Spark and Spark-SQL Interview Questions

**What is Apache Spark? What is the reason behind the evolution of this framework?**

**Ans.**[**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is an open source big data framework. It has an expressive APIs to allow big data professionals to efficiently execute streaming as well as the batch. Apache Spark provides faster and more general data processing platform engine. It basically designed for fast computation and developed at UC Berkeley in 2009. Spark is an Apache project which is also call as “lighting fast cluster computing“. It distributes data in a file system across the cluster, and process that data in parallel. Spark covers a wide range of workloads like batch applications, iterative algorithms, interactive queries and streaming. It lets you write an application in Java, Python or Scala.

It was developed to overcome the limitations of [**MapReduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) cluster computing paradigm. Spark keeps things in memory whereas map reduces keep shuffling things in and out of disk. It allows to cache data in memory which is beneficial in iterative algorithm those used in machine learning.

Spark is easier to develop as it knows how to operate on data. It supports SQL queries, streaming data as well as graph data processing. Spark doesn’t need Hadoop to run, it can run on its own using other storages like Cassandra, S3 from which spark can read and write. In terms of speed spark run programs up to 100x faster in memory or 10x faster on disk than Map Reduce.

**Explain the features of Apache Spark because of which it is superior to Apache MapReduce?**

**Ans.**[**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/) is designed for batch processing. Batch processing is very efficient in the processing of high volume data.  
Hadoop MapReduce is batch-oriented processing tool, it takes large dataset in the input, processes it and produces a result.  
Hadoop MapReduce adopted the batch-oriented model. Batch is essentially processing data at rest, taking a large amount of data at once and producing output. MapReduce process is slower than spark because due to produce a lot of intermediary data.

Spark also supports batch processing system as well as stream processing.

[**Spark streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/) processes data streams in micro-batches, Micro batches are an essentially collect and then process kind of  
computational model. Spark processes faster than map reduces because it caches input data in memory by [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/).

**Why is Apache Spark faster than Apache Hadoop?**

**Ans.** Apache Spark is faster than Apache Hadoop due to below reasons:

* Apache Spark provides [**in-memory computing**](http://data-flair.training/blogs/apache-spark-in-memory-computing/). Spark is designed to transform data In-memory and hence reduces time for disk I/O. While MapReduce writes intermediate results back to Disk and reads it back.
* Spark utilizes [**Direct Acyclic Graph**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) that helps to do all the optimization and computation in a single stage rather than multiple stages in the MapReduce model
* Apache Spark core is developed using [**SCALA**](http://data-flair.training/blogs/category/scala/) programming language which is faster than JAVA. SCALA provides inbuilt concurrent execution by providing immutable collections. While in JAVA we need to use Thread to achieve parallel execution.

**List down the languages supported by Apache Spark.**

**Ans.**Apache Spark supports Scala, Python, Java, and R.  
Apache Spark is written in Scala. Many people use Scala for the purpose of development. But it also has API in Java, Python, and R.

**Different type of APIs for accessing SparkSQL:**

**SQL:**Executing SQL queries or [**Hive**](http://data-flair.training/blogs/category/hive/) queries, the result is going to become in the variety of DataFrame.

1. **DataFrame:**It is similar to the relative table in SparkSQL and distributed the assortment of tabular information having rows and named column. It will perform filter, intersect, join, sort mixture and much more. DataFrames powerfully trusts [**options of RDD**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/). As it trusts RDD, it is [**lazy evaluated**](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/) and immutable in nature.  
   DataFrameAPI is offered in Scala, Java, and Python.
2. **Datasets API:**Dataset is new API to supply benefit of RDD because it is robust written and declarative in nature. Dataset is the assortment of object or records with the familiar schema. It should model in some data structure. DataSets API offers improvement of DataFrames and static kind safety of Scala. We can convert information set to Data Frame.

* [**Spark Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/)

Spark Streaming is a light-weight API that permits developers to perform execution and streaming of information application. Discretized Streams kind the bottom abstraction in Spark Streaming. It makes use of an endless stream of {input information|input file|computer file} to method data in the time period. It leverages the quick programming capability of Apache Spark core to perform streaming analytics by ingesting information in mini-batches. Information in Spark Streaming accepts from varied information sources and live streams like Twitter, Apache Kafka, IoT Sensors, Amazon response, Apache Flume, etc. in an event-driven, fault-tolerant, and type-safe applications.

* **Spark element MLlib**  
  MLlib in Spark stands for machine learning (ML) library. Its goal is to form sensible machine learning effective, ascendible and straightforward. It consists of some learning algorithms and utilities, as well as classification, regression, clustering, collaborative filtering, spatial property reduction, further as lower-level improvement primitives and higher-level pipeline genus Apis.
* **GraphX**  
  GraphX, a distributed graph process framework on prime of Apache Spark. because it predicate on RDDs, that square measure immutable, graphs square measure immutable and so GraphX is unsuitable for graphs that require being updated, in addition to in an exceedingly transactional manner sort of a graph info.
* **Is it possible to run Apache Spark without Hadoop?**
* **Ans.**Yes, Apache Spark can run without Hadoop, standalone, or in the cloud. Spark doesn’t need a Hadoop cluster to work. It can read and then process data from other file systems as well. [**HDFS**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) is just one of the file systems that Spark supports.
* Spark does not have any storage layer, so it relies on one of the distributed storage systems for distributed computing like HDFS, Cassandra etc.
* However, there are a lot of advantages to running Spark on top of Hadoop (HDFS (for storage) + [**YARN**](http://data-flair.training/blogs/category/yarn/) (resource manager)), but it’s not the mandatory requirement. It meant for distributed computing. In this case, the data distribute across the computers and Hadoop’s distributed file system HDFS is used to store data that does not fit in memory.
* One more reason for using Hadoop with Spark is they both are open source and both can integrate with each other rather easily as compared to other data storage system.

**What is RDD in Apache Spark? How are they computed in Spark? what are the various ways in which it can create?**

**Ans.**RDD in Apache Spark is the representation of a set of records, it is the immutable collection of objects with distributed computing. RDD is the large collection of data or an array of reference of partitioned objects. Each and every dataset in RDD is logically partitioned across many servers so that they can compute on different nodes of the cluster. [**RDDs are fault tolerant**](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/) i.e. self-recovered / recomputed in the case of failure. The dataset could data load externally by the users which can be in the form of JSON file, CSV file, text file or database via JDBC with no specific data structure.

[**RDD is Lazily Evaluated**](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/) i.e. it is memorized or called when required or needed, which saves lots of time. RDD is a read-only, partitioned collection of data. RDDs can creates through deterministic operations or on stable storage or from other RDDs. It can also generates by parallelizing an existing collection in your application or referring a dataset in an external storage system. It is cacheable. As it operates on data over multiple jobs in computations such as logistic regression, k-means clustering, PageRank algorithms, which makes it reuse or share data among multiple jobs.

**What are the features of RDD, that makes RDD an important abstraction of Spark?**

**Ans.**RDD (Resilient Distributed Dataset) is a basic abstraction in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/). Spark RDD is an immutable, partitioned collection of elements on the cluster which can operates in parallel.

**Each RDD is characterized by five main properties :**

Below operations are lineage operations.

* List or Set of partitions.
* List of dependencies on other (parent) RDD
* A function to compute each partition

Below operations are used for optimization during execution.

* Optional preferred location **[i.e. block location of an HDFS file] [it’s about data locality]**
* Optional partitioned info **[i.e. Hash-Partition for Key/Value pair –> When data shuffled how data will travel]**

Examples :

**# HadoopRDD:**HadoopRDD provides core functionality for reading data stored in Hadoop ([**HDFS**](http://data-flair.training/blogs/hdfs-data-read-operation/), [**HBase**](http://data-flair.training/blogs/category/hbase/), Amazon S3..) using the older [**MapReduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/)API (org.apache.hadoop.mapred)

Properties of HadoopRDD :

1. List or Set of partitions: One per HDFS block.
2. List of dependencies on parent RDD: None.
3. A function to compute each partition: read respective HDFS block
4. Optional Preferred location: HDFS block location
5. Optional partitioned info: None

**#FilteredRDD :**Properties of FilteredRDD:

1. List or Set of partitions: No. of partitions same as parent RDD\
2. List of dependencies on parent RDD: ‘one-to-one’ as the parent (same as parent)
3. A function to compute each partition: compute parent and then filter it
4. Optional Preferred location: None (Ask Parent)
5. Optional partitioned info: None

**List out the ways of creating RDD in Apache Spark.**

**Ans.**There are three ways to create [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/)

(1) By Parallelizing collections in the driver program

(2) By loading an external dataset

(3) Creating RDD from already existing RDDs.

**Create RDD By Parallelizing collections :**  
Parallelized collections are created by calling **parallelize**() method on an existing collection in driver program.

1. val rdd1 = **Array**(1,2,3,4,5)
2. val rdd2 = sc.**parallelize**(rdd1)

**OR**

1. val myList = sc.**parallelize**(**List**(1 to 1000), 5) where 5 is the number of partitions
2. [If we do not specify then default partition is 1

**Create by loading an external Dataset**

In Spark, the distributed dataset can form from any data source supported by Hadoop, including the local file system, HDFS, Cassandra, [**HBase**](http://data-flair.training/blogs/category/hbase/) etc. In this, the data is loaded from the external dataset. To create text file RDD, we can use SparkContext’s textFile method. It takes URL of the file and read it as a collection of line. URL can be a local path on the machine or a hdfs://, s3n://, etc. Use SparkSession.read to access an instance of DataFrameReader. DataFrameReader supports many file formats-

**i) csv (String path)**

1. import org.apache.spark.sql.SparkSession
2. def **main**(args: Array[String]):Unit = {
3. object DataFormat {
4. val spark = SparkSession.builder.**appName**("AvgAnsTime").**master**("local").**getOrCreate**()
5. val dataRDD = spark.read.**csv**("path/of/csv/file").rdd

**ii) json (String path)**

1. val dataRDD = spark.read.**json**("path/of/json/file").rdd

**iii) textFile (String path)**

1. val dataRDD = spark.read.**textFile**("path/of/text/file").rdd

**Creating RDD from existing RDD:**  
[**Transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) mutates one RDD into another RDD, thus transformation is the way to create an RDD from already existing RDD.

1. val words=spark.sparkContext.**parallelize**(**Seq**("the", "quick", "brown", "fox", "jumps", "over", "the", "lazy",
2. "dog"))
3. val wordPair = words.**map**(w => (w.**charAt**(0), w))
4. wordPair.**foreach**(println)

**Explain Transformation in RDD. How is lazy evaluation helpful in reducing the complexity of the System?**

**Ans.**Transformations are lazy evaluated operations on RDD that create one or many new RDDs, e.g. map, filter, reduceByKey, join, cogroup, randomSplit. Transformations are functions which take an RDD as the input and produces one or many RDDs as output. They don’t change the input RDD as RDDs are immutable and hence cannot change or modify but always produces new RDD by applying the computations operations on them. By applying transformations you incrementally build an RDD lineage with all the ancestor RDDs of the final RDD(s).

[**Transformations are lazy**](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/), i.e. are not executed immediately. Transformations can execute only when actions are called. After executing a transformation, the result RDD(s) will always be different from their ancestors RDD and can be smaller (e.g. filter, distinct, sample), bigger (e.g. flatMap, union, cartesian) or the same size (e.g. map) or it can vary in size.

RDD allows you to create dependencies b/w RDDs. Dependencies are the steps for producing results in a program. Each RDD in lineage chain, string of dependencies has a function for operating its data and has a pointer dependency to its ancestor RDD. Spark will divide RDD dependencies into stages and tasks and then send those to workers for execution.

**What are the types of Transformation in Spark RDD Operations?**

**Ans.**There are two kinds of transformations:

**Narrow transformations:**

Narrow transformations are the result of map, filter and in which data to transform id from a single partition only, i.e. it is self-sustained.  
An output RDD has partitions with records that originate from a single partition in the parent **RDD**.

**Wide Transformations**

Wide transformations are the result of groupByKey and reduceByKey. The data required to compute the records in a single partition may  
reside in many partitions of the parent RDD.

Wide transformations are also called shuffle transformations as they may or may not depend on a shuffle. All of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy these operations, **Spark**must execute RDD shuffle, which transfers data across the cluster and results in a new stage with a new set of partitions.

**What is the reason behind Transformation being a lazy operation in Apache Spark RDD? How is it useful?**

**Ans.**Whenever a **transformation operation** is performed in **Apache Spark**, it is lazily evaluated. It won’t execute until an action is performed. Apache Spark just adds an entry of the transformation operation to the [**DAG (Directed Acyclic Graph)**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) of computation, which is a directed finite graph with no cycles. In this DAG, all the operations are classified into different stages, with no shuffling of data in a single stage.

By this way, Spark can optimize the execution by looking at the DAG at its entirety, and return the appropriate result to the driver program.

For example, consider a 1TB of a log file in HDFS containing errors, warnings, and other information. Below are the operations being performed in the driver program:

1. [**Create an RDD**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) of this log file
2. It perform a flatmap() operation on this [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) to split the data in the log file based on tab delimiter.
3. Perform a filter() operation to extract data containing only error messages
4. Perform first() operation to fetch only the first error message.

If all the transformations in the above driver program are eagerly evaluated, then the whole log file will load into memory, all of the data within the file will split base on the tab, now either it needs to write the output of FlatMap somewhere or keep it in the memory. Spark needs to wait until the next operation is performed with the resource blocked for the upcoming operation. Apart from this for each and every operation spark need to scan all the records, like for FlatMap process all the records then again process them in filter operation.

On the other hand, if all the transformations are lazily evaluated, Spark will look at the DAG on the whole and prepare the execution plan for the application, now this plan will optimize the operation will combine/merge into stages then the execution will start. The optimized plan created by Spark improves job’s efficiency and overall throughput.

By this lazy evaluation in Spark, the number of switches between driver program and cluster is also reduced thereby saving time and resources in memory, and also there is an increase in the speed of computation.

**What is RDD lineage graph? How is it useful in achieving Fault Tolerance?**

**Ans.**The RDD Lineage Graph or RDD operator graph could be a graph of the entire parent [**RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/) of an RDD. It’s engineered as a result of materializing [**transformations to the RDD**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) and then creating a logical execution set up.

The RDDs in Apache Spark rely on one or a lot of alternative RDDs. The illustration of dependencies in between RDDs is understood because of the lineage graph. Lineage graph info is employed to cypher every RDD on demand, so whenever a district of persistent RDD is lost, {the data | the info | the info} that’s lost will recover using the lineage graph information.

**What is the FlatMap Transformation in Apache Spark RDD?**

**Ans.**FlatMap is a **transformation operation in Apache Spark**to **create an RDD** from existing **RDD**. It takes one element from an RDD and can produce 0, 1 or many outputs based on business logic. It is similar to Map operation, but Map produces one to one output. If we perform Map operation on an RDD of length N, output RDD will also be of length N. But for FlatMap operation output RDD can be of different length based on business logic

X——A x———–a  
Y——B y———–b,c  
Z——C z———–d,e,f

Map Operation FlatMap Operation

We can also say as flatMap transforms an RDD of length N into a collection of N collection, then flattens into a single RDD of results.

If we observe the below example data1 RDD which is the output of Map operation has same no of element as of data RDD,  
But data2 RDD does not have the same number of elements. We can also observe here as data2 RDD is a flattened output of data1 RDD

pranshu@pranshu-virtual-machine:~$ cat pk.txt

1 2 3 4

5 6 7 8 9

10 11 12

13 14 15 16 17

18 19 20

1. scala> val data = sc.**textFile**(“/home/pranshu/pk.txt”)

17/05/17 07:08:20 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes

data: org.apache.spark.rdd.RDD[String] = /home/pranshu/pk.txt MapPartitionsRDD[1] at textFile at <console>:24

1. scala> data.collect

res0: Array[String] = Array(1 2 3 4, 5 6 7 8 9, 10 11 12, 13 14 15 16 17, 18 19 20)

1. scala>
2. scala> val data1 = data.**map**(line => line.**split**(” “))

data1: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:26

1. scala>
2. scala> val data2 = data.**flatMap**(line => line.**split**(” “))

data2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at flatMap at <console>:26

1. scala>
2. scala> data1.collect

res1: Array[Array[String]] = Array(Array(1, 2, 3, 4), Array(5, 6, 7, 8, 9), Array(10, 11, 12), Array(13, 14, 15, 16, 17), Array(18, 19, 20))

1. scala>
2. scala> data2.collect

res2: Array[String] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

**Describe coalesce() operation. When can you coalesce to a larger number of partitions? Explain.**

**Ans.**It is a [**transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) and it’s in a package **org.apache.spark.rdd.ShuffledRDD**

**def coalesce(numPartitions: Int, shuffle: Boolean = false, partitionCoalescer: Option[PartitionCoalescer] = Option.empty)(implicit ord: Ordering[(K, C)] = null): RDD[(K, C)]**

Return a new [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) that is reduced into numPartitions partitions.

This results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead, each of the 100 new partitions will claim 10 of the current partitions.

However, if you’re doing a drastic coalesce, e.g. to numPartitions = 1, this may result in your computation taking place on fewer nodes than you like (e.g. one node in the case of numPartitions = 1). To avoid this, you can pass shuffle = true. This will add a shuffle step but means the current upstream partitions will execut in parallel (per whatever the current partitioning is).

Note: With shuffle = true, you can actually coalesce to a larger number of partitions. This is useful if you have a small number of partitions, say 100, potentially with a few partitions being abnormally large. Calling coalesce(1000, shuffle = true) will result in 1000 partitions with the data distributed using a hash partitioner.

Coalesce() operation changes a number of the partition where data is stored. It combines original partitions to a new number of partitions, so it reduces the number of partitions. Coalesce() operation is an optimized version of repartition that allows data movement, but only if you are decreasing the number of RDD partitions. It runs operations more efficiently after filtering large datasets.

**Example :**

1. val myrdd1 = sc.**parallelize**(1 to 1000, 15)
2. myrdd1.partitions.length
3. val myrdd2 = myrdd1.**coalesce**(5,false)
4. myrdd2.partitions.length
5. Int = 5

**Output :**

Int = 15

Int = 5

**Explain pipe() operation. How it writes the result to the standard output?**

**Ans. I**t is a transformation.

**def pipe(command: String): RDD[String]**

Return an RDD created by piping elements to a forked external process.

* In general, Spark is using Scala, Java, and Python to write the program. However, if that is not enough, and one want to pipe (inject) the data which written in other languages like ‘R’, Spark provides the general mechanism in the form of pipe() method.
* Spark provides the pipe() method on RDDs.
* With Spark’s pipe() method, one can write a transformation of an RDD that can read each element in the RDD from standard input as String.
* It can write the results as String to the standard output

**What is the key difference between textFile and wholeTextFile method?**

**Ans.**Both are the method of org.apache.spark.SparkContext.

**textFile() :**

* def textFile(path: String, minPartitions: Int = defaultMinPartitions): RDD[String]
* Read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings
* For example sc.textFile(“/home/hdadmin/wc-data.txt”) so it will create RDD in which each individual line an element.
* Everyone knows the use of textFile.

**wholeTextFiles() :**

* def wholeTextFiles(path: String, minPartitions: Int = defaultMinPartitions): RDD[(String, String)]
* Read a directory of text files from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI.
* Rather than create basic RDD, the wholeTextFile() returns pairRDD.
* For example, you have few files in a directory so by using wholeTextFile() method,  
  it creates pair RDD with a filename with a path as key,  
  and value is the whole file as a string

1. val myfilerdd = sc.**wholeTextFiles**("/home/hdadmin/MyFiles")
2. val keyrdd = myfilerdd.keys
3. keyrdd.collect
4. val filerdd = myfilerdd.values
5. filerdd.collec

**What is Action, how it process data in Apache Spark?**

**Ans. Actions** return final result of [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) computations/operation. It triggers execution using [**lineage graph**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) to load the data into original RDD, and carries out all intermediate transformations and returns final result to Driver program or write it out to file system.

**For example:** First, take, reduce, collect, count, aggregate are some of the actions in spark.

Action produces a value back to the [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) driver program. It may trigger a previously constructed, [**lazy RDD**](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/) to evaluate. It is an RDD operations that produce non-RDD values. Action function materializes a value in a Spark program. So basically an action is RDD operation that returns a value of any type but RDD[T] is an action. Actions are one of two ways to send data from executors to the driver (the other being accumulators)

**How is Transformation on RDD different from Action?**

**Ans. Transformations**[**create new RDD**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) from existing RDD  
Transformations are executed on demand.([**Lazy computation**](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/))  
Ex: filter(), union()

An **Action** will return a non-RDD type (your stored value types usually)  
Actions trigger execution using [**lineage graph**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) to load the data into original RDD  
Ex: count(), first()

**What are the ways in which one can know that the given operation is Transformation or Action?**

**Ans.**In order to identify the operation, one needs to look at the return type of an operation.

* **If the operation returns a new RDD, in that case, an operation is ‘Transformation’**
* **If the operation returns any other type than RDD, in that case, an operation is ‘Action’**

Hence, Transformation constructs a new RDD from an existing one (previous one) while Action computes the result based on applied transformation and returns the result to either driver program or save it to the external storage.

**Describe Partition and Partitioner in Apache Spark.**

**Ans.**Partition in Spark is similar to split in HDFS. A partition in Spark is a logical division of data stored on a node in the cluster. They are the basic units of parallelism in Apache Spark. RDDs are a collection of partitions. When some actions are executed, a task is launched per partition.

By default, partitions are automatically created by the framework. However, the number of partitions in Spark are configurable to suit the needs. For the number of partitions, if spark.default.parallelism is set, then we should use the value from SparkContext defaultParallelism, othewrwise we should use the max number of upstream partitions. Unless spark.default.parallelism is set, the number of partitions will be the same as that of the largest upstream RDD, as this would least likely cause out-of-memory errors.

A partitioner is an object that defines how the elements in a key-value pair RDD are partitioned by key, maps each key to a partition ID from 0 to numPartitions – 1. It captures the data distribution at the output. With the help of partitioner, the scheduler can optimize the future operations. The contract of partitioner ensures that records for a given key have to reside on a single partition.

We should choose a partitioner to use for a cogroup-like operations. If any of the RDDs already has a partitioner, we should choose that one. Otherwise, we use a default HashPartitioner.

There are three types of partitioners in Spark :

* Hash Partitioner
* Range Partitioner
* Custom Partitioner

**Hash – Partitioner:** Hash- partitioning attempts to spread the data evenly across various partitions based on the key.

**Range – Partitioner:** In Range- Partitioning method, tuples having keys with same range will appear on the same machine.

RDDs can create with specific partitioning in two ways :

i) Providing explicit partitioner by calling partitionBy method on an RDD

ii) Applying transformations that return RDDs with specific partitioners.

**How can you manually partition the RDD?**

**Ans.**When we create the [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) from a file stored in [**HDFS**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/).

1. data = context.**textFile**("/user/dataflair/file-name")

By default one partition is created for one block. ie. if we have a file of size 1280 MB (with 128 MB block size) there will be 10 HDFS blocks, hence the similar number of partitions (10) will create.

If you want to create more partitions than the number of blocks, you can specify the same while [**RDD creation**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/):

1. data = context.**textFile**("/user/dataflair/file-name", 20)

It will create 20 partitions for the file. ie for each block 2 partitions will create.

**NOTE:** It is often recommended to have more no of partitions than no of the block, it improves the performance

**Name the two types of shared variable available in Apache Spark.**

**Ans.**There are two types of shared variables available in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/):

* **Accumulators**: used to Aggregate the Information.
* **Broadcast variable**: to efficiently distribute large values.

When we pass the function to Spark, say filter(), this function can use the variable which defined outside of the function but within the Driver program but when we submit the task to Cluster, each worker node gets a new copy of variables and update from these variables not propagated back to Driver program.

Accumulators and Broadcast variable are used to remove above drawback ( i.e. we can get the updated values back to our Driver program)

**hat are accumulators in Apache Spark?**

**Ans.**This discussion is in continuation with a question, Name the two types of shared variable available in Apache Spark.

**Introduction of Accumulator :**

* An accumulator is a shared variable in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/), used to aggregating information across the cluster.
* In other words, aggregating information/values from worker nodes back to the driver program. ( How we will see in below session)

**Why Accumulator :**

* When we use a function inside the operation like map(), filter() etc these functions can use the variables which defined outside these function scope in the driver program.
* When we submit the task to cluster, each task running on the cluster gets a new copy of these variables and updates from these variable do not propagate back to the driver program.
* *Accumulator* lowers this restriction.

**Use Cases :**

* One of the most common uses of accumulator counts the events that occur during job execution for debugging purpose.
* Meaning count the no. of blank lines from the input file, no. of bad packets from a network during a session, during Olympic data analysis we have to find age where we said (age != ‘NA’) in SQL query in short finding bad/corrupted records.

**Examples :**

1. scala> val record = spark.read.**textFile**("/home/hdadmin/wc-data-blanklines.txt")

record: org.apache.spark.sql.Dataset[String] = [value: string]

1. scala> val emptylines = sc.**accumulator**(0) warning: there were two deprecation

warnings; re-run with -deprecation for details e

mptylines: org.apache.spark.Accumulator[Int] = 0

1. scala> val processdata = record.**flatMap**(x =>
2. {
3. **if**(x == "")
4. emptylines += 1
5. x.**split**(" ")
6. })

processdata: org.apache.spark.sql.Dataset[String] = [value: string]

1. scala> processdata.collect

16/12/02 20:55:15 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes

**Explain SparkContext in Apache Spark.**

**Ans.**A [**SparkContext**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/) is a client of Spark’s execution environment and it acts as the master of the [**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/)application. SparkContext sets up internal services and establishes a connection to a Spark execution environment. You can [**create RDDs**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/), accumulators and broadcast variables, access Spark services and run jobs (until SparkContext stops) after the creation of SparkContext. Only one SparkContext may be active per JVM. You must stop() the active SparkContext before creating a new one.

In Spark shell, a special interpreter-aware SparkContext is already created for the user, in the variable called sc.

The first step of any Spark driver application is to create a SparkContext. The SparkContext allows the Spark driver application to access the cluster through a resource manager. The resource manager can be [**YARN**](http://data-flair.training/blogs/category/yarn/), or [**Spark’s Cluster Manager**](http://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/).

**Few functionalities which SparkContext offers are:**

1. We can get the current status of a Spark application like configuration, app name.
2. We can set Configuration like master URL, default logging level.
3. One can create Distributed Entities like [**RDDs.**](http://data-flair.training/blogs/rdd-in-apache-spark/)

**Discuss the role of Spark driver in Spark application.**

**Ans.**The spark driver is that the program that defines the [**transformations and actions on RDDs**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) of knowledge and submits a request to the master. Spark driver is a program that runs on the master node of the machine which declares transformations and actions on knowledge RDDs.

In easy terms, the driver in Spark creates **SparkContext**, connected to a given Spark Master. It conjointly delivers the RDD graphs to Master, wherever the standalone cluster manager runs.

**What role does worker node play in Apache Spark Cluster? And what is the need to register a worker node with the driver program?**

**Ans.**[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) follows a master/slave architecture, with one master or driver process and more than one slave or worker processes

1. The master is the driver that runs the main() program where the spark context is created. It then interacts with the cluster manager to schedule the job execution and perform the tasks.
2. The worker consists of processes that can run in parallel to perform the tasks scheduled by the driver program. These processes are called executors.

Whenever a client runs the application code, the driver programs instantiates Spark Context, converts the [**transformations and actions**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) into logical[**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) of execution. This logical DAG is then converted into a physical execution plan, which is then broken down into smaller physical execution units. The driver then interacts with the cluster manager to negotiate the resources required to perform the tasks of the application code. The cluster manager then interacts with each of the worker nodes to understand the number of executors running in each of them.

**The role of worker nodes/executors:**

1. Perform the data processing for the application code
2. Read from and write the data to the external sources
3. Store the computation results in memory, or disk.

The executors run throughout the lifetime of the Spark application. This is a static allocation of executors. The user can also decide how many numbers of executors are required to run the tasks, depending on the workload. This is a dynamic allocation of executors.

Before the execution of tasks, the executors are registered with the driver program through the cluster manager, so that the driver knows how many numbers of executors are running to perform the scheduled tasks. The executors then start executing the tasks scheduled by the worker nodes through the cluster manager.

Whenever any of the worker nodes fail, the tasks that are required to perform will automatically allocates to any other worker nodes

**Discuss the various running mode of Apache Spark.**

**Ans.**We can launch spark application in four modes:

1) Local Mode (local[\*],local,local[2]…etc)

-> When you launch spark-shell without control/configuration argument, It will launch in local mode

spark-shell –master local[1]

-> spark-submit –class com.df.SparkWordCount SparkWC.jar local[1]

2) Spark Standalone cluster manger:

-> spark-shell –master spark://hduser:7077

-> spark-submit –class com.df.SparkWordCount SparkWC.jar spark://hduser:7077

3) Yarn mode (Client/Cluster mode):

-> spark-shell –master yarn or

(or)

->spark-shell –master yarn –deploy-mode client

Above both commands are same.

To launch spark application in cluster mode, we have to use a spark-submit command. We cannot run yarn-cluster mode via spark-shell because when we run spark application, driver program will be running as part application master container/process. So it is not possible to run cluster mode via spark-shell.

-> spark-submit –class com.df.SparkWordCount SparkWC.jar yarn-client

-> spark-submit –class com.df.SparkWordCount SparkWC.jar yarn-cluster

4) Mesos mode:

-> spark-shell –master mesos://HOST:5050

**Describe the run-time architecture of Spark.**

**Ans.**There are 3 important components of Runtime architecture of [Apache Spark](http://data-flair.training/forums/topic/explain-the-run-time-architecture-of-spark) as described below.

* Client process
* Driver
* Executor

**Responsibilities of the client process component**

The client process starts the driver program.

For example, the client process can be a spark-submit script for running applications, a spark-shell script, or a custom application using Spark API. The client process prepares the classpath and all configuration options for the Spark application.

It also passes application arguments, if any, to the application running on the driver.

**Responsibilities of the driver component**

The driver orchestrates and monitors the execution of a Spark application. There’s always one driver per Spark application.

The driver is like a wrapper around the application. The driver and its subcomponents (the Spark context and scheduler ) are responsible for:

* requesting memory and CPU resources from cluster managers
* breaking application logic into stages and tasks
* sending tasks to executors
* collecting the results

**Responsibilities of the executors**

The executors, which is a JVM processes, accept tasks from the driver, execute those tasks, and return the results to the driver. Each executor has several task slots (or CPU cores) for running tasks in parallel.

**What is SparkSession in Apache Spark? Why is it needed?**

**Ans.**Starting from **Apache Spark** 2.0, Spark Session is the new entry point for Spark applications.

Prior to 2.0, [**SparkContext**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/) was the entry point for spark jobs. [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) was one of the main APIs then, and it was created and manipulated using Spark Context. Every other APIs, different contexts were required – For SQL, SQL Context was required; For [**Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/), Streaming Context was required; For [**Hive**](http://data-flair.training/blogs/category/hive/), Hive Context was required.

But from 2.0, RDD along with DataSet and its subset [**DataFrame**](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/) APIs are becoming the standard APIs and are a basic unit of data abstraction in Spark. All of the user-defined code will be written and evaluated against the DataSet and DataFrame APIs as well as RDD.

So, there is a need for a new entry point build for handling these new APIs, which is why Spark Session has been introduced. Spark Session also includes all the APIs available in different contexts – Spark Context, SQL Context, Streaming Context, Hive Context.

**Explain API create Or Replace TempView().**

**Ans.**It’s basic Dataset function and under org.apache.spark.sql

* **def createOrReplaceTempView(viewName: String): Unit**
* **Creates a temporary view using the given name.**
* **The lifetime of this temporary view is tied to the SparkSession that was used to create this Dataset.**

**What are the various advantages of DataFrame over RDD in Apache Spark?**

**Ans. DataFrames** are the distributed collection of data. In DataFrame, data is organized into named columns. It is conceptually similar to a table in a relational database.  
we can construct DataFrames from a wide array of sources. Such as structured data files, tables in Hive, external databases, or existing RDDs.

As same as [**RDDs**](https://data-flair.training/forums/topic/what-are-the-advantages-of-dataframe-in-apache-spark), DataFrames are evaluated lazily([**Lazy Evaluation**](https://data-flair.training/forums/topic/what-are-the-advantages-of-dataframe-in-apache-spark)). In other words, computation only happens when an action (e.g. display result, save output) is required.

Out of the box, DataFrame supports reading data from the most popular formats, including JSON files, Parquet files, Hive tables. Also, can read from distributed file systems ([**HDFS**](https://data-flair.training/forums/topic/what-are-the-advantages-of-dataframe-in-apache-spark)), local file systems, cloud storage (S3), and external relational database systems through JDBC. In addition, through [**Spark SQL’s**](https://data-flair.training/blogs/spark-sql-tutorial/) external data sources API, DataFrames can extend to support any third-party data formats or sources. Existing third-party extensions already include Avro, CSV, ElasticSearch, and Cassandra.

There is much more to know about DataFrames. Refer link: [**Spark SQL DataFrame**](https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/)

**What is a DataSet? What are its advantages over DataFrame and RDD?**

**Ans.**In [**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/), Datasets are an extension of DataFrame API. It offers object-oriented programming interface. Through Spark SQL, it takes advantage of [**Spark’s Catalyst optimizer**](https://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/) by exposing e data fields to a query planner.

In [**SparkSQL**](https://data-flair.training/blogs/spark-sql-tutorial/), Dataset is a data structure which is strongly typed and is a map to a relational schema. Also, represents structured queries with encoders. DataSet has been released in Spark 1.6.

In serialization and deserialization (SerDe) framework, encoder turns out as a primary concept in Spark SQL. Encoders handle all translation process between JVM objects and Spark’s internal binary format. In Spark, we have built-in encoders those are very advanced. Even they generate bytecode to interact with off-heap data.

On-demand access to individual attributes without having to de-serialize an entire object is provided by an encoder. Spark SQL uses a SerDe framework, to make input-output time and space efficient. Due to encoder knows the schema of record, it became possible to achieve serialization as well as deserialization.

Spark Dataset is structured and lazy query expression([**lazy Evolution**](https://data-flair.training/blogs/apache-spark-lazy-evaluation/)) that triggers the action. Internally dataset represents a logical plan. The logical plan tells the computational query that we need to produce the data. the logical plan is a base catalyst query plan for the logical operator to form a logical query plan. When we analyze this and resolve we can form a physical query plan.

As Dataset introduced after [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) and [**DataFrame**](https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/), it clubs the features of both. It offers the following similar features:

1. The convenience of RDD.  
2. Performance optimization of DataFrame.  
3. Static type-safety of Scala.

Hence, we have observed that Datasets provides a more functional programming interface to work with structured data.

**On what all basis can you differentiate RDD, DataFrame, and DataSet?**

**Ans. DataFrame:**A Data Frame is used for storing data into tables. It is equivalent to a table in a relational database but with richer optimization. Spark DataFrame is a data abstraction and domain-specific language (DSL) applicable on a structure and semi-structured data. It is distributed the collection of data in the form of named column and row. It has a matrix-like structure whose column may be different types (numeric, logical, factor, or character ). We can say data frame has the two-dimensional array like structure where each column contains the value of one variable and row contains one set of values for each column and combines feature of list and matrices.

For more details about DataFrame, please refer: [**DataFrame in Spark**](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/)

**RDD**is the representation of a set of records, immutable collection of objects with distributed computing. RDD is a large collection of data or RDD is an array of reference of partitioned objects. Each and every dataset in RDD is logically partitioned across many servers so that they can compute on different nodes of the cluster. RDDs are fault tolerant i.e. self-recovered/recomputed in the case of failure. The dataset can load externally by the users which can be in the form of JSON file, CSV file, text file or database via JDBC with no specific data structure.

**DataSet** in Apache Spark, Datasets are an extension of DataFrame API. It offers object-oriented programming interface. Through Spark SQL, it takes advantage of Spark’s Catalyst optimizer by exposing e data fields to a query planner.

**Explain the Parquet File format in Apache Spark. When is it the best to choose this?**

**Ans.**Parquet is the columnar information illustration that is that the best choice for storing long run massive information for analytics functions. It will perform each scan and write operations with Parquet file. It could be a columnar information storage format.

Parquet, create to urge the benefits of compressed, economical columnar information illustration accessible to any project, despite the selection of knowledge process framework, data model, or programming language.

Parquet could a format which will process by a variety of various systems: [**Spark-SQL**](http://data-flair.training/blogs/spark-sql-tutorial/), Impala, [**Hive**](http://data-flair.training/blogs/category/hive/), Pig, niggard etc. It doesn’t lock into a particular programming language since the format is outlined exploitation, Thrift that supports numbers of programming languages. as an example, Aepyceros melampus is written in C++ whereas Hive is written in Java however they will simply interoperate on an equivalent Parquet information.

**Why does the picture of Spark come into existence?**

To overcome the drawbacks of **Apache Hadoop**, Spark came into the picture. Some of the drawbacks of Hadoop that Apache Spark overcomes are:

* Hadoop used only Java to build applications. Because it uses Java there were some security concerns as Java is prone to cyber crime.
* Apache Hadoop was apt only for batch processing. So, it does not support stream processing which was overcome in Spark.
* Hadoop used disk-based processing which results in slower retrieving of data. Spark overcomes this by in-memory computation.

**What are the features of Spark?**

Some of the features of Apache Spark are:

* The processing speed of Apache Spark is very high.
* Apache Spark is dynamic in nature. There are about 80 high-level operators, thus, using which we can build a parallel application.
* We can reuse code for join stream against historical data or for batch processing.
* Through [**RDD**](http://data-flair.training/blogs/apache-spark-rdd-tutorial/) we achieve fault tolerance. Thus, the data recovery is possible in RDD.
* Spark support many languages like **Java, Scala, Python, and**[**R**](http://data-flair.training/blogs/r-programming-tutorial/). Thus, makes it more user-friendly and is dynamic in nature.
* It can run independently and also on other cluster managers like Hadoop YARN.
* Apache Spark is cost effective solution for Big data problem. While Hadoop needs large storage and the large data center during replication.

**What are the limitations of Spark?**

* Does not have its file management system. Thus, it needs to integrate with Hadoop or other cloud-based data platforms.
* In-memory capability can become a bottleneck. Especially when it comes to cost-efficient processing of Bigdata.
* Memory consumption is very high. And the issues for the same are not handled in a user-friendly manner. d. It requires large data.
* MLlib lack in some available algorithms, for example, Tanimoto distance.

**List the languages supported by Apache Spark.**

Apache Spark Supports the following Languages: **Scala, Java, R, Python.**

**What are the cases where Apache Spark surpasses Hadoop?**

The data processing speed increases in the **Apache Spark**. This is because of the support of *in-memory computation* by the system. Thus, the performance of the system increase by 10x-1000x times. Apache Spark uses various languages for distributed application development.  
On the top of spark core, various libraries are present. These libraries enable workload that uses streaming, SQL, graph and machine learning. Some of these workloads are also supported by Hadoop. Spark facilitates the development by joining them into the same application. Apache Spark adopts micro-batching. Which is essentially used for handling near real time processing data model.

**Compare Hadoop and Spark.**

* **Cost Efficient –**In Hadoop, during replication, a large number of servers, huge amount of storage, and the large data center is required. Thus, installing and using Apache Hadoop is expensive. While using Apache Spark is a cost effective solution for big data environment.
* **Performance –**The basic idea behind Spark was to improve the performance of data processing. And Spark did this to 10x-100x times. And all the credit of faster processing in Spark goes to in-memory processing of data. In Hadoop, the data processing takes place in disc while in Spark the data processing takes place in memory. It moves to the disc only when needed. The Spark in-memory computation is beneficial for iterative algorithms. When it comes to performance, because of batch processing in Hadoop it’s processing is quite slow while the processing speed of Apache is faster as it supports micro-batching.
* **Ease of development –**The core in Spark is the distributed execution engine. Various languages are supported by Apache Spark for distributed application development. For example, Java, Scala, Python, and R. On the top of spark core, various libraries are built that enables workload. they make use of streaming, SQL, graph and machine learning. Hadoop also supports some of these workloads but Spark eases the development by combining all into the same application. d. Failure recovery: The method of Fault
* **Failure recovery –**The method of Fault Recovery is different in both Apache Hadoop and Apache Spark. In Hadoop after every operation data is written to disk. The data objects are stored in Spark in RDD distributed across data cluster. The RDDs are either in memory or on disk and provides full recovery from faults or failure.
* **File Management System –**Hadoop has its own File Management System called HDFS (Hadoop Distributed File System). While Apache Spark an integration with one, it may be even HDFS. Thus, Hadoop can run over Apache Spark.
* **Computation model –**Apache Hadoop uses batch processing model i.e. it takes a large amount of data and processes it. But Apache Spark adopts micro-batching. Must for handling near real time processing data model. When it comes to performance, because of batch processing in Hadoop it’s processing is quite slow. The processing speed of Apache is faster as it supports micro-batching.
* **Lines of code –**Apache Hadoop has near about 23, 00,000 lines of code while Apache Spark has 20,000 lines of code.
* **Caching –**By caching partial result in memory of distributed workers Spark ensures low latency computations. While MapReduce is completely disk oriented, there is no provision of caching.
* **Scheduler** – Because of in-memory computation in Spark, it acts as its own flow scheduler. While with Hadoop MapReduce we need an extra job scheduler like Azkaban or Oozie so that we can schedule complex flows.
* **Spark API –**Because of very Strict API in Hadoop MapReduce, it is not versatile. But since Spark discards many low-level details it is more productive.
* **Window criteria –**Apache Spark has time-based window criteria. But Apache Hadoop does not have window criteria since it does not support streaming.
* **Faster –**Apache Hadoop executes job 10 to 100 times faster than Apache Hadoop MapReduce.
* **License –**Both Apache Hadoop and Apache MapReduce has a License Version 2.0.
* **DAG() –**In Apache Spark, there is cyclic data flow in machine learning algorithm, which is a direct acyclic graph. While in Hadoop MapReduce data flow does not have any loops, rather it is a chain of the image.
* **Memory Management –**Apache Spark has automatic memory management system. While Memory Management in Apache Hadoop can be either statistic or dynamic.
* **Iterative Processing –**In Apache Spark, the data iterates in batches. Here processing and scheduling of each iteration are separate. While in Apache Hadoop there is no provision for iterative processing.
* **Latency** – The time taken for processing by Apache Spark is less as compared to Hadoop since it caches its data on memory by means of RDD, thus the latency of Apache Spark is less as compared to Hadoop.

**What are the components of Spark Ecosystem?**

The various components of Apache Spark are:

* **Spark Core –**Spark Core is the foundation of the whole project. All the functionality that is in Spark, is present on the top of Spark Core.
* **Spark Streaming –** It allows fault-tolerant streaming of live data streams. It is an add-on to core Spark API. Here it makes use of micro-batching for real-time streaming. Thus it packages live data into small batches and delivers to the batch system for processing.
* **Spark SQL –** Spark SQL component is distributed framework for structured data processing. Using Spark SQL Spark gets more information about the structure of data and the computation being performed. As a result, by using this information Spark can perform extra optimization.
* **Spark MLlib –** MLlib is a scalable learning library that discusses both: High-quality algorithm, High speed. The motive behind MLlib creation is to make machine learning scalable and easy. Thus. it contains machine learning libraries that have an implementation of various machine learning algorithms.
* **Spark GraphX** – GraphX is API for graphs and graph parallel execution. In order to support graph computation, graphX contains set of fundamental operators like sub graph, joinvertices and an optimized variant of Pregel API. Also, clustering, classification, traversal, searching, and pathfinding is possible in graphX.
* **SparkR –** SparkR is Apache Spark 1.4 release. The key component of SparkR is SparkR DataFrame. *Data frames* are a fundamental data structure for [**data processing in R**](http://data-flair.training/blogs/manipulating-and-processing-data-in-r/) and the concept of data frames extends to other languages with libraries like Pandas etc.

**What is Spark Core?**

**Spark Core** is a common execution engine for Spark platform. It provides parallel and distributed processing for large data sets. All the components on the top of it. Spark core provides speed through in-memory computation. And for ease of development, it also supports Java, Scala and Python APIs.  
RDD is the basic data structure of Spark Core. RDDs are immutable, a partitioned collection of record that can operate in parallel. We can [create RDDs](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) by transformation on existing RDDs. Also by loading an external dataset from stable storage like HDFS or[**HBase**](http://data-flair.training/blogs/hbase-tutorial-beginners-guide/), we can form RDD.

**What are the abstractions of Apache Spark?**

The main abstraction provided by Apache Spark is**Resilient Distributed Dataset**. RDDs are fault tolerant in nature. We cannot improve the changes made in RDD. RDDs creation starts with the file in a file system like Hadoop file system and then transforming it. The shared variable is the second abstraction provided by Apache Spark

**Explain the operations of Apache Spark RDD.**

Apache Spark RDD supports two types of operations: *Transformations and Actions-*

* **Transformations** are lazy operations on an RDD that create one or many new**RDDs**. For example *Map, filter, reduceByKey* etc creates new RDD. RDD create a new dataset from an existing one. That executes on demand. That means they compute lazily. Whenever we perform any transformation on RDD, it creates a new RDD each time.
* **Action** returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD. After application of all the intermediate transformation, it gives the final result to *driver program* or writes it out to file system. Upon applying Actions on an RDD non-RDD values gets generate.

**How many types of Transformation are there?**

There are two types of transformation namely *narrow transformation* and *wide transformation*.

* **Narrow Transformation** is the result of *map, filter* and such that the data is from a single partition only. As a result, the data is self-sustained. The RDD that we get as an output has a partition with records that originate from a single partition in the parent RDD.
* **Wide transformations** are the result of *groupByKey* and *reduceByKey.* The data that we will need to compute the records in a single partition is kept at many partitions of the parent RDD.

**In how many ways RDDs can be created? Explain.**

There are three ways to create an RDD:

* **Parallelized collection –** In the initial stages, the RDD is generally created by parallelized collection. In this method, we take the existing collection in the program and pass it to *parallelize()* method of *SparkContext.* The thing that should be noticed in the parallelized collection is the number of partition the dataset is cut into. For each partition of the cluster, Spark will run one task. Spark set a number of partition based on our cluster. But the number of partitions can also be set manually. Pass the number of partition as the second parameter for manual partition. e.g. sc.parallelize(data, 20), here we have manually given a number of partition as 20.
* **External Datasets (Referencing a dataset) –** In Spark one can create distributed dataset from any data source supported by Hadoop. For example the local file system, HDFS, Cassandra, HBase etc. In this, the data is loaded from the external dataset. To create text file RDD we can use [**SparkContext**](http://data-flair.training/blogs/learn-apache-spark-sparkcontext/) textFile method. It takes URL of the file and read it as a collection of line. URL can be a local path on the machine or a hdfs://, s3n://, etc.
* **Creating RDD from existing RDD –** Transformation converts one RDD into another RDD. By using transformation we can create an RDD from existing RDD. Transformation acts as a function that intakes an RDD and produces one.

**What are Paired RDD?**

***Paired RDDs*** are the RDD-containing *key-value pair*. A key-value pair (KYP) contains two linked data item. Here *Key* is the identifier and*Value* are the data corresponding to the key value.

**What is meant by in-memory processing in Spark?**

In **in-memory computation**, we keep data in random access memory in place of some slow disk drives. The processing of data is in parallel. Using this we can also identify the pattern, analyze large data Spark offers in in-memory capabilities. As a result, this increases the processing speed because it retrieves the data from memory in place of the disk. Also, the execution time of the process decreases. Keeping the data in-memory improves the performance by the order of magnitudes.  
The main abstraction of Spark is its **RDDs**. Also, we can cache RDD using the***cache()*** or ***persist()***method. In*cache()* method all the RDD are in-memory. The dissimilarity between cache() and persist() is the default storage level. For cache() it is MEMORY\_ONLY. While in *persist()* there are various storage levels like:

* MEMORY\_ONLY,
* MEMORY\_AND\_DISK,
* MEMORY\_ONLY\_SER
* MEMORY\_AND\_DISK\_SER
* DISK\_ONLY
* **How is fault tolerance achieved in Apache Spark?**
* The basic fault-tolerant semantic of Spark are:
* Since all RDD is an immutable data set. Each RDD keeps track of the lineage of the deterministic operation that employee on fault-tolerant input dataset to create it.
* If any partition of an RDD is lost due to a worker node failure, then that partition can be re-computed from the original fault-tolerant dataset using the lineage of operations.
* Assuming that all of the RDD transformations are deterministic, the data in the final transformed RDD will always be the same irrespective of failures in the [Spark cluster.](http://data-flair.training/blogs/install-apache-spark-multi-node-cluster/)
* To achieve fault tolerance for all the generated RDDs, the achieved data replicates among multiple Spark executors in worker node in the cluster. This result in two types of data that should recover in the event of failure:
* **Data received and replicated –** In this, the data replicates on one of the other nodes. Thus we can retrieve data when a failure occurs.
* **Data received but buffered for replication –** the data does not replicate. Thus the only way to recover fault is by retrieving it again from the source.
* Failure can also occur in worker and driver nodes.
* **Failure of worker node –** The node which runs the application code on the cluster is worker node. These are the slave nodes. Any of the worker nodes running executor can fail, thus resulting in loss of in-memory data. If any receivers were running on failed nodes, then their buffer data will vanish.
* **Failure of driver node –** If the driver node running the Spark Streaming application fails, then there is the loss of [**SparkContent**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/). All executors along with their in-memory data vanishes.
* **What is Directed Acyclic Graph(DAG)?**
* **RDDs** are formed after every transformation. At high level when we apply action on these RDD, Spark creates a**DAG**. DAG is a finite directed graph with *no directed cycles*.
* There are so many vertices and edges, where each edge is directed from one vertex to another. It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence. It is a strict generalization of [MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) model. DAG lets you get into the stage and expand in detail on any stage.
* In the stage view, the details of all RDDs that belong to that stage are expanded.

**What is lineage graph?**

***Lineage graph*** refers to the graph that has all the parent RDDs of an RDD. It is the result of all the transformation on the RDD. It creates a logical execution plan.

A *logical execution plan* is a plan that starts with the very first RDD. Also, it does not have any dependency on any RDD. It then ends at the RDD which produces the result of an action that has been called to execute.

**What is lazy evaluation in Spark?**

The **lazy evaluation** known as*call-by-need* is a strategy that delays the execution until one requires a value. The *transformation* in Spark is lazy in nature. Spark evaluate them lazily. When we call some operation in RDD it does not execute immediately; Spark maintains the graph of which operation it demands. We can execute the operation at any instance by calling the action on the data. The data does not loads until it is necessary.

**What are the benefits of lazy evaluation?**

Using lazy evaluation we can:

* Increase the manageability of the program.
* Saves computation overhead and increases the speed of the system.
* Reduces the time and space complexity.
* provides the optimization by reducing the number of queries.

**What do you mean by Persistence?**

***RDD persistence*** is an optimization technique which saves the result of RDD evaluation. Using this we save the intermediate result for further use. It reduces the computation overhead. We can make persisted RDD through***cache()*** and ***persist()***methods. It is a key tool for the interactive algorithm. Because, when RDD is persisted each node stores any partition of it that it computes in memory.

**Explain various level of persistence in Apache Spark.**

The persist method allows seven storage level:

* **MEMORY\_ONLY –**Store RDD as deserialized Java objects. If the RDD does not fit in memory, then some partitions will not be cached and will recompute on the fly each time needed. This is the default level.
* **MEMORY\_AND\_DISK –**Store RDD as deserialized Java objects. If the RDD does not fit in memory, store the partitions that don’t fit on the disk, and read them from there when they’re needed.
* **MEMORY\_ONLY\_SER (Java and Scala) –**Store RDD as serialized Java objects. This is more space-efficient than deserialized objects. especially when using a fast serializer. but it is hard for CPU to read.
* **MEMORY\_AND\_DISK\_SER(Java and Scala) –**Like MEMORY\_ONLY\_SER, but spills partitions that don’t fit in memory to disk.
* **DISK\_ONLY –**It stores the RDD partitions only on disk.
* **MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2 –**It replicates each partition on two cluster nodes.
* **OFF\_HEAP –**Like MEMORY\_ONLY\_SER, but store the data in off-heap memory. This requires enabling of off-heap memory.

**Explain various cluster manager in Apache Spark?**

The various cluster manages supported by Apache Spark are Standalone, Hadoop YARN, Apache Mesos.

* **Standalone Cluster Manager –** Standalone is a simple cluster manager of Spark that makes it easy to setup a cluster. In many cases, it is the simplest way to run Spark application in a clustered environment. It has masters and number of workers with the configured amount of memory and CPU cores. In standalone cluster mode, Spark allocates resources based on the core. It has the constraints that only one executor can be allocated on each worker per application.
* **Hadoop YARN –** YARN became the sub-project of Hadoop in the year 2012. The key idea behind YARN is to bifurcate the functionality of resource manager and job scheduling into different daemons. The plan is to have a Global [**Resource Manager (RM)**](http://data-flair.training/blogs/hadoop-yarn-resource-manager-guide-tutorial/)and per-application ***Application Master (AM)***. An application is either a **DAG** of graphs or an individual job. The data computation framework is a combination of the ResourceManager and the [**NodeManager**](http://data-flair.training/blogs/hadoop-yarn-node-manager-tutorial-guide/).
* **Apache Mesos –** Apache Mesos handles the workload in a distributed environment.It is healthful for deployment and management of applications in large-scale cluster environments. Mesos clubs together the existing resource of the machines/nodes into a cluster, as a result from this a variety of workloads may utilize. This is known as node abstraction, thus it decreases an overhead of allocating a specific machine for different workloads. It is resource management platform for Hadoop and [**Big Data**](http://data-flair.training/blogs/big-data-history-use-cases/) cluster. In some way, Apache Mesos is the reverse of virtualization. Because in virtualization one physical resource divides into multiple virtual resources, while in Mesos multiple physical resources groups into a single virtual resource.

**Define Partition in Apache Spark.**

Partition refers to, a logical block of large distributed Dataset. Logically partitioning the data and distributing it over the cluster provides parallelism. It also minimizes network traffic for sending data between executors. It determines how to access the entire hardware resources during job execution. RDD is automatically partitioned in Spark. We can change the size and number of the partition.

**What are shared variables?**  
***Shared variables*** are one of the abstractions of Apache Spark. Shared variables can be used in parallel operations.

Whenever Spark runs a function in parallel as a set of tasks on different nodes, each variable that is used in function are circulated to each task. Sometimes there is a need to share the variables across the tasks or between the task and the driver program.

Apache Spark supports two types of shared variables namely ***broadcast variable*** and ***accumulator.***  
Using broadcast variables we cache a value in memory on all nodes while we add accumulators to, such as counters and sums.

**What is the difference between DSM and RDD?**

**a) READ**

* **RDD:** In RDD the read operation is *coarse grained or fine grained*. In **coarse-grained** we can transform the whole dataset but not an individual element. While in**fine-grained** we do the transformation of an individual element on a dataset.
* **Distributed Shared Memory:** The read operation in Distributed shared memory is fine-grained.

**b) Write:**

* **RDD:** The write operation is coarse-grained in RDD.
* **Distributed Shared Memory:** In distributed shared system the write operation is fine grained.

**c) Consistency:**

* **RDD:** The consistency of RDD is trivial meaning it is immutable in nature. Any changes made to an RDD cannot roll back, it is permanent. So the level of consistency is high.
* **Distributed Shared Memory:** The system guarantees that if the programmer follows the rules, the memory will be consistent. It also guarantees that the results of memory operations will be predictable.

**d) Fault-recovery mechanism:**

* **RDD:** Using lineage graph at any point in time we can easily find the lost data in an RDD.
* **Distributed Shared Memory:** Fault tolerance is achieved by a checkpointing technique. It allows applications to roll back to a recent checkpoint rather than restarting.

**e) Straggler mitigation:** Stragglers, in general, are those tasks that take more time to complete than their peers.

* **RDD:** in RDD it is possible to mitigate stragglers using backup task.
* **Distributed Shared Memory:** It is quite difficult to achieve straggler mitigation.

**f) Behavior if not enough RAM:**

* **RDD:** If there is not enough space to store RDD in RAM then the RDDs are shifted to disk.
* **Distributed Shared Memory:** In this type of system the performance decreases if the RAM runs out of storage.

**How can data transfer be minimized when working with Apache Spark?**

By minimizing data transfer and avoiding shuffling of data we can increase the performance. In Apache Spark, we can minimize the data transfer in three ways:

* **By using a broadcast variable –** Since broadcast variable increases the efficiency of joins between small and large RDDs. the broadcast variable allows keeping a read-only variable cached on every machine in place of shipping a copy of it with tasks. We create broadcast variable v by calling *SparlContext.broadcast(v)* and we can access its value by calling the value method.
* **Using Accumulator –** Using accumulator we can update the value of a variable in parallel while executing. Accumulators can only be added through the associative and commutative operation. We can also implement counters (as in MapReduce) or sums using an accumulator. Users can create named or unnamed accumulator. We can create numeric accumulator by calling *SparkContext.longAccumulator()* or *SparkContext.doubleAccumulator()* for Long or Double respectively.
* By avoiding operations like ByKey, repartition or any other operation that trigger shuffle. we can minimize the data transfer.

**How does Apache Spark handles accumulated Metadata?**

By triggering automatic cleanup Spark handles the automatic Metadata. We can trigger cleanup by setting the parameter “***spark.cleaner.ttl***“. the default value for this is infinite. It tells for how much duration Spark will remember the metadata. It is periodic cleaner. And also ensure that metadata older than the set duration will vanish. Thus, with its help, we can run Spark for many hours.

**What are the common faults of the developer while using Apache Spark?**

The common mistake by developers are:

* Customer hit web-service several time by using multiple clusters.
* Customer runs everything on local node instead of distributing it.

**Which among the two is preferable for the project- Hadoop MapReduce or Apache Spark?**

The answer to this question depends on the type of project one has. As we all know Spark makes use of a large amount of RAM and also needs a dedicated machine to provide an effective result. Thus the answer depends on the project and the budget of the organization.

**What is Spark.executor.memory in a Spark Application?**

The default value for this is 1 GB. It refers to the amount of memory that will be used per executor process.

**What is DataFrames?**

It is a collection of data which organize in named columns. It is theoretically equivalent to a table in relational database. But it is more optimized. Just like RDD, DataFrames evaluates lazily. Using lazy evaluation we can optimize the execution. It optimizes by applying the techniques such as bytecode generation and predicate push-downs.

**What are the advantages of DataFrame?**

1. It makes large data set processing even easier. Data Frame also allows developers to impose a structure onto a distributed collection of data. As a result, it allows higher-level abstraction.
2. Data frame is both space and performance efficient.
3. It can deal with both structured and unstructured data formats, for example, Avro, CSV etc . And also storage systems like HDFS, HIVE tables, MySQL, etc.
4. The DataFrame API’s are available in various programming languages. For example Java, Scala, Python, and R.
5. It provides Hive compatibility. As a result, we can run unmodified Hive queries on existing Hive warehouse.
6. Catalyst tree transformation uses DataFrame in four phases: a) Analyze logical plan to solve references. b) Logical plan optimization c) Physical planning d) Code generation to compile part of the query to Java bytecode.
7. It can scale from kilobytes of data on the single laptop to petabytes of data on the large cluster.

**What is DataSet?**

[**Spark Datasets**](http://data-flair.training/blogs/apache-spark-dataset-tutorial/) are the extension of Dataframe API. It creates object-oriented programming interface and type-safety. Dataset is Spark 1.6 release. It makes use of Spark’s catalyst optimizer. It reveals expressions and data fields to a query optimizer. Dataset also influences fast in-memory encoding. It also provides provision for compile time type-safety. We can check for errors in an application when they run.

**What are the advantages of DataSets?**

* It provides run-time type safety.
* Influences fast in-memory encoding.
* It provides a custom view of structured and semi-structured data.
* It owns rich semantics and an easy set of domain-specific operations, as a result, it facilitates the use of structured data.
* Dataset API decreases the use of memory. As Spark knows the structure of data in the dataset, thus it creates an optimal layout in memory while caching.

**Explain Catalyst framework.**

The **Catalyst** is a framework which represents and manipulate a DataFrame graph. Data flow graph is a tree of relational operator and expressions. The three main features of catalyst are:

* It has a TreeNode library for transforming tree. They are expressed as Scala case classes.
* A logical plan representation for relational operator.
* Expression library.

The *TreeNode* builds a query optimizer. It contains a number of the query optimizer. **Catalyst Optimizer** supports both *rule-based* and *cost-based optimization*. In rule-based optimization the optimizer use set of rule to determine how to execute the query. While the cost based optimization finds the most suitable way to carry out SQL statement. In cost-based optimization, many plans are generates using rules. And after this, it computes their cost. Catalyst optimizer makes use of standard features of Scala programming like pattern matching.

**List the advantage of Parquet files.**

* It is efficient for large scale queries.
* It supports various efficient compression and encoding Scheme.
* It consumes less space