

### Problem Statement:

- 1) Pen down the limitations of MapReduce.
- 2) What is RDD? Explain few features of RDD?
- 3) List down few Spark RDD operations and explain each of them.

### 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

## 7. Issue with small files

Hadoop 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.

### What Is RDD? Explain few features of RDD?

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark and main logical data unit. 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.

Distributed means, each RDD is divided into multiple partitions. Each of these partitions can reside in memory or stored on disk of different machines in a cluster. RDDs are immutable (Read Only) data structure. You can't change original RDD, but you can always transform it into different RDD with all changes you want.

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. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs – **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

## Features of Spark RDD:



### 1. In-memory Computation

Spark RDDs have a provision of **in-memory computation**. It stores intermediate results in distributed memory (RAM) instead of stable storage (disk).

### 2. Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program.

### 3. Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself.

#### 4. Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

#### 5. Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

#### 6. Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

#### 7. Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

#### 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.

### List down few Spark RDD operations and explain each of them

#### Transformations:

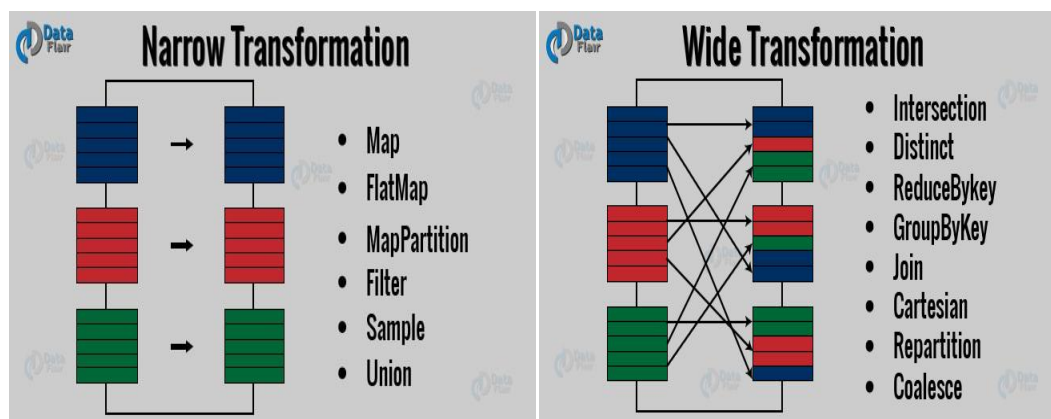
Any function that returns an RDD is a transformation. Transformations are functions that create a new data set from an existing one by passing each data set element through a function and returning a new RDD representing the results.

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.

There are two types of transformations:

- **Narrow transformation** – In *Narrow transformation*, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. *Narrow transformations* are the result of *map()*, *filter()*.
- **Wide transformation** – In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupByKey()* and *reduceByKey()*.



- **map(func):**

Map will take each row as input and return an RDD for the row. 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 example, in RDD {1, 2, 3, 4, 5} if we apply “`rdd.map(x=>x+2)`” we will get the result as (3, 4, 5, 6, 7).

- **flat Map():**

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

For Example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val flatmapFile = data.flatMap(lines => lines.split(" "))
flatmapFile.foreach(println)
```

In above code, flatMap() function splits each line when space occurs.

- **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.

For Example:

```
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value =>
value=="spark")
println(mapFile.count())
```

In above code, flatMap function map line into words and then count the word “Spark” using count() Action after filtering lines containing “Spark” from mapFile.

- **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.

For example, the elements of **RDD1** are (Spark, Spark, **Hadoop**, **Flink**) and that of **RDD2** are (**Big data**, Spark, Flink) so the resultant **rdd1.union(rdd2)** will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

- **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.

Consider an example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant **rdd1.intersection(rdd2)** will have elements (spark).

- **distinct()**

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

For example, if RDD has elements (Spark, Spark, Hadoop, Flink), then **rdd.distinct()** will give elements (Spark, Hadoop, Flink).

- **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.

Spark provides the provision to save data to disk when there is more data shuffled onto a single executor machine than can fit in memory. Follow this link to [learn about RDD Caching and Persistence mechanism](#) in detail.

**groupByKey() 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 **groupByKey()** will group the integers on the basis of same key(alphabet). After that **collect()** action will return all the elements of the dataset as an Array.

- **reduceByKey(func, [numTasks])**

**reduceByKey** takes a pair of key and value pairs and combines all the values for each unique key. So the pairs on the same machine with the same key are combined, before the data is shuffled.

For Example:

```
val words =
```



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

The above code 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.

- **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.

**sortByKey() example:**

```
val data =
spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",
82), ("computer",65), ("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)
```

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

- **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.

**Join() 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(","))
```

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

For Example:

```
val rdd1 =  
spark.sparkContext.parallelize(Array("jan","feb","mar","april","may","jun"),3)  
val result = rdd1.coalesce(2)  
result.foreach(println)
```

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

## **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.

When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values.

- **collect()**

**collect** is used to return all the elements in the RDD. The application of `collect()` is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.

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(","))
```

The `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.

- **count()**

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

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD

“`rdd.count()`” will give the result 8

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

- **take(n)**

The action *take(n)* returns n number of elements from RDD. 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 example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “*take (4)*” will give result {2, 2, 3, 4}

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()
val twoRec = result.take(2)
twoRec.foreach(println)
```

The *take(2)* Action will return an array with the first 2 elements of the data set defined in the taking argument.

- **top()**

If ordering is present in our RDD, then we can extract top elements from our RDD using *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)
```

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

- **countByValue()**

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)}

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.

- **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. It accepts commutative and associative operations as an argument.

**Example:**

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

The *reduce()* action in above code will add the elements of the source RDD.

- **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, `rdd.fold(0)((x, y) => x + y)`.

**Example:**

```
val rdd1 = spark.sparkContext.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.

- **aggregate()**

It gives us the flexibility to get data type different from the input type. The **aggregate()** takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

- **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.