1.Limitations of Mapreduce:

**1. Processing speed**

In [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), 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**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) 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**](https://goo.gl/VKRPf6) 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](http://data-flair.training/blogs/rdd-in-apache-spark/) 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 3 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.

There are two operations in RDD namely Transformation and Action

Features:

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 can not 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

Rdd, 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 DAGScheduler places the partitions in such a way that task is close to data as much as possible. Thus speed up computation

### 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.

### 10. Typed

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

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

RDD Operations are 2 types:

1.Transformations

2.Actions

Transformations:

Any function that returns an RDD is a transformation, elaborating it further we can say that Transformation is functions which create a new data set from an existing one by passing each data set element through a function and returns a new RDD representing the results.

All transformations in Spark are lazy. They do not compute their results right away. Instead, they just remember the transformations applied to some base data set (e.g. a file). The transformations are only computed when an action requires a result that needs to be returned to the driver program.

**Map:**  
Map will take each row as input and return an RDD for the row.

val x = sc.parallelize(List("spark", "rdd", "example", "sample", "example"), 3)

scala> val y = x.map(x => (x, 1))

**Flat map:  
flatMap** will take an iterable data as input and returns the RDD as the contents of the iterator.

val x = sc.parallelize(List("spark rdd example", "sample example"), 2)

val y = x.map(x => x.split(" ")) *// split(" ") returns an array of words*

scala> y.collect

res0: Array[Array[String]] = Array(Array(spark, rdd, example), Array(sample, example))

*// flatMap operation will return Array of words in following case : Check type of res1*

scala> val y = x.flatMap(x => x.split(" "))

scala> y.collect

res1: Array[String] = Array(spark, rdd, example, sample, example)

*// rdd y can be re written with shorter syntax in scala as*

scala> val y = x.flatMap(\_.split(" "))

scala> y.collect

res2: Array[String] = Array(spark, rdd, example, sample, example)

**Filter:  
filter** returns an RDD which meets the filter condition.

scala> val x = sc.parallelize(1 to 10, 2)

*// filter operation*

scala> val y = x.filter(e => e%2==0)

scala> y.collect

res0: Array[Int] = Array(2, 4, 6, 8, 10)

*// rdd y can be re written with shorter syntax in scala as*

scala> val y = x.filter(\_ % 2 == 0)

scala> y.collect

res1: Array[Int] = Array(2, 4, 6, 8, 10)

**ReduceByKey:  
reduceByKey**takes a pair of key and value pairs and combines all the values for each unique key. Below is the

* reduceByKey is a transformation operation in Spark hence it is lazily evaluated
* It is a wide operation as it shuffles data from multiple partitions and creates another RDD
* Before sending data across the partitions, it also merges the data locally using the same associative function for optimized data shuffling
* It can only be used with RDDs which contains key and value pairs kind of elements
* It accepts a Commutative and Associative function as an argument
  + The parameter function should have two arguments of the same data type
  + The return type of the function also must be same as argument types
* scala> val x = sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),
* | ("a", 1), ("b", 1), ("b", 1),
* | ("b", 1), ("b", 1)), 3)
* x: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[1] at parallelize at <console>:21
* *// Applying reduceByKey operation on x*
* scala> val y = x.reduceByKey((accum, n) => (accum + n))
* y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[2] at reduceByKey at <console>:23
* scala> y.collect
* res0: Array[(String, Int)] = Array((a,3), (b,5))
* *// Another way of applying associative function*
* scala> val y = x.reduceByKey(\_ + \_)
* y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3] at reduceByKey at <console>:23
* scala> y.collect
* res1: Array[(String, Int)] = Array((a,3), (b,5))
* *// Define associative function separately*
* scala> def sumFunc(accum:Int, n:Int) = accum + n
* sumFunc: (accum: Int, n: Int)Int
* scala> val y = x.reduceByKey(sumFunc)
* y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:25
* scala> y.collect
* res2: Array[(String, Int)] = Array((a,3), (b,5))

**Actions:**

Actions return final results of RDD computations. Actions trigger execution using lineage graph to load the data into original RDD and carry out all intermediate transformations and return the final results to the Driver program or writes it out to the file system.

**Collect:**  
collect is used to return all the elements in the RDD

val x = sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),

| ("a", 1), ("b", 1), ("b", 1),

| ("b", 1), ("b", 1)), 3)

x: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[1] at parallelize at <console>:21

*// Applying reduceByKey operation on x*

scala> val y = x.reduceByKey((accum, n) => (accum + n))

y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[2] at reduceByKey at <console>:23

scala> y.collect

**Count:  
count**is used to return the number of elements in the RDD

**CountByValue:  
countByValue**is used to count the number of occurrences of the elements in the RDD

**Reduce:**

* reduce is an action operation in Spark hence it triggers execution of DAG and gets execute on final RDD
* It is a wide operation as it is shuffling data from multiple partitions and reduces to a single value
* It accepts a Commutative and Associative function as an argument
  + The parameter function should have two arguments of the same data type
  + The return type of the function also must be same as argument types
* val x = sc.parallelize(1 to 10, 2)
* scala> val y = x.reduce((accum,n) => (accum + n))
* y: Int = 55
* *// shorter syntax*
* scala> val y = x.reduce(\_ + \_)
* y: Int = 55
* *// same thing for multiplication*
* scala> val y = x.reduce(\_ \* \_)
* y: Int = 3628800

**Take: take**will display the number of records we explicitly specify