Question no. 1 - Pen down the limitations of MapReduce.

While development of Hadoop's MapReduce the vision was pretty limited, it was developed just to handle 1 problem: **Batch processing**.

MapReduce cannot handle:

1. Interactive Processing
2. Real-time (stream) Processing
3. Iterative (delta) Processing
4. In-memory Processing
5. Graph Processing

Let's discuss the limitations in great details:

1. 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.

2. Slow Processing Speed

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: **Map** and **Reduce** and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

3. Support for Batch Processing only

Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the **Hadoop cluster** to the maximum.

4. No Real-time Data Processing

Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

5. No Delta Iteration

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

6. 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.

7. Not Easy to 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 adding one such as **hive** and **pig** makes working with MapReduce a little easier for adopters.

8. No Caching

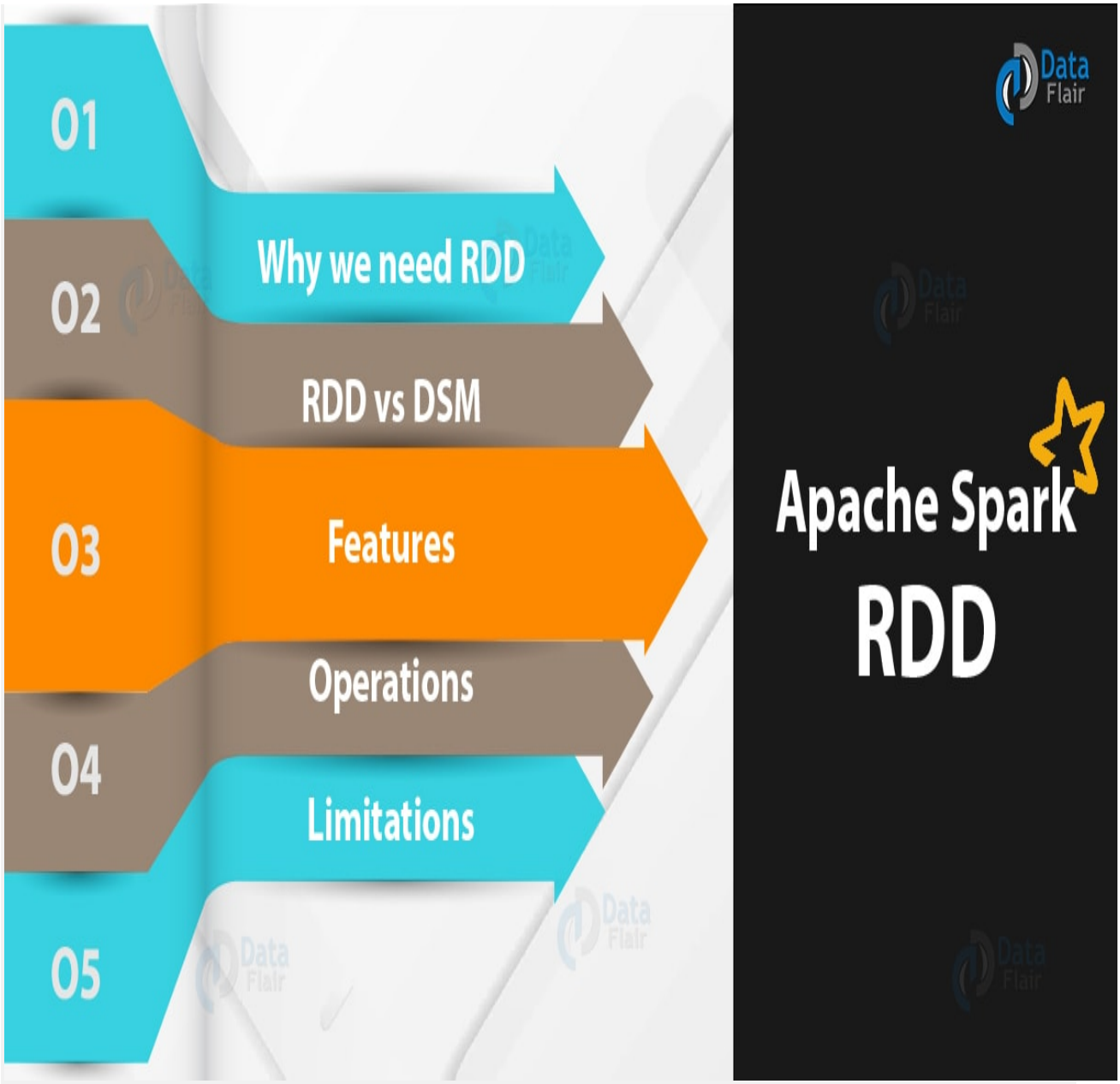
Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

Question no. 2 What is RDD? Explain few features of RDD?

Spark RDD – Introduction, Features & Operations of RDD

1. Objective – Spark RDD

RDD (Resilient Distributed Dataset) is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is logically partitioned across many servers so that they can be computed on different nodes of the cluster. In this blog, we are going to get to know about what is RDD in Apache Spark. What are the features of RDD, What is the motivation behind RDDs, RDD vs DSM? We will also cover Spark RDD operation i.e. transformations and actions, various limitations of RDD in Spark and how RDD make Spark feature rich in this Spark tutorial.



Spark RDD – Introduction, Features & Operations of RDD

.

2. What is Apache Spark RDD?

RDD stands for "**Resilient Distributed Dataset"**. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster.
Decomposing the name RDD:

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph(**DAG**) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed**, since Data resides on multiple nodes.
- **Dataset** represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Hence, each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. It posses self-recovery in the case of failure.

There are three **ways to create RDDs in Spark** such as – *Data in stable storage, other RDDs, and parallelizing already existing collection in driver program*. One can also operate Spark RDDs in parallel with a low-level API that offers *transformations* and *actions*. We will study these Spark RDD Operations later in this section.
Spark RDD can also be **cached** and **manually partitioned**. Caching is beneficial when we use RDD several times. And manual partitioning is important to correctly balance partitions. Generally, smaller partitions allow distributing RDD data more equally, among more executors. Hence, fewer partitions make the work easy.
Programmers can also call a **persist** method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs **in memory** by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to persist.

3. Why do we need RDD in Spark?

The key motivations behind the concept of RDD are-

- Iterative algorithms.
- Interactive data mining tools.
- **DSM** (Distributed Shared Memory) is a very general abstraction, but this generality makes it harder to implement in an efficient and fault tolerant manner on commodity clusters. Here the need of RDD comes into the picture.

- In distributed computing system data is stored in intermediate stable distributed store such as **HDFS** or Amazon S3. This makes the computation of job slower since it involves many IO operations, replications, and serializations in the process.

In first two cases we keep data in-memory, it can improve performance by an order of magnitude.
The main challenge in designing RDD is defining a program interface that provides fault tolerance efficiently. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on **coarse-grained transformation** rather than **fine-grained** updates to shared state.
Spark exposes RDD through language integrated API. In integrated API each data set is represented as an object and transformation is involved using the method of these objects.

Apache Spark evaluates RDDs lazily. It is called when needed, which saves lots of time and improves efficiency. The first time they are used in an action so that it can pipeline the transformation. Also, the programmer can call a persist method to state which RDD they want to use in future operations.

4. Spark RDD vs DSM (Distributed Shared Memory)

In this Spark RDD tutorial, we are going to get to know the difference between RDD and DSM which will take RDD in Apache Spark into the limelight.

i. Read

- **RDD –** The read operation in RDD is either coarse grained or fine grained. Coarse-grained meaning we can transform the whole dataset but not an individual element on the dataset. While fine-grained means we can transform individual element on the dataset.
- **DSM –** The read operation in Distributed shared memory is fine-grained.

ii. Write

- **RDD –** The write operation in RDD is coarse grained.
- **DSM –** The Write operation is fine grained in distributed shared system.

iii. Consistency

- **RDD –** The consistency of RDD is trivial meaning it is immutable in nature. Any changes on RDD is permanent i.e we can not realtor the content of RDD. So the level of consistency is high.
- **DSM –** In Distributed Shared Memory the system guarantees that if the programmer follows the rules, the memory will be consistent and the results of memory operations will be predictable.

iv. Fault-Recovery Mechanism

- **RDD –** The lost data can be easily recovered in Spark RDD using lineage graph at any moment. Since for each transformation, new RDD is formed and RDDs are immutable in nature so it is easy to recover.
- **DSM –** Fault tolerance is achieved by a checkpointing technique which allows applications to roll back to a recent checkpoint rather than restarting.

v. Straggler Mitigation

Stragglers, in general, are those that take more time to complete than their peers. This could happen due to many reasons such as load imbalance, I/O blocks, garbage collections, etc.

The problem with stragglers is that when the parallel computation is followed by synchronizations such as reductions. This would cause all the parallel tasks to wait for others.

- **RDD –** In RDD it is possible to mitigate stragglers using backup task.
- **DSM –** It is quite difficult to achieve straggler mitigation.

vi. Behavior if not enough RAM

- **RDD –** If there is not enough space to store RDD in RAM then the RDDs are shifted to disk**.**
- **DSM –** In this type of system, the performance decreases if the RAM runs out of storage.

5. Features of Spark RDD

Several features of Apache Spark RDD are:

Features of Spark RDD

i. 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).

ii. 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. Follow this guide for the deep study of **Spark Lazy Evaluation.**

iii. 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. Follow this guide for the deep study of **RDD Fault Tolerance.**

iv. 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.

v. 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.

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

vii. Coarse-grained Operations
It applies to all elements in datasets through maps or filter or group by operation.

viii. 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.

6. Spark RDD Operations

RDD in Apache Spark supports two types of operations:

- Transformation
- Actions

i. Transformations

Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are

immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey() etc.

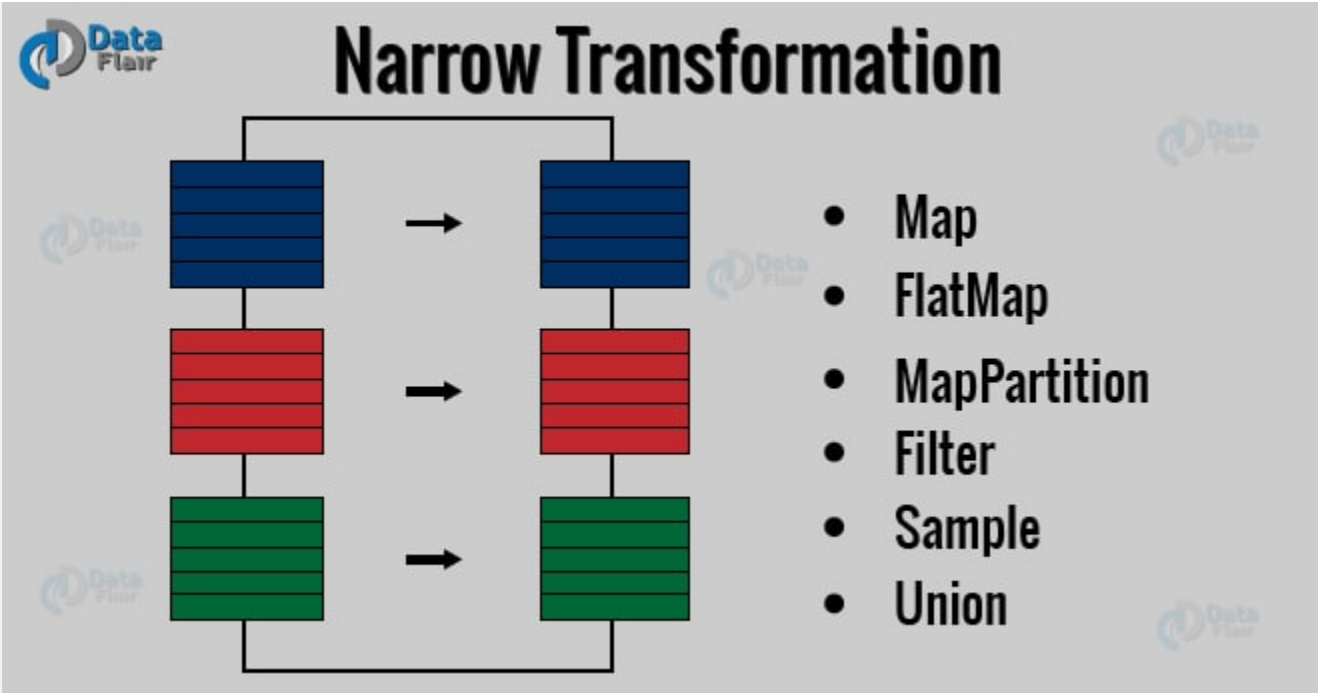
Transformations are **lazy** operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

Certain transformations can be pipelined which is an optimization method, that Spark uses to improve the performance of computations. There are two kinds of transformations: narrow transformation, wide transformation.

a. **Narrow** Transformations

It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

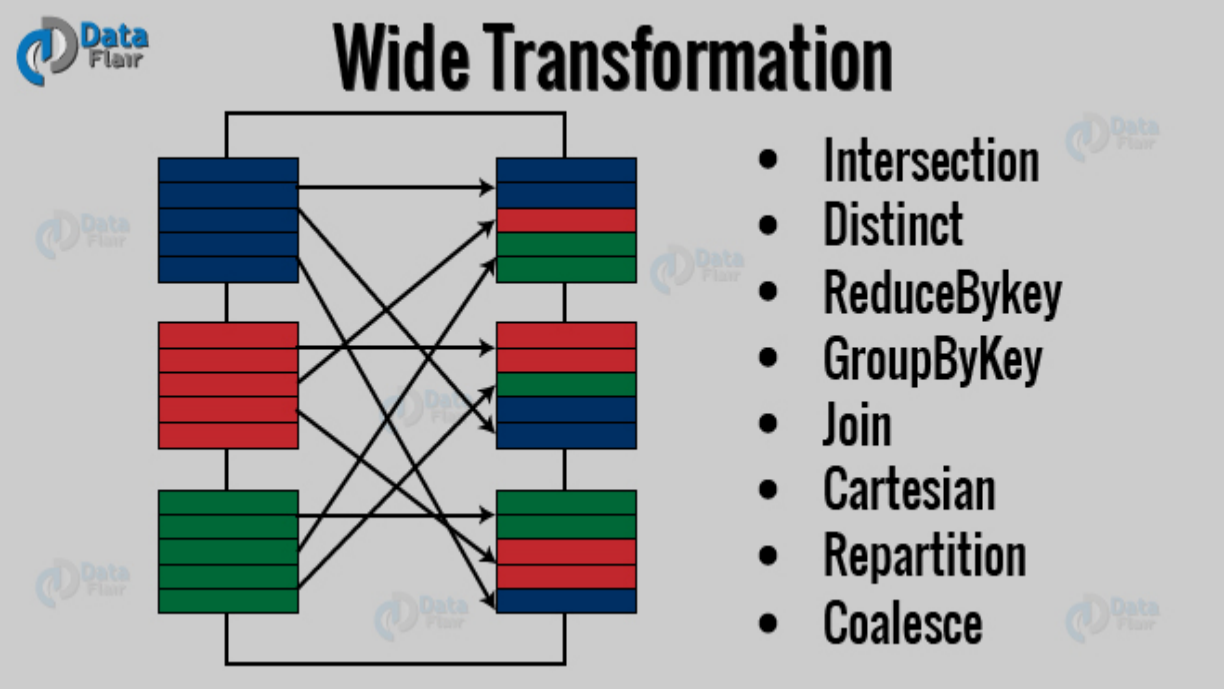
Spark groups narrow transformations as a stage known as **pipelining**.



Spark RDD – Narrow Transformation

b. Wide Transformations

It is the result of groupByKey() and reduceByKey() like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.



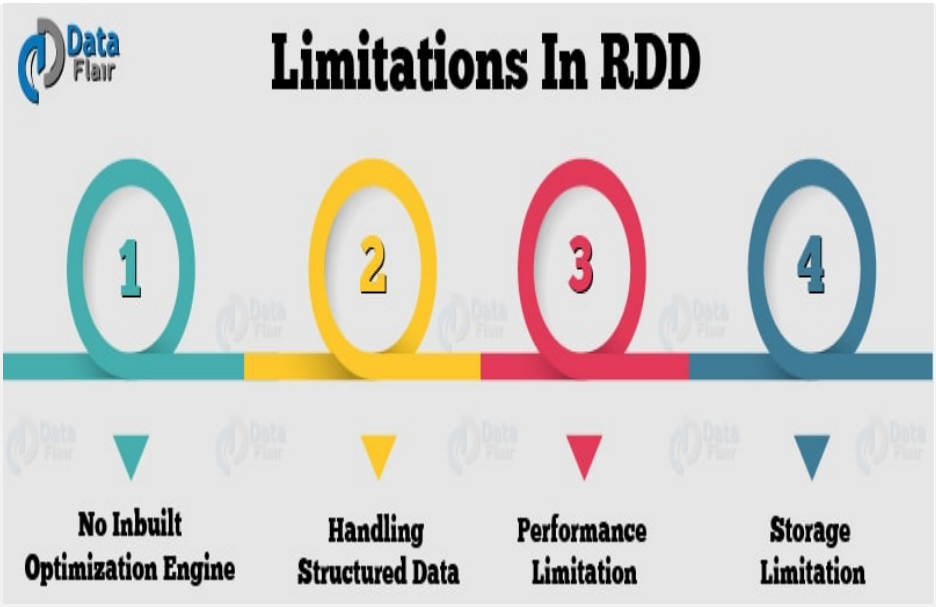
Wide Transformation

ii. Actions

An **Action** in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.
Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.
Using transformations, one can create RDD from the existing one. But when we want to work with the actual dataset, at that point we use Action. When the Action occurs it does not create the new RDD, unlike transformation. Thus, actions are RDD operations that give no RDD values. Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.

7. Limitation of Spark RDD

There is also some limitation of Apache Spark RDD. Let's discuss them one by one-



Limitations of Apache Spark RDD

i. No inbuilt optimization engine

When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including **catalyst optimizer** and **Tungsten execution engine**. Developers need to optimize each RDD based on its attributes.

ii. Handling structured data

Unlike **Dataframe** and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

iii. Performance limitation

Being in-memory JVM objects, RDDs involve the overhead of Garbage Collection and Java Serialization which are expensive when data grows.

iv. Storage limitation

RDDs degrade when there is not enough memory to store them. One can also store that partition of RDD on disk which does not fit in RAM. As a result, it will provide similar performance to current data-parallel systems.

So, this was all in Spark RDD Tutorial. Hope you like our explanation

8. Conclusion – Spark RDD

In conclusion to RDD, the shortcomings of Hadoop MapReduce was so high. Hence, it was overcome by Spark RDD by introducing in-memory processing, immutability etc. But there were some limitations of RDD. For example No inbuilt optimization, storage and performance limitation etc.
Because of the above-stated limitations of RDD to make spark more versatile DataFrame and Dataset evolved.

Question no. - 3) List down few Spark RDD operations and explain each of them.

Spark RDD Operations-Transformation & Action with Example

1. Spark RDD Operations

Two types of **Apache Spark** RDD operations are- Transformations and Actions.
A **Transformation** is a function that produces new **RDD** from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed. When the action is triggered after the result, new RDD is not formed like transformation. In this Apache Spark RDD operations tutorial we will get the detailed view of what is Spark RDD, what is the transformation in Spark RDD, various RDD transformation operations in Spark with examples, what is action in Spark RDD and various RDD action operations in Spark with examples.

Spark RDD Operations-Transformation & Action with Example

2. Apache Spark RDD Operations

Before we start with Spark RDD Operations, let us deep dive into **RDD in Spark.**
Apache Spark RDD supports two types of Operations-

- Transformations
- Actions

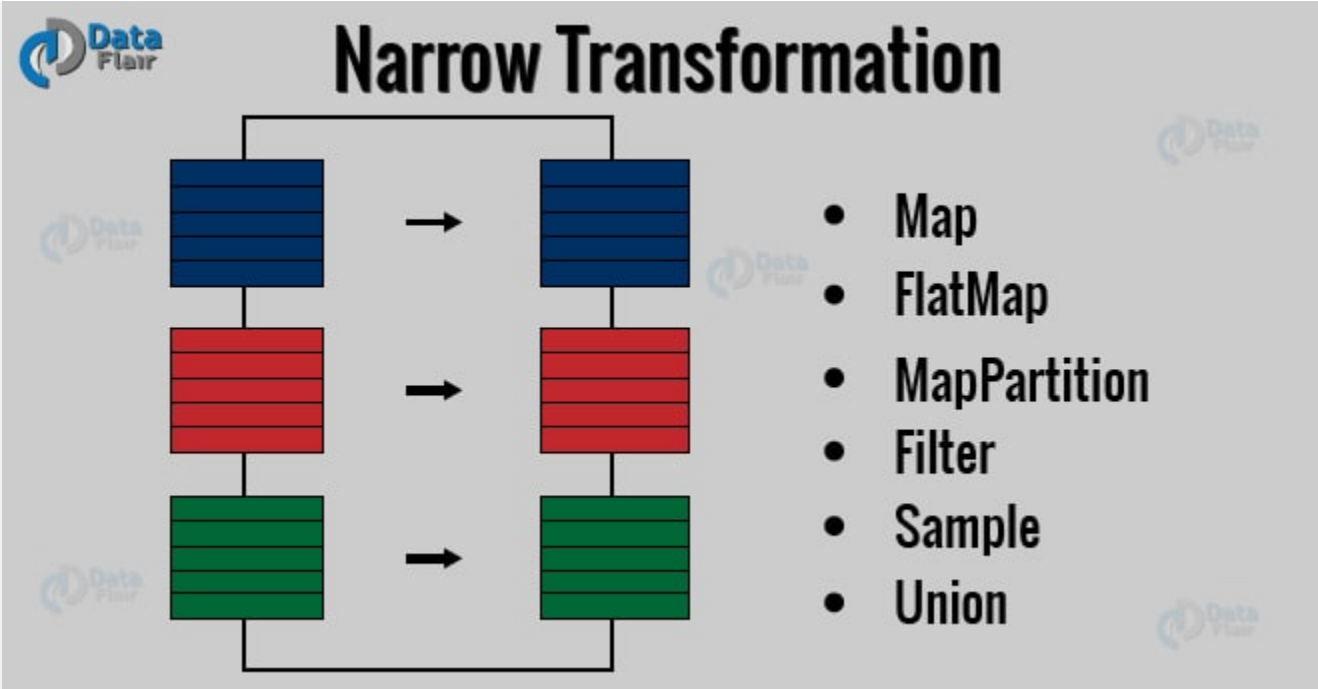
Now let us understand first what is Spark RDD Transformation and Action-

3. RDD Transformation

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.
Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as **RDD operator graph** or **RDD dependency**

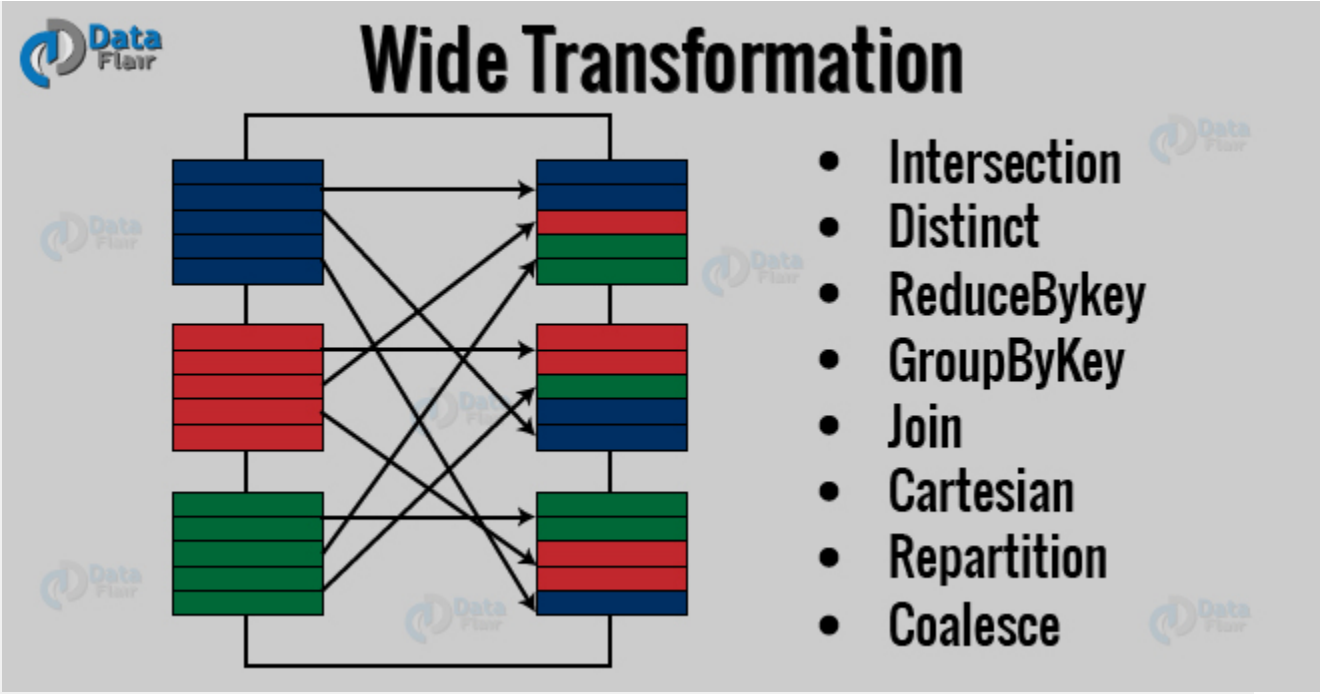
graph. It is a logical execution plan i.e., it is Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.
Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).
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().*



Apache Spark Narrow Transformation Operation

- **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()*.



Spark Wide Transformation Operations

There are various functions in RDD transformation. Let us see RDD transformation with examples.

3.1. 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.

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).

Also Read: **How to create RDD**
Map() example:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
object mapTest{
def main(args: Array[String]) = {
val spark = SparkSession.builder.appName("mapExample").master("local").getOrCreate()
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
mapFile.foreach(println)
}
}
```

spark_test.txt”

```
hello...user! this file is created to check the operations of spark.


?, and how can we apply functions on that RDD partitions?. All this will be done through spark programming which is done with the help of scala language support...
```

- ***Note –*** In above code, map() function map each line of the file with its length.

3.2. 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.

flatMap() example:

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

- ***Note –*** In above code, flatMap() function splits each line when space occurs.

3.3. 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, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

Filter() example:

```
1. val data = spark.read.textFile("spark_test.txt").rdd
2. val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
3. println(mapFile.count())
```

- ***Note*** – 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.

3.4. 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.

3.5. 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.

3.6. 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).

Union() 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)
```

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

3.7. 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).

Intersection() 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),(1,"jan",2016)))
val comman = rdd1.intersection(rdd2)
comman.foreach(Println)
```

- ***Note –*** The intersection() operation return a new RDD. It contains the intersection of elements in the rdd1 & rdd2.

3.8. 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).

Distinct() example:

```
val rdd1 = park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
```

```
2. val result = rdd1.distinct()
3. println(result.collect().mkString(", "))
```

3.9. 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.

groupByKey() example:

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

- ***Note –*** 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.

3.10. 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.

reduceByKey() example:

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

- ***Note –*** 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.

3.11. 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:

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

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

3.12. 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:

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

- ***Note*** – The join() transformation will join two different RDDs on the basis of Key.

3.13. 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.
Coalesce() example:

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

- ***Note –*** The coalesce will decrease the number of partitions of the source RDD to numPartitions define in coalesce argument.

4. RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. 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. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.
An action is one of the ways of sending data from *Executer* to the *driver.* Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

4.1. count()

Action **count()** returns the number of elements in RDD.
For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.count()" will give the result 8.

Count() example:

```
1. val data = spark.read.textFile("spark_test.txt").rdd
2. val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
3. println(mapFile.count())
```

- ***Note –*** In above code *flatMap()* function maps line into words and count the word "Spark" using *count()* Action after filtering lines containing "Spark" from mapFile.

4.2. 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. As a result, it makes easy to compare the result of RDD with the expected result.
Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

Collect() 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(","))
```

- ***Note –*** *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.

4.3. 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}

Take() 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)
```

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

4.4. 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.
Top() 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)
```

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

4.5. 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)}

countByValue() example:

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

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

4.6. 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.

Reduce() example:

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

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

4.7. 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).

Fold() 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)
```

- ***Note –*** In above code *additionalMarks* is an initial value. This value will be added to the int value of each record in the source RDD.

4.8. 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.

4.9. 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, inserting a record into the database.

Foreach() 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)
```

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

5. Conclusion

In conclusion, on applying a transformation to an RDD creates another RDD. As a result of this RDDs are immutable in nature. On the introduction of an action on an RDD, the result gets computed. Thus, this lazy evaluation decreases the overhead of computation and make the system more efficient.