****

**Spark**:It is an open-source big data framework. It provides a faster and more general-purpose data processing engine. Spark is basically designed for fast computation. It also covers a wide range of workloads — for example, batch, interactive, iterative, and streaming.

**Hadoop MapReduce**:It is also an open-source framework for writing applications. It also processes structured and unstructured data that are stored in HDFS. Hadoop MapReduce is designed in a way to process a large volume of data on a cluster of commodity hardware. MapReduce can process data in batch mode.

**Data Processing**

**Spark**:Apache Spark is a good fit for both batch processing and stream processing, meaning it’s a hybrid processing framework. **Spark speeds up batch processing via in-memory computation and processing optimization**. It’s a nice alternative for streaming workloads, interactive queries, and machine learning. Spark can also work with Hadoop and its modules. Its real-time data processing capability makes Spark a top choice for big data analytics.

Its resilient distributed dataset (RDD) allows Spark to transparently store data in-memory and send to disk only what’s important or needed. As a result, a lot of time that's spent on the disk read and write is saved.

**Hadoop**: Apache Hadoop provides batch processing. Hadoop develops a great deal in creating new algorithms and component stack to improve access to large scale batch processing.

MapReduce is Hadoop’s native batch processing engine. Several components or layers (like YARN, HDFS, etc.) in modern versions of Hadoop allow easy processing of batch data. Since MapReduce is about permanent storage, **it stores data on-disk, which means it can handle large datasets**. MapReduce is scalable and has proved its efficacy to deal with tens of thousands of nodes. However, Hadoop’s data processing is slow as MapReduce operates in various sequential steps.

**Real-Time Analysis**

**Spark**:It can process real-time data, i.e. data coming from real-time event streams at the rate of millions of events per second, such as Twitter and Facebook data. Spark’s strength lies in its ability to process live streams efficiently.

**Hadoop MapReduce**:MapReduce fails when it comes to real-time data processing, as it was designed to perform batch processing on voluminous amounts of data.

**Ease of Use**

**Spark**: Spark is easier to use than Hadoop, as it comes with user-friendly APIs for Scala (its native language), Java, Python, and Spark SQL. Since Spark provides a way to perform streaming, batch processing, and machine learning in the same cluster, users find it easy to simplify their infrastructure for data processing. An interactive [REPL](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop) (Read-Eval-Print Loop) allows Spark users to get instant feedback for commands.

**Hadoop**: Hadoop, on the other hand, is written in Java, is difficult to program, and requires abstractions. Although there is no interactive mode available with Hadoop MapReduce, tools like Pig and Hive make it easier for adopters to work with it.

**Graph Processing**

**Spark**: Spark comes with a graph computation library called GraphX to make things simple. In-memory computation coupled with in-built graph support allows the algorithm to perform much better than traditional MapReduce programs. Netty and Akka make it possible for Spark to distribute messages throughout the executors.

**Hadoop**: Most processing algorithms, like PageRank, perform multiple iterations over the same data. MapReduce reads data from the disk and, after a particular iteration, sends results to the HDFS, and then again reads the data from the HDFS for the next iteration. Such a process increases latency and makes graph processing slow.

In order to evaluate the score of a particular node, message passing needs to contain scores of neighboring nodes. These computations require messages from its neighbors, but MapReduce doesn’t have any mechanism for that. Although there are fast and scalable tools like Pregel and GraphLab for efficient graph processing algorithms, they aren't suitable for complex multi-stage algorithms.

**Fault Tolerance**

**Spark**:Spark uses RDD and various data storage models for fault tolerance by minimizing network I/O. In the event of partition loss of an RDD, the RDD rebuilds that partition through the information it already has. So, Spark does not use the replication concept for fault tolerance.

**Hadoop**:Hadoop achieves fault tolerance through replication. MapReduce uses TaskTracker and JobTracker for fault tolerance. However, TaskTracker and JobTracker have been replaced in the second version of MapReduce by Node Manager and ResourceManager/ApplicationMaster, respectively.

**Security**

**Spark**: Spark’s security is currently in its infancy, offering only authentication support through shared secret (password authentication). However, organizations can run Spark on HDFS to take advantage of HDFS ACLs and file-level permissions.

**Hadoop MapReduce**: Hadoop MapReduce has better security features than Spark. Hadoop supports Kerberos authentication, which is a good security feature but difficult to manage. Hadoop MapReduce can also integrate with Hadoop security projects, like Knox Gateway and Sentry. Third-party vendors also allow organizations to use Active Directory Kerberos and LDAP for authentication. Hadoop’s Distributed File System is compatible with access control lists (ACLs) and a traditional file permissions model.

**Cost**

Both Hadoop and Spark are open-source projects, therefore come for free. However, Spark uses large amounts of RAM to run everything in-memory, and RAM is more expensive than hard disks. Hadoop is disk-bound, so saves the costs of buying expensive RAM, but requires more systems to distribute the disk I/O over multiple systems.

As far as costs are concerned, organizations need to look at their requirements. If it’s about processing large amounts of big data, Hadoop will be cheaper since hard disk space comes at a much lower rate than memory space.

**Compatibility**

Both Hadoop and Spark are compatible with each other. Spark can integrate with all the data sources and file formats that are supported by Hadoop. So, it’s not wrong to say that Spark’s compatibility with data types and data sources is similar to that of Hadoop MapReduce.

Both Hadoop and Spark are scalable. One may think of Spark as a better choice than Hadoop. However, MapReduce turns out to be a good choice for businesses that need huge datasets brought under control by commodity systems. Both frameworks are good in their own sense. Hadoop has its own file system that Spark lacks, and Spark provides a way for real-time analytics that Hadoop does not possess.

Hence, the differences between Apache Spark vs. Hadoop MapReduce shows that **Apache Spark is much more advanced cluster computing engine than MapReduce**. Spark can handle any type of requirements (i.e. batch, interactive, iterative, streaming, graph) while MapReduce limits to batch processing.



**RDD:**

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. 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.

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 RDD**

**5.1. In-memory Computation**

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

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

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

**5.4. 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.

**5.5. 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.

**5.6. Persistence**

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

**5.7. Coarse-grained Operations**

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

**5.8. 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.



Apache Spark RDD supports two types of Operations-

* Transformations
* Actions

**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:

**1. 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().*

**2. 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)**

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()](http://data-flair.training/blogs/map-vs-flatmap-operation-in-apache-spark/) is map() returns only one element, while flatMap() can return a list of elements.

**flatMap() example:**

1. val data = spark.read.textFile("spark\_test.txt").rdd
2. val flatmapFile = data.flatMap(lines => lines.split(" "))
3. flatmapFile.foreach(println)

**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())

**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.

**Union() example:**

1. val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))
2. val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
3. val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
4. val rddUnion = rdd1.union(rdd2).union(rdd3)

**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:**

1. val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014, (16,"feb",2014)))
2. val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
3. val comman = rdd1.intersection(rdd2)
4. comman.foreach(Println)

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

**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:**

1. 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(", "))

* ***Note –*** In the above example, the distinct function will remove the duplicate record i.e. (3,'”nov”,2014).

**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](http://data-flair.training/blogs/apache-spark-rdd-persistence-caching/) in detail.

**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)

**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.

**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).

**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.

**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)

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

**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:

**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.

**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:**

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* –** *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.

**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:**

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

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

**Top() example:**

1. val data = spark.read.textFile("spark\_test.txt").rdd
2. val mapFile = data.map(line => (line,line.length))
3. val res = mapFile.top(3)
4. res.foreach(println)

**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.

**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.

**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:**

1. val rdd1 = spark.sparkContext.parallelize(List(("maths", 80),("science", 90)))
2. val additionalMarks = ("extra", 4)
3. val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc.\_2 + marks.\_2
4. ("total", add)
5. }
6. 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.

**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, inserting a record into the database.

**Foreach() 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)