**Parallelize**

This method creates an RDD from a local Scala collection. It partitions and distributes the elements of a

Scala collection and returns an RDD representing those elements

val xs = (1 to 10000).toList

val rdd1 = sc.parallelize(xs)

**textFile**

The textFile method creates an RDD from a text file. It can read a file or multiple files in a directory stored

on a local file system, HDFS, Amazon S3, or any other Hadoop-supported storage system. It returns an RDD

of Strings, where each element represents a line in the input file.

val rdd2 = sc.textFile("/acadgild/learning.txt")

**wholeTextFiles**

This method reads all text files in a directory and returns an RDD of key-value pairs. Each key-value pair in

the returned RDD corresponds to a single file. The key part stores the path of a file and the value part stores

the content of a file. This method can also read files stored on a local file system, HDFS, Amazon S3, or any

other Hadoop-supported storage system.

val rdd3 = sc.wholeTextFiles("("/acadgild/\*.txt")

**sequenceFile**

The sequenceFile method reads key-value pairs from a sequence file stored on a local file system, HDFS, or

any other Hadoop-supported storage system. It returns an RDD of key-value pairs. In addition to providing

the name of an input file, you have to specify the data types for the keys and values as type parameters when

you call this method.

val rdd4 = sc.sequenceFile[String, String](" /acadgild/file1")

**RDD Operations**

RDD operations can be categorized into two types: transformation and action. A transformation creates

a new RDD. An action returns a value to a driver program.

**Transformations**

A transformation method of an RDD creates a new RDD by performing a computation on the source RDD.

**map**

The map method is a higher-order method that takes a function as input and applies it to each element in

the source RDD to create a new RDD. The input function to map must take a single input parameter and

return a value.

val lines = sc.textFile("/acadgild/file.txt")

val lengths = lines map { l => l.length}

**filter**

The filter method is a higher-order method that takes a Boolean function as input and applies it to each

element in the source RDD to create a new RDD. A Boolean function takes an input and returns true or

false. The filter method returns a new RDD formed by selecting only those elements for which the input

Boolean function returned true. Thus, the new RDD contains a subset of the elements in the original RDD.

val lines = sc.textFile(("/acadgild/file.txt”)

val longLines = lines filter { l => l.length > 80}

**flatMap**

The flatMap method is a higher-order method that takes an input function, which returns a sequence for

each input element passed to it. The flatMap method returns a new RDD formed by flattening this collection

of sequence.

val lines = sc.textFile("/acadgild/file.txt")

val words = lines flatMap { l => l.split(" ")}

**mapPartitions**

The higher-order mapPartitions method allows you to process data at a partition level. Instead of passing

one element at a time to its input function, mapPartitions passes a partition in the form of an iterator.

The input function to the mapPartitions method takes an iterator as input and returns another iterator as

output. The mapPartitions method returns new RDD formed by applying a user-specified function to each

partition of the source RDD.

val lines = sc.textFile("...")

val lengths = lines mapPartitions { iter => iter.map { l => l.length}}

**union**

The union method takes an RDD as input and returns a new RDD that contains the union of the elements in

the source RDD and the RDD passed to it as an input.

val linesFile1 = sc.textFile("/acadgild/file1")

val linesFile2 = sc.textFile("/acadgild/file2")

val linesFromBothFiles = linesFile1.union(linesFile2)

**Actions**

Actions are RDD methods that return a value to a driver program.

**collect**

The collect method returns the elements in the source RDD as an array. This method should be used with

caution since it moves data from all the worker nodes to the driver program. It can crash the driver program

if called on a very large RDD.

val rdd = sc.parallelize((1 to 10000).toList)

val filteredRdd = rdd filter { x => (x % 1000) == 0 }

val filterResult = filteredRdd.collect

**count**

The count method returns a count of the elements in the source RDD.

val rdd = sc.parallelize((1 to 10000).toList)

val total = rdd.count

**countByValue**

The countByValue method returns a count of each unique element in the source RDD. It returns an instance

of the Map class containing each unique element and its count as a key-value pair.

val rdd = sc.parallelize(List(1, 2, 3, 4, 1, 2, 3, 1, 2, 1))

val counts = rdd.countByValue

**first**

The first method returns the first element in the source RDD.

val rdd = sc.parallelize(List(10, 5, 3, 1))

val firstElement = rdd.first

**max**

The max method returns the largest element in an RDD.

val rdd = sc.parallelize(List(2, 5, 3, 1))

val maxElement = rdd.max