Spark\_Interview\_Question\_Part-2

**What is the need of Apache Spark?**

Basically, we had so many general purpose cluster computing tools. For example [Hadoop MapReduce](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/), Apache Storm, Apache Impala, Apache Storm, Apache Giraph and many more. But each one has some limitations in their functionality as well. Such as:

1. Hadoop MapReduce can only allow for batch processing.  
2. If we talk about stream processing only Apache Storm / S4 can perform.  
3. Again for interactive processing, we need Apache Impala / Apache Tez.  
4. While we need to perform graph processing, we opt for Neo4j / Apache Giraph.

Therefore, No single engine can perform all the tasks together. hence there was a big demand for a powerful engine that can process the data in real-time (streaming) as well as in batch mode  
Also, which can respond to sub-second and perform [in-memory processing](https://data-flair.training/blogs/apache-spark-in-memory-computing/)  
.

In this way, Apache Spark comes in picture. It is a powerful open-source engine that offers interactive processing, real-time [stream processing](https://data-flair.training/blogs/apache-spark-streaming-tutorial/), graph processing, in-memory processing as well as batch processing. Even with very fast speed, ease of use and also standard interface at the same time.

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

**Apache spark consists of following components**  
1.Spark Core  
2.Spark SQL  
3.Spark Streaming  
4.MLlib  
5.GraphX

**Spark Core:** Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines [resilient distributed datasets (RDDs)](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/), which are Spark’s main programming abstraction.It also provides many APIs for building and manipulating these RDDS.

**Spark SQL:** Spark SQL provides an interface to work with structured data.It allows querying in SQL as well as Apache [Hive](https://data-flair.training/blogs/apache-hive-tutorial/)variant of SQL(HQL).It supports many sources.

**Spark Streaming:** It is spark component that enables processing of live streams of data.

**MLlib:** Spark comes with common machine learning package called MLlib

**GraphX:** GraphX is a library for manipulating graphs (e.g., a social network’s friend graph)and performing graph-parallel computations.

**What is Spark Core?**

Spark Core is the fundamental unit of the whole Spark project. It provides all sort of functionalities like task dispatching, scheduling, and input-output operations etc.Spark makes use of Special data structure known as [RDD (Resilient Distributed Dataset)](http://data-flair.training/blogs/rdd-in-apache-spark/). It is the home for API that defines and manipulate the RDDs. Spark Core is distributed execution engine with all the functionality attached on its top. For example, MLlib, [SparkSQL](http://data-flair.training/blogs/spark-sql-tutorial/), GraphX, [Spark Streaming](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/). Thus, allows diverse workload on single platform. All the basic functionality of Apache Spark Like [in-memory computation](http://data-flair.training/blogs/apache-spark-in-memory-computing/)**,**[fault tolerance](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/), memory management, monitoring, task scheduling is provided by Spark Core.  
Apart from this Spark also provides the basic connectivity with the data sources. For example, [HBase](http://data-flair.training/blogs/category/hbase/), Amazon S3, [HDFS](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/)etc.

**Which all languages Apache Spark supports?**

Apache Spark is written in Scala language. Spark provides an API in Scala, Python, and Java in order to interact with Spark. It also provides APIs in R language.

**How is Apache Spark better than Hadoop?**

[Apache Spark](https://data-flair.training/blogs/apache-spark-for-beginners/) is lightening fast cluster computing tool. It is up to 100 times faster than [Hadoop MapReduce](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/) due to its very fast in-memory data analytics processing power.  
Apache Spark is a [Big Data](https://data-flair.training/blogs/what-is-big-data/) Framework. Apache Spark is a general purpose data processing engine and is generally used on top of HDFS. Apache Spark is suitable for the variety of data processing requirements ranging from Batch Processing to Data Streaming.

[Hadoop](https://data-flair.training/blogs/hadoop-tutorial-for-beginners/) is an open source framework which processes data stored in [HDFS](https://data-flair.training/blogs/hadoop-hdfs-tutorial/). Hadoop can process structured, unstructured or semi-structured data. Hadoop MapReduce can process the data only in Batch mode.

Apache Spark surpasses Hadoop in many cases such as  
1. Processing the data in memory which is not possible in Hadoop  
2. Processing the data that is in batch, iterative, interactive & [streaming](https://data-flair.training/blogs/apache-spark-streaming-tutorial/) i.e. Real Time mode. Whereas Hadoop processes only in batch mode.  
3. Spark is faster because it reduces the number of disk read-write operations due to its virtue of storing intermediate data in memory. Whereas in Hadoop MapReduce intermediate output which is output of Map() is always written on local hard disk  
4. Apache Spark is easy to program as it has hundreds of high-level operators with [RDD (Resilient Distributed Dataset)](https://data-flair.training/blogs/apache-spark-rdd-tutorial/)  
5. Apache Spark code is compact due compared to Hadoop MapReduce. Use of Scala makes it very short, reduces programming efforts. Also, Spark provides rich APIs in various languages such as Java, [Scala](https://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/), Python, and [R](https://data-flair.training/blogs/r-programming-tutorial/).  
6. Spark & Hadoop are both highly [fault-tolerant](https://data-flair.training/blogs/fault-tolerance-in-apache-spark/).  
7. Spark application running in Hadoop clusters is up to 10 times faster on disk than Hadoop MapReduce.

**What are the different methods to run Spark over Apache Hadoop?**

Instead of [MapReduce](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/) we can use [spark](https://data-flair.training/blogs/apache-spark-for-beginners/) on top of [Hadoop ecosystem](https://data-flair.training/blogs/hadoop-ecosystem-components/)  
-spark with [HDFS](https://data-flair.training/blogs/hadoop-hdfs-tutorial/)  
you can read and write data in HDFS  
-spark with [Hive](https://data-flair.training/blogs/apache-hive-tutorial/)  
you can read and analyse and write back to the hive

**What is SparkContext in Apache Spark?**

A SparkContext 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/)

**What is SparkSession in Apache Spark?**

Starting from [Apache Spark](https://data-flair.training/forums/topic/what-is-sparksession-in-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. For 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.

**SparkSession vs SparkContext in Apache Spark.**

Prior to [Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) 2.0.0 sparkContext was used as a channel to access all spark functionality.  
The spark driver program uses spark context to connect to the cluster through a resource manager ([YARN](http://data-flair.training/blogs/category/yarn/) orMesos..).  
sparkConf is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node.

In order to use APIs of [SQL](http://data-flair.training/blogs/spark-sql-tutorial/), [HIVE](http://data-flair.training/blogs/category/hive/), and [Streaming](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/), separate contexts need to be created.

Ex:

creating sparkConf :

val conf = new SparkConf().setAppName(“RetailDataAnalysis”).setMaster(“spark://master:7077”).set(“spark.executor.memory”, “2g”)

creation of sparkContext:

val sc = new SparkContext(conf)

**Spark Session:**

SPARK 2.0.0 onwards, SparkSession provides a single point of entry to interact with underlying Spark functionality and  
allows programming Spark with [DataFrame](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/) and Dataset APIs. All the functionality available with sparkContext are also available in sparkSession.

In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.

Once the SparkSession is instantiated, we can configure Spark’s run-time config properties.

**Example:**

Creating Spark session:  
val spark = SparkSession  
.builder  
.appName(“WorldBankIndex”)  
.getOrCreate()

Configuring properties:  
spark.conf.set(“spark.sql.shuffle.partitions”, 6)  
spark.conf.set(“spark.executor.memory”, “2g”)

Spark 2.0.0 onwards, it is better to use sparkSession as it provides access to all the spark Functionalities that sparkContext does. Also, it provides APIs to work on DataFrames and Datasets.

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

here are several abstractions of [Apache Spark](https://data-flair.training/blogs/apache-spark-for-beginners/):

**1. RDD:**  
An RDD refers to Resilient Distributed Datasets. RDDs are Read-only partition collection of records. It is Spark’s core abstraction and also a fundamental data structure of Spark. It offers to conduct [in-memory computations](https://data-flair.training/blogs/apache-spark-in-memory-computing/) on large clusters. Even in a [fault-tolerant](https://data-flair.training/blogs/fault-tolerance-in-apache-spark/) manner. For more detailed insights on RDD.follow link: [Spark RDD – Introduction, Features & Operations of RDD](https://data-flair.training/blogs/apache-spark-rdd-tutorial/)

**2. DataFrames:**  
It is a Dataset organized into named columns. DataFrames are equivalent to the table in a relational database or data frame in [R](https://data-flair.training/blogs/r-programming-tutorial/) /Python. In other words, we can say it is a relational table with good optimization technique. It is an immutable distributed collection of data. Allowing higher-level abstraction, it allows developers to impose a structure onto a distributed collection of data,. For more detailed insights on DataFrames. refer link:[Spark SQL DataFrame Tutorial – An Introduction to DataFrame](https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/)

**3. Spark Streaming:**  
It is a Spark’s core extension, which allows Real-time stream processing From several sources. For example Flume and Kafka. To offer a unified, continuous DataFrame abstraction that can be used for interactive and batch queries these two sources work together. It offers scalable, high-throughput and fault-tolerant processing. For more detailed insights on Spark Streaming. refer link: [Spark Streaming Tutorial for Beginners](https://data-flair.training/blogs/apache-spark-streaming-tutorial/)

**4. GraphX**  
It is one more example of specialized data abstraction. It enables developers to analyze social networks. Also, other graphs alongside Excel-like two-dimensional data. For more detailed insights on GaphX. refer link: [Apache Spark GraphX](https://data-flair.training/blogs/apache-spark-ecosystem-components/)

**How can we create RDD in Apache Spark?**

In 3 ways we can create RDD in [Apache Spark](https://data-flair.training/blogs/apache-spark-for-beginners/):  
1. Through distributing collection of objects  
2. By loading an external dataset  
3. From existing Apache Spark RDDs

**Using parallelized collection**

RDDs are generally created by parallelizing an existing collection  
i.e. by taking an existing collection in the program and passing  
it to [SparkContext’s](https://data-flair.training/blogs/learn-apache-spark-sparkcontext/) parallelize() method.

scala > val data = Array(1,2,3,4,5)  
scala > val dataRDD = sc.parallelize (data)  
scala > dataRDD.count

2. **External Datasets**

In Spark, a distributed dataset can be formed from any data source supported by Hadoop.

val dataRDD = spark.read.textFile(“F:/BigData/DataFlair/Spark/Posts.xml”).rdd

3. **Creating RDD from existing RDD**

Transformation is the way to create an RDD from already existing RDD.

Transformation acts as a function that intakes an RDD and produces another resultant RDD.  
The input RDD does not get changed,  
Some of the [operations](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/) applied on RDD are: filter, Map, FlatMap

val dataRDD = spark.read.textFile(“F:/Mritunjay/BigData/DataFlair/Spark/Posts.xml”).rdd

val resultRDD = data.filter{line => {line.trim().startsWith(“<row”)}

**Why is Spark RDD immutable?**

Following are the reasons:  
– Immutable data is always safe to share across multiple processes as well as multiple threads.  
– Since [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) is immutable we can recreate the RDD any time. (From lineage graph).  
– If the computation is time-consuming, in that we can cache the RDD which result in performance improvement.

**Explain the term paired RDD in Apache Spark**

Paired RDD is a distributed collection of data with the key-value pair. It is a subset of [Resilient Distributed Dataset](https://data-flair.training/blogs/apache-spark-rdd-tutorial/). So it has all the [feature of RDD](https://data-flair.training/blogs/apache-spark-rdd-features/) and some new feature for the key-value pair. There are many [transformation operations](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/) available for Paired RDD. These operations on Paired RDD are very useful to solve many use cases that require sorting, grouping, reducing some value/function.  
Commonly used operations on paired RDD are: groupByKey() reduceByKey() countByKey() join() etc  
Creation of Paired RDD:  
val pRDD:[(String),(Int)]=sc.textFile(“path\_of\_your\_file”)  
.flatMap(line => line.split(” “))  
.map{word=>(word,word.length)}  
Also using subString method(if we have a file with id and some value, we can create paired rdd with id as key and value as other details)

val pRDD2[(Int),(String)]=sc.textFile(“path\_of\_your\_file”)  
.keyBy(line=>line.subString(1,5).trim().toInt)  
.mapValues(line=>line.subString(10,30).trim())

**How is RDD in Spark different from Distributed Storage Management?**

Some of the differences between an [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) and Distributed Storage are as follows:

Resilient Distributed Dataset (RDD) is the primary abstraction of data for [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/)framework.  
Distributed Storage is simply a file system which works on multiple nodes.

RDDs store data [in-memory](http://data-flair.training/blogs/apache-spark-in-memory-computing/) (unless explicitly cached).  
Distributed Storage stores data in persistent storage.

RDDs can re-compute itself in the case of failure or data loss.  
If data is lost from the Distributed Storage system it is gone forever (unless there is an internal replication system).

**Explain transformation and action in RDD in Apache Spark.**

Transformations are [operations on RDD](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) that create one or more new [RDDs](http://data-flair.training/blogs/rdd-in-apache-spark/). E.g. map, filter, reduceByKey etc. In other words, transformations are functions that take an RDD as the input and produce one or more RDDs as the output. There is no change in the input RDD, but it always produces one or more new RDDs by applying the computations they represent.Transformations are lazy, i.e. are not executed immediately. Only after calling an action are transformations executed.

Actions are RDD operations that produce non-RDD values. In other words, an RDD operation that returns a value of any type but an RDD is an action. They trigger execution of RDD transformations to return values. Simply put, an action evaluates the [RDD lineage graph](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/). E.g. collect, reduce, count, foreach etc.

**Explain the RDD properties.**

* RDD (Resilient Distributed Dataset) is a basic abstraction in [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/).
* RDD is an immutable, partitioned collection of elements on the cluster which can be operated in parallel.
* **Each RDD is characterized by five main properties :**
* *Below operations are lineage operations.*1. List or Set of partitions.  
  2. List of dependencies on other (parent) RDD  
  3. A function to compute each partition

*Below operations are used for optimization during execution.*

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

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 :**

<li style=”list-style-type: none”>

* 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 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*

**What is lineage graph in Apache Spark?**

When we apply a different [transformation on RDD](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) it [creates RDD](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) Linage graph. It is a new [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) from already existing RDDs. It is the dependencies graph between the existing and the new RDD formed. the need of RDD lineage graph arrives when we want to compute new RDD or if we want to recover the lost data from the lost persisted RDD.

**Explain the terms  Spark Partitions and Partitioners.**

**PARTITIONS :**

* Partitions also known as ‘Split’ in HDFS, is a logical chunk of data set which may be in the range of Petabyte, Terabytes and distributed across the cluster.
* By Default, Spark creates one Partition for each block of the file (For HDFS)
* Default block size for HDFS block is 64 MB (Hadoop Version 1) / 128 MB (Hadoop Version 2) so as the split size.
* However, one can explicitly specify the number of partitions to be created.
* Partitions are basically used to speed up the data processing.

**PARTITIONER :**

* 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 (number of partitions – 1)
* Partitioner captures the data distribution at the output. A scheduler can optimize future operation based on the type of partitioner. (i.e. if we perform any operation say transformation or action which require shuffling across nodes in that we may need the partitioner. Please refer reduceByKey() transformation in the forum)
* Basically there are three types of partitioners in Spark:

(1) Hash-Partitioner (2) Range-Partitioner (3) One can make its *Custom Partitioner*

**Property Name :**spark.default.parallelism  
**Default Value:**For distributed shuffle operations like reduceByKey and join, the largest number of partitions in a parent RDD. For operations like parallelize with no parent RDDs, it depends on the cluster manager:  
•Local mode: number of cores on the local machine  
•Mesos fine-grained mode: 8  
•Others: total number of cores on all executor nodes or 2, whichever is larger  
**Meaning :**Default number of partitions in RDDs returned by transformations like join.

**By Default, how many partitions are created in RDD in Apache Spark?**

* By Default, Spark creates one Partition for each block of the file (For HDFS)
* Default block size for HDFS block is 64 MB (Hadoop Version 1) / 128 MB (Hadoop Version 2).
* However, one can explicitly specify the number of partitions to be created.
* **numbers of cores in Cluster = no. of partitions**

**What is Spark DataFrames?**

**DataFrame** consists of two words data and frame, means data has to be fit in some kind of frame. We can understand a frame as a schema of the relational database.

In Spark, DataFrame is a collection of distributed data over the network with some schema. We can understand it as the data formatted as row/column manner. DataFrame can be created from Hive data, JSON file, CSV, Structured data or raw data that can be framed in structured data. We can also create a DataFrame from [RDD](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) if some schema can be applied on that RDD.  
Temporary view or table can also be created from DataFrame as it has data and schema. We can also run [SQL](https://data-flair.training/blogs/spark-sql-tutorial/) query on created table/view to get the faster result.  
It is also evaluated lazily ([Lazy Evaluation](https://data-flair.training/blogs/apache-spark-lazy-evaluation/)) for better resource utilization.

**What are benefits of DataFrame in Spark?**

Following are the Benefits of DataFrames.

1.DataFrame is distributed collection of data. In DataFrames, data is organized in named column.

2. They are conceptually similar to a table in a relational database. Also, have richer optimizations.

3. DataFrames empower SQL queries and the DataFrame API.

4. we can process both structured and unstructured data formats through it. Such as: Avro, CSV, elastic search, and Cassandra. Also, it deals with storage systems [HDFS](https://data-flair.training/blogs/hadoop-hdfs-tutorial/), HIVE tables, MySQL, etc.

5. In DataFrames, Catalyst supports optimization([catalyst Optimizer](https://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/)). There are general libraries available to represent trees. In four phases, DataFrame uses Catalyst tree transformation:

– Analyze logical plan to solve references  
– Logical plan optimization  
– Physical planning  
– Code generation to compile part of a query to Java bytecode.

6. The DataFrame API’s are available in various programming languages. For example Java, [Scala](https://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/), Python, and [R](https://data-flair.training/blogs/r-programming-tutorial/).

7. It provides Hive compatibility. We can run unmodified Hive queries on existing Hive warehouse.

8. It can scale from kilobytes of data on the single laptop to petabytes of data on a large cluster.

9. DataFrame provides easy integration with Big data tools and framework via Spark core.

**What is Spark Dataset?**

A **Dataset** is an immutable collection of objects, those are mapped to a relational schema. They are strongly-typed in nature.  
There is an encoder, at the core of the Dataset API. That Encoder is responsible for converting between JVM objects and  
tabular representation. By using Spark’s internal binary format, the tabular representation is stored that allows to carry out operations on serialized data and improves memory utilization. It also supports automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long) and Scala case classes. It offers many functional [transformations](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/)(e.g. map, flatMap, filter).

**What are the advantages of datasets in spark?**

**1)Static typing-**  
With Static typing feature of Dataset, a developer can catch errors at compile time (which saves time and costs).  
**2)Run-time Safety**:-  
[Dataset](https://data-flair.training/blogs/apache-spark-dataset-tutorial) APIs are all expressed as lambda functions and JVM typed objects, any mismatch of typed-parameters will be  
detected at compile time. Also, analysis error can be detected at compile time too, when using Datasets,  
hence saving developer-time and costs.  
**3)Performance and Optimization**  
Dataset APIs are built on top of the Spark SQL engine, it uses Catalyst to generate an optimized logical and physical query plan providing the space and speed efficiency.  
4) For processing demands like high-level expressions, filters, maps, aggregation, averages, sum,  
[SQL](https://data-flair.training/blogs/spark-sql-tutorial/) queries, columnar access and also for use of lambda functions on semi-structured data, DataSets are best.  
5) Datasets provides rich semantics, high-level abstractions, and domain-specific APIs

**What is Directed Acyclic Graph in Apache Spark?**

In mathematical term, the **Directed Acyclic Graph** is a graph with cycles which are not directed. DAG is a graph which contains set of all the operations that are applied on [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/). On RDD when any [action](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) is called. [Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) creates the DAG and submits it to the DAG scheduler. Only after the DAG is built, Spark creates the query optimization plan. The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together.  
Fault tolerance is achieved in Spark using the Directed Acyclic Graph. The query optimization is possible in Spark by the use of DAG. Thus, we get the better performance by using DAG.

**What is the need for Spark DAG?**

**Need of DAG in Spark**  
As we know there were some [limitations with Hadoop MapReduce](https://data-flair.training/blogs/13-limitations-of-hadoop/). To Overcome those limitations, Apache Software introduces DAG in [Spark](https://data-flair.training/blogs/apache-spark-for-beginners/). Let’s first study the computation process of MapReduce. It was generally carried in three steps:

1. HDFS is used to read data.  
2. After that, we apply Map and Reduce operations.  
3. Again the result of the computation is written back to HDFS.

In Hadoop, each [MapReduce](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/) operation is independent of each other. Even [HADOOP](https://data-flair.training/blogs/hadoop-tutorial-for-beginners/) has no idea of which Map reduce may come next. Therefore, sometimes it is irrelevant to read and write back the immediate result between two map-reduce jobs for some iterations. As a result, disk memory or the memory in stable storage ([HDFS](https://data-flair.training/blogs/hadoop-hdfs-tutorial/)) gets wasted.

While we talk about multiple-step, all the jobs are blocked from the beginning till the completion of the previous job. Hence, complex computation can require a long time with small data volume.

But, After DAG introduced in Spark, the execution plan is optimized, e.g. to minimize shuffling data around. Since, a DAG (Directed Acyclic Graph) of consecutive computation stages is formed.

**What is the difference between DAG and Lineage?**

**Lineage graph**  
As we know, that whenever a series of transformations are performed on an [RDD](https://data-flair.training/blogs/apache-spark-rdd-tutorial/), they are not evaluated immediately, but lazily([Lazy Evaluation](https://data-flair.training/blogs/apache-spark-lazy-evaluation/)). When a new RDD has been created from an existing RDD, that new RDD contains a pointer to the parent RDD. Similarly, all the dependencies between the RDDs will be logged in a graph, rather than the actual data. This graph is called the lineage graph.

Now coming to DAG,

**Directed Acyclic Graph(DAG)**  
DAG in [Apache Spark](https://data-flair.training/blogs/apache-spark-for-beginners/) is a combination of Vertices as well as Edges. In DAG vertices represent the RDDs and the edges represent the Operation to be applied on RDD. Every edge in DAG is directed from earlier to later in a sequence.When we call an[Action](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/), the created DAG is submitted to DAG Scheduler which further splits the graph into the stages of the task.

**What is the difference between Caching and Persistence in Apache Spark?**

Cache and Persist both are optimization techniques for [Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) computations.

Cache is a synonym of Persist with MEMORY\_ONLY storage level(i.e) using Cache technique we can save intermediate results in memory only when needed.

Persist marks an [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) for persistence using storage level which can be MEMORY, MEMORY\_AND\_DISK, MEMORY\_ONLY\_SER, MEMORY\_AND\_DISK\_SER, DISK\_ONLY, MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2

Just because you can cache an RDD in memory doesn’t mean you should blindly do so. Depending on how many times the dataset gets accessed and the amount of work involved in doing so, recomputation can be faster by the increased memory pressure.

It should go without saying that if you only read a dataset once there is no point in caching it, it will actually make your job slower.

**What are the different ways of representing data in Spark?**

**1. RDD**  
RDD refers to “Resilient Distributed Dataset”. RDD is core abstraction and fundamental data structure of Apache Spark. It is an immutable collection of objects which computes on the different node of the cluster. As we know RDDs are immutable, though we can not make any changes in it we can apply following operations like [Transformation and Actions](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/) on them.It perform [in-memory computations](https://data-flair.training/blogs/apache-spark-in-memory-computing/) on large clusters in a [fault-tolerant](https://data-flair.training/blogs/fault-tolerance-in-apache-spark/) manner. Basically, There are three ways to create RDDs in Spark such as – Data in stable storage, other RDDs, and parallelizing already existing collection in driver program.Follow this link to learn [Spark RDD](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) in great detail.

**2. DataFrame**  
In DataFrame, data organized into named columns. This table is as similar as a table in a relational database. DataFrames is also an immutable distributed collection of data. It allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. Follow this link to learn [Spark DataFrame](https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/) in detail.

**3. Spark Dataset APIs**  
It is an extension of DataFrame API. It provides type-safe, object-oriented programming interface. It takes advantage of Spark’s Catalyst optimizer, by exposing d

**What is write ahead log(journaling) in Spark?**

There are two types of failures in any [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) job – Either the **driver failure** or the **worker failure**.

When any worker node fails, the executor processes running in that worker node will be killed, and the tasks which were scheduled on that worker node will be automatically moved to any of the other running worker nodes, and the tasks will be accomplished.

When the driver or master node fails, all of the associated worker nodes running the executors will be killed, along with the data in each of the executors’ memory. In the case of files being read from reliable and fault tolerant file systems like HDFS, zero data loss is always guaranteed, as the data is ready to be read anytime from the file system. Checkpointing also ensures [fault tolerance in Spark](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/) by periodically saving the application data in specific intervals.

In the case of [Spark Streaming](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/)application, zero data loss is not always guaranteed, as the data will be buffered in the executors’ memory until they get processed. If the driver fails, all of the executors will be killed, with the data in their memory, and the data cannot be recovered.

To overcome this data loss scenario, **Write Ahead Logging (WAL) has been introduced in Apache Spark 1.2.** With WAL enabled, the intention of the operation is first noted down in a log file, such that if the driver fails and is restarted, the noted operations in that log file can be applied to the data. For sources that read streaming data, like Kafka or Flume, receivers will be receiving the data, and those will be stored in the executor’s memory. With WAL enabled, these received data will also be stored in the log files.

WAL can be enabled by performing the below:

1. Setting the checkpoint directory, by using streamingContext.checkpoint(path)

2. Enabling the WAL logging, by setting spark.stream.receiver.WriteAheadLog.enable to True.

**Explain catalyst query optimizer in Apache Spark.**

The most important component of [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is Spark SQL that deals with [DataFrame](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/) API and SQL queries. Inside [Spark SQL](http://data-flair.training/blogs/spark-sql-tutorial/) there lies an optimizer called Catalyst Query Optimizer. using this Spark creates an extensible query optimizer. This query optimizer Spark is based on Scala’s functional programming construct.

**Need of query optimizer:**

* To get solution to tackle various problem with Bigdata.
* As a solution to extend the optimizer.

**We use catalyst general tree transformation frame work in four phases**

* Analysis
* Logical Optimization.
* Physical Planning.
* Code generation.

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

Metadata accumulates on the driver as consequence of shuffle operations. It becomes particularly tedious during long-running jobs.  
To deal with the issue of accumulating metadata, there are two options:

* First, set the **spark.cleaner.ttl**parameter to trigger automatic cleanups. However, this will vanish any [persisted RDDs.](http://data-flair.training/blogs/apache-spark-rdd-persistence-caching/)
* The other solution is to **simply split long-running jobs into batches and write intermediate results to disk**. This facilitates a fresh environment for every batch and don’t have to worry about metadata build-up.

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

* Standalon Cluster Manager
* Apache Mesos
* Hadoop YARN

**Standalone Cluster:** The cluster consists of master and number of worker node. In this mode, the allocation of resources is based on a number of cores. An application grabs all the cores in the cluster.

**Apache Mesos:** By dynamic resource sharing and isolation Apache Mesos shares the workload in distributed environment. It joins the existing resource of the machine/node in the cluster. It acts as a resource management platform for Hadoop and Bigdata cluster. In this various physical resources are joined in single virtual resources. As a result, it is opposite of virtualization.

[Hadoop YARN](http://data-flair.training/blogs/hadoop-yarn-tutorial/)**:** YARN stands for Yet another Resource Negotiator. It is a combination of resource manager and node manager which can run on both Linux and Windows. It is also known bt Mapreduce 2.0. It lets different data processing engine like graph processing, stream processing to run and process data stored in HDFS.

**What is Speculative Execution in Apache Spark?**

The **Speculative task** in [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is task that runs slower than the rest of the task in the job.It is health check process that verifies the task is speculated, meaning the task that runs slower than the median of successfully completed task in the task sheet. Such tasks are submitted to another worker. It runs the new copy in parallel rather than shutting down the slow task.

In the cluster deployment mode, the thread starts as *TaskSchedulerImp1with spark.speculation enabled. It executes periodically every spark.speculation.interval after the initial spark.speculation.interval* passes.

**What is the role of Spark Driver in spark applications?**

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 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](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/), connected to a given Spark Master.It conjointly delivers the RDD graphs to Master, wherever the standalone cluster manager runs.

**Explain Accumulator in Spark.**

* 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 propagated back to the driver program.
* *Accumulator* lowers this restriction.  
  **Use Cases :**
* One of the most common use of accumulator is count 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 network during session, during Olympic data analysis we have to find age where we said (age != ‘NA’) in SQL query in short finding bad / corrupted records.

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

) Management of [DAG’s](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)– People often do mistakes in DAG controlling. Always try to use reducebykey instead of groupbykey. The ReduceByKey and GroupByKey can perform almost similar functions, but GroupByKey contains large data. Hence, try to use ReduceByKey to the most. Always try to lower the side of maps as much as possible. Try not to waste more time in Partitioning.Try not to shuffle more. Try to keep away from Skews as well as partitions too.

2) Maintain the required size of the shuffle blocks.

**Why we need compression and what are the different compression format supported?**

* In Big Data, when we used the compression, it saves the storage space and reduce the network overhead.
* One can specify the compression coded while writing the data to [HDFS](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) ( Hadoop format)
* One can also read the compressed data, for that also we can use compression codec.
* Following are the different compression format support in BigData:

\* gzip  
\* lzo  
\* bzip2  
\* Zlib  
\* Snappy

**groupByKey vs reduceByKey in Apache Spark**

On applying **groupByKey()** on a dataset of (K, V) pairs, the data shuffle according to the key value K in another [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/). In this transformation, lots of unnecessary data transfer over the network.

Spark provides the provision to save data to disk when there is more data shuffling onto a single executor machine than can fit in memory.

On applying **reduceByKey** on a dataset (K, V), before shuffeling of data the pairs on the same machine with the same key are combined.

**Explain mapPartitions() and mapPartitionsWithIndex()**

mapPartitions() and mapPartitionsWithIndex() are both transformation.

**mapPartitions() :**  
> mapPartitions() can be used as an alternative to map() and foreach() .  
> mapPartitions() can be called for each partitions while map() and foreach() is called for each elements in an RDD  
> Hence one can do the initialization on per-partition basis rather than each element basis

**mapPartitions() :**  
> mapPartitionsWithIndex is similar to mapPartitions() but it provides second parameter index which keeps the track of partition.

MapPartitions:

It runs one at a time on each partition or block of the Rdd, so function must be of type iterator<T>. It improves performance by reducing creation of object in map function.  
2.5. MappartionwithIndex:

It is similar to MapPartition but with one difference that it takes two parameters, the first parameter is the index and second is an iterator through all items within this partition (Int, Iterator<t>).

**mapPartitions() syntax**  
def mapPartitions[U](f: (Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]  
Permalink  
Return a new RDD by applying a function to each partition of this RDD.

**mapPartitionsWithIndex syntax**

def mapPartitionsWithIndex[U](f: (Int, Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

**What is Map in Apache Spark?**

Map is a [transformation](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) applied to each element in a [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) and it provides a new RDD as a result. In Map transformation, user-defined business logic will be applied to all the elements in the RDD.  
It is similar to FlatMap, but unlike FlatMap Which can produce 0, 1 or many outputs, Map can only produce one to one output.  
Map operation will transforms an RDD of length N into another RDD of length N.

A——->a  
B——->b  
C——->c  
Map Operation

Map transformation will not shuffle data from one partition to many. It will keep the operation narrow.

**What is FlatMap in Apache Spark?**

FlatMap is a [transformation operation in Apache Spark](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/)to [create an RDD](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) from existing [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/). 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.

**Explain API createOrReplaceTempView()**

* Its basic Dataset function.
* Its 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.

**Explain textFile Vs wholeTextFile in Spark**

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

* 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 filename with path as key,  
  and value being the whole file as string

val myfilerdd = sc.wholeTextFiles("/home/hdadmin/MyFiles")

val keyrdd = myfilerdd.keys

keyrdd.collect

val filerdd = myfilerdd.values

filerdd.collect

**Explain cogroup() operation in Spark**

It’s a transformation.

def cogroup[W1, W2, W3](other1: RDD[(K, W1)], other2: RDD[(K, W2)], other3: RDD[(K, W3)]): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2], Iterable[W3]))]

For each key k in this or other1 or other2 or other3, return a resulting RDD that contains a tuple with the list of values for that key in this, other1, other2 and other3.

**Explain pipe() operation in Apache Spark**

**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 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.

**Explain Spark coalesce() operation**

**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 be executed 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.

**Explain the repartition() operation in Spark**

**def repartition(numPartitions: Int)(implicit ord: Ordering[(K, C)] = null): RDD[(K, C)]**  
Return a new [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/) that has exactly numPartitions partitions.  
Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data.  
If you are decreasing the number of partitions in this RDD, consider using coalesce, which can avoid performing a shuffle.

**What is PageRank in Spark?**

**What are the roles and responsibilities of worker nodes in the Apache Spark cluster? Is Worker Node in Spark is same as Slave Node?**

[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 be performed will be automatically allocated to any other worker nodes

**How to split single HDFS block into partitions RDD?**

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/).  
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 be created.

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/):

data = context.textFile("/user/dataflair/file-name", 20)  
It will create 20 partitions for the file. ie for each block 2 partitions will be created.