SPARK

Here are some Frequently asked Spark Interview Questions and Answers for freshers and experienced.

**Q.1) What is Apache Spark?**

**Apache Spark** is open source, wide range data processing engine. It is data processing engine with high APIs. It allows data worker to execute streaming, machine learning or SQL workloads. These jobs need fast iterative access to datasets. Spark provides API in various languages like **Python, R,**[**Scala**](http://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/)**, Java**. We can run Spark by itself or on various existing [cluster manager](http://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/). There is various deployment option in Spark. For example, **Standalone Deploy Mode,**[**Apache Mesos**](http://data-flair.training/blogs/apache-mesos-tutorial-learn-mesos/)**,**[**Hadoop YARN**.](http://data-flair.training/blogs/hadoop-yarn-tutorial/)

The design of the Apache is so dynamic that it can integrate with all the[**Big Data**](http://data-flair.training/blogs/why-learn-big-data-use-cases/) tools. For example, Spark can access data from any of the [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/) data source. It can also run in Hadoop data cluster. Spark does not have its own storage system. It relies on [**HDFS**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) or other file storage for storing the data.

[Read about Apache Spark in detail.](http://data-flair.training/blogs/apache-spark-tutorial/)

**Q.2) 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.

**Q.3) 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.

**Q.4) 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.

**Q.5) List the languages supported by Apache Spark.**

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

**Q.6) 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.

**Q.7) 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.

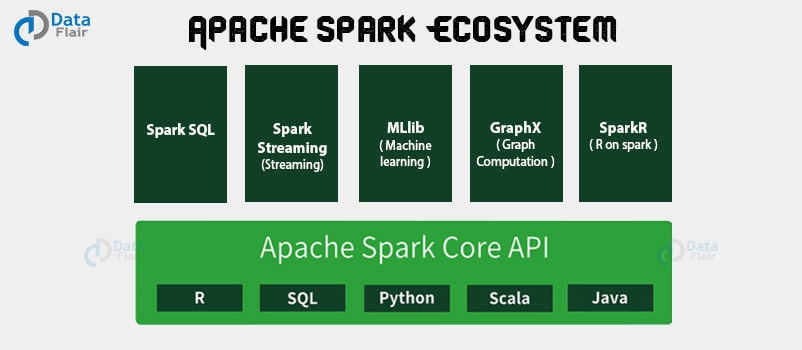
**Memory Management –**Apache Spark has automatic memory management system. While Memory Management in Apache Hadoop can be either statistic or dynamic.

**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.

**Q.8) What are the components of Spark Ecosystem?**

The various components of Apache Spark are:

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/apache-spark-ecosystem.jpg)

**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.

**Q.9) What is Spark Core?**

**Spark Core** is a common execution engine for Spark platform. It provides parallel anddistributed 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.

**Q.10) How is data represented in Spark?**

The data can be represented in three ways in Apache Spark: RDD, DataFrame, DataSet.

**RDD:** RDD stands for Resilient Distributed Datasets. It is also aRead-only partition collection of records. RDD is the fundamental data structure of Spark. Hence, RDDs can only be created through deterministic operation on either:

Data in stable storage.

Parallelizing already existing collection in driver program.

Other RDDs. RDD allows programmer perform in-memory computations on large clusters in a fault-tolerant manner. Thus, speed up the task.

**DataFrame:** Unlike an RDD, the data organizes into named columns, like a table in a relational database. It is also an immutable distributed collection of data. [**DataFrame**](http://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/) allows developers to impose a structure onto a distributed collection of data, therefore, allowing higher-level abstraction.

**DataSet:** Dataset is an extension of DataFrame API which provides type-safe, object-oriented programming interface. [**DataSet**](http://data-flair.training/blogs/apache-spark-dataset-tutorial/) also takes advantage of [**Spark’s Catalyst optimizer**](http://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/) by exposing expressions and data fields to a query planner.

**Q.11) 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. We can use this in parallel operations.

**Q.12) 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.

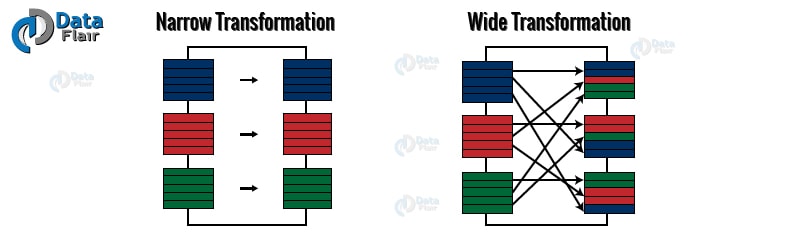
**Action** returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD. After application of all the intermediate transformation, it gives the final result to driver program or writes it out to file system. Upon applying Actions on an RDD non-RDD values gets generate.

**Q.13) 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.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/spark-narrow-and-wide-transformations.jpg)

**Q.14) 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.

**Q.15) 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 andValue are the data corresponding to the key value.

**Q.16) 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. Incache() 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

**Q.17) 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.

**Q.18) 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.

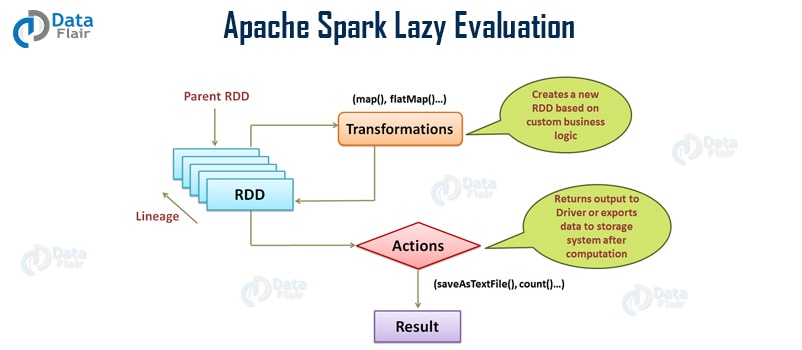
**Q.19) What is lineage graph?**

***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.

**Q.20) What is lazy evaluation in Spark?**

The **lazy evaluation** known ascall-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.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/apache-spark-lazy-evaluation-1.jpg)

[Read about Spark Lazy Evaluation in detail.](http://data-flair.training/blogs/apache-spark-lazy-evaluation/)

**Q.21) 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.

**Q.22) 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. Thus makes it reusable for future use. This process speeds up the further computation ten times.

[Read about RDD Persistence and Caching Mechanism in detail.](http://data-flair.training/blogs/apache-spark-rdd-persistence-caching/)

**Q.23) 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.

**Q.24) Explain the run-time architecture of Spark?**

The components of the [**run-time architecture of Spark**](http://data-flair.training/blogs/how-apache-spark-works-run-time-spark-architecture/) are as follows:  
a. The driver  
b. Cluster manager  
c. Executors

**The Driver –**The main() method of the program runs in the driver. The process that runs the user code which creates RDDs performs transformation and action, and also creates SparkContext is called diver. When the Spark Shell is launched, this signifies that we have created a driver program. The application finishes, as the driver terminates. Finally, driver program splits the Spark application into the task and schedules them to run on the executor.

**Cluster Manager –** Spark depends on cluster manager to launch executors. In some cases, even the drivers are launched by cluster manager. It is a pluggable component in Spark. On the cluster manager, the Spark scheduler schedules the jobs and action within a spark application in FIFO fashion. Alternatively, the scheduling can also be done in Round Robin fashion. The resources used by a Spark application can also be dynamically adjusted based on the workload. Thus, the application can free unused resources and request them again when there is a demand. This is available on all coarse-grained [cluster managers,](http://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/) i.e. **standalone mode**, **YARN mode**, and **Mesos** coarse-grained mode.

**The Executors –** Each task in the Spark job runs in the Spark executors. thus, Executors are launched once in the beginning of Spark Application and then they run for the entire lifetime of an application. Even after the failure of Spark executor, the Spark application can continue with ease.

There are two main roles of the executors:

Runs the task that makes up the application and returns the result to the driver.

Provide in-memory storage for RDDs that the user program cache.

**Q.25) 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.

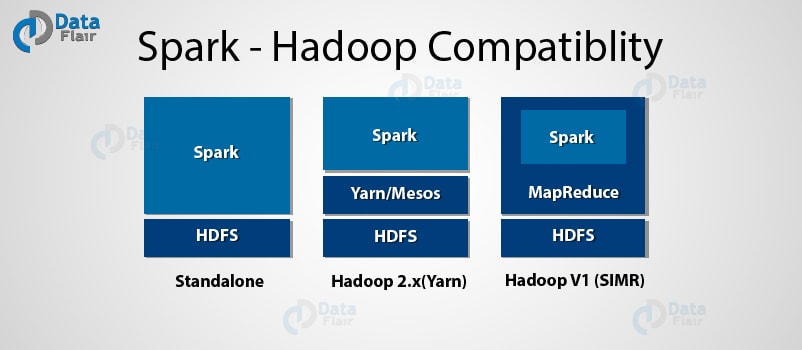
**Hadoop YARN –** YARN became the sub-project of Hadoop in the year 2012. The key idea behind YARN is to bifurcate the functionality of resource manager and job scheduling into different daemons. The plan is to have a Global [**Resource Manager (RM)**](http://data-flair.training/blogs/hadoop-yarn-resource-manager-guide-tutorial/)and 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.

[Read about Spark Cluster Managers in detail.](http://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/)

**Q.26) In how many ways can we use Spark over Hadoop?**

In three ways we can run Spark over Hadoop: Standalone, YARN, SIMR

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/apache-spark-hadoop-compatiblity.jpg)

**Standalone –** In this, we can either divide resource on all machines or subset of machines in [Hadoop cluster.](http://data-flair.training/blogs/install-hadoop-2-x-ubuntu-hadoop-multi-node-cluster/)

**YARN –** We can run Spark on YARN without any pre-requisites. Thus, we can integrate Spark in Hadoop stack and take an advantage of facilities of Spark.

**SIMR (Spark in MapReduce) –** Another way to do this is by launching Spark job inside Map reduce. With SIMR we can use Spark shell in few minutes after downloading it. This reduces the overhead of deployment, and we can play with Spark.

**Q.27) What is YARN?**

**YARN** became the sub-project of Hadoop in the year 2012. It is also known as MapReduce 2.0. 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 GlobalResource Manager(RM) 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.

The **Resource Manager** manages resource among all the applications in the system. The Resource Manager has scheduled and Application Manager. The Scheduler allocates resource to the various running application. The Scheduler is pure Scheduler if it performs no monitoring or tracking of the status of the application. The **Application Manager** manages applications across all the nodes. NodeManager contains ApplicationMaster and container. A**container** is a place where a unit of work happens. Each task of MapReduce runs in one container. The per-application ApplicationMaster is a framework specific library. It negotiates resources from the ResourceManager and continues with **NodeManager(s)** to execute and watch the tasks. Application or job requires one or more containers. NodeManager looks after containers, resource usage (CPU, memory, disk, and network) and reporting this to the ResourceManager.

[Read about YARN in detail.](http://data-flair.training/blogs/hadoop-yarn-tutorial/)

**Q.28) How can we launch Spark application on YARN?**

There are two deployment modes to launch Spark application on YARN: the cluster mode and the client mode.

**In cluster mode,** the Spark driver runs inside Application Master Process and this is managed by YARN on the cluster.

**In client mode,** the driver runs in the client process. The Application Master requests a resource from YARN. And it provides it to the driver program.

**Q.29) 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.

[Read Spark Catalyst Optimizer in detail.](http://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/)

**Q.30) 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.

**Q.31) What is Accumulator?**

The accumulator is the type of Shared variable that is only added through associative and commutative operations. Using accumulator we can update the value of the variable while executing. 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.

**Q.32) 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.

**Q.33) 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.

**Q.34) 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.

**Q.35) 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.

**Q.36) 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.

**Q.37) List the popular use cases of Apache Spark.**

The most popular use-cases of Apache Spark are:  
1. Streaming  
2. Machine Learning  
3. interactive Analysis  
4. fog computing  
5. Using Spark in the real world

**Q.38) 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.

We have categorized the above Spark Interview Questions for Freshers and Experienced-

Spark Interview Questions and Answers for Fresher – Q.No.1-8, 37

Spark Interview Questions and Answers for Experienced – Q.No. 9-36, 38

Follow this link to [read more Spark Basic interview Questions with Answers.](http://data-flair.training/blogs/apache-spark-interview-questions-answers/)

Spark SQL Interview Questions and Answers

In this section, we will discuss some basic Spark SQL Interview Questions and Answers.

**Q.39) 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.

[Read about Spark DataFrame in detail.](http://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/)

**Q.40) What are the advantages of DataFrame?**

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.

Data frame is both space and performance efficient.

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.

The DataFrame API’s are available in various programming languages. For example Java, Scala, Python, and R.

It provides Hive compatibility. As a result, we can run unmodified Hive queries on existing Hive warehouse.

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.

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

**Q.41) 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.

**Q.42) 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.

**Q.43) 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.

**Q.44) 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.

We have categorized the above frequently asked Spark SQL Interview Questions for Freshers and Experienced-

Spark Interview Questions and Answers for Fresher – Q.No. 39-42

Spark Interview Questions and Answers for Experienced – Q.No. 43, 44

Follow this link to [read more basic Spark interview Questions with Answers.](http://data-flair.training/blogs/50-apache-spark-interview-questions/)

Spark Streaming Interview Questions and Answers

In this section, we will discuss some basic Spark Streaming Interview Questions and Answers.

**Q.45) What is Spark Streaming?**

Through [**Spark streaming**,](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/) we achieve fault tolerant processing of live data stream. The input data can be from any source. For example, likeKafka, Flume, kinesis, twitter or HDFS/S3. It gives the data to filesystems, databases, and live dashboards after processing. The working of Spark Streaming is as under:

Spark Streaming takes in the live data.

The input data stream is divided into batches of input data.

The Spark engine processes the batches of input data. The final result is also in batches.

**Q.46) What is DStream?**

***DStream*** is the high-level abstraction provided by Spark Streaming. It represents a continuous stream of data. Thus, DStream is internally a sequence of RDDs. There are two ways to create DStream:

by using data from different sources such as Kafka, Flume, and Kinesis.

by applying high-level operations on other DStreams.

**Q.47) Explain different transformation on DStream.**

DStream is a basic abstraction of Spark Streaming. It is a continuous sequence of RDD which represents a continuous stream of data. Like RDD, DStream also supports many transformations which are available on normal Spark RDD. For example, map(func), flatMap(func), filter(func) etc.

**Q.48) Does Apache Spark provide checkpointing?**

Yes, Apache Spark provides checkpointing. Apache supports two types of checkpointing:

**Reliable Checkpointing:** It refers to that checkpointing in which the actual RDD is saved in the reliable distributed file system, e.g. HDFS. To set the checkpoint directory call: SparkContext.setCheckpointDir(directory: String). When running on the cluster the directory must be an HDFS path since the driver tries to recover the checkpointed RDD from a local file. While the checkpoint files are actually on the executor’s machines.

**Local Checkpointing:** In this checkpointing, in Spark Streaming or GraphX we truncate the RDD lineage graph in Spark. In this, the RDD is persisted to local storage in the executor.

**Q.49) What is write ahead log(journaling)?**

The **write-ahead log** is a technique that provides durability in a database system. It works in the way that all the operation that applies on data, we write it to write-ahead log. The logs are durable in nature. Thus, when the failure occurs we can easily recover the data from these logs. When we enable the write-ahead log Spark stores the data in fault-tolerant file system.

**Q.50) What is a reliable and unreliable receiver in Spark?**

**Reliable Receiver –** A reliable receiver acknowledges to the source on receiving data and stores it. Implementing this receiver involves examining of the semantics of source acknowledgments.

**Unreliable Receiver –** An unreliable receiver does not send an acknowledgment to a source. It is for sources that do not bear an acknowledgment. It is also for reliable sources when one does not want to enter the complexity of acknowledgment.

We have categorized the above Spark Interview Questions for Freshers and Experienced-

Spark Interview Questions and Answers for Fresher – Q.No. 45-47

Spark Interview Questions and Answers for Experienced – Q.No. 48-50

Spark MLlib Interview Questions and Answers

Here are some Spark MLlib Interview Questions and Answers for freshers and experienced.

**Q.51) What is Spark MLlib?**

***MLlib*** is the name of Spark’s machine learning library. The various tools provided by MLlib are:

**ML Algorithms:** It contains common learning algorithms such as classification, regression, etc.

**Featurization:** it has tools for feature extraction, dimensionality reduction, and selection. It also has tools for constructing, evaluating and tuning ML Pipelines.

**Q.52) What is Sparse Vector?**

A local vector contains both the integer type and 0-based indices. It also has double-typed values, which is stored on the single machine. In***MLlib*** two types of local vectors are supportive namely Dense and Sparse vector. The sparse vector is one in which most of the entries are zero.

**Q.53) How to create a Sparse vector from a dense vector?**

Vector sparseVector = Vectors.sparse(4, new int[] {1, 3}, new double[] {3.0, 4.0});

65 QUESTIONS DATA FLAIR

**1) What is Apache Spark?**

Apache Spark is a powerful open source flexible data processing framework built around speed, ease of use, and sophisticated analytics.Apache Spark is lightening fast in cluster computing system. Spark can run on [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), standalone or in the cloud and is capable of accessing data from various sources including [**HDFS**](http://data-flair.training/blogs/category/hdfs/), [**HBase**](http://data-flair.training/blogs/category/hbase/), Cassandra or others.

Because of in-cluster computing in Spark, it doesn’t require to keep shuffling things in and out of disk. This results in faster processing of data in spark.

Spark has several advantages compared to other big data and MapReduce technologies like Hadoop and Storm. Few of them are:  
**1.Speed**  
It can run program up to 100 times faster than Hadoop-MapReduce in memory, or 10 times faster on disk.  
**2.Ease of Use**  
Spark has easy-to-use APIs for operating on large data sets. This includes a collection of over 100 operators for  
transforming data and familiar data frame APIs for manipulating semi-structured data.  
We can write applications in Java, [**Scala**](http://data-flair.training/blogs/category/scala/), Python, [**R**](http://data-flair.training/blogs/category/r/).  
**3.A Unified Engine**  
Spark comes with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing.  
**4.Runs Everywhere**  
Spark can run on top of Hadoop, Mesos, standalone, or in the cloud.

[**Spark ecosystem**](https://data-flair.training/blogs/apache-spark-ecosystem-components/)

Below is the brief overview of Spark Ecosystem and its components.  
It consists of:  
[**Spark Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/)**:** Spark Streaming is used for processing the real-time streaming data.  
[**Spark SQL**](http://data-flair.training/blogs/spark-sql-tutorial/)**:** Spark SQL component is a library on top of Spark cluster, by using we can run SQL queries on Spark data.  
**Spark MLlib:** MLlib is Spark’s scalable machine learning library.  
**Spark GraphX:** GraphX is for graphs and graph-parallel computation.

**2) What are the features and characteristics of Apache Spark?**

[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is the Next-Gen Big Data tool (considered as future of Big Data and successor of [**Hadoop MapReduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/)), below are the features of Spark:

**Speed:**Speed always matters for processing data, organizations want to process voluminous data as fast as possible. Spark is Lightning fast processing tool makes it speedier to handle complex processing. As it follows the concept of [**RDD (Resilient Distributed Dataset)**](http://data-flair.training/blogs/rdd-in-apache-spark/) which allows it to store data transparently in memory, which helps in reducing read & write to disc one of the main time-consuming factor.

**Usability:**Ability to support multiple languages makes it dynamic. It allows you quickly write application in Java, [**Scala**](http://data-flair.training/blogs/category/scala/), Python and [**R**](http://data-flair.training/blogs/category/r/).

[**In-Memory Computing**](http://data-flair.training/blogs/apache-spark-in-memory-computing/): Keeping data in servers' RAM as it makes accessing stored data quickly. In memory, analytics accelerates iterative machine learning algorithms as it saves data read and write round trip from/to disk.

**Pillar to Sophisticated Analytics:**Spark comes with tools for interactive/declarative queries, streaming data, machine learning which is addition to simple map and reduce, so that users can combine all this into single workflow.

**Real Time Stream Processing:**Spark streaming can handle real-time stream processing along with integration of other frameworks which concludes that spark's streaming ability is easy, fault tolerance and Integrated.

**Compatibility with Hadoop & existing Hadoop Data:**Spark is compatible with both versions of [**Hadoop ecosystem**](http://data-flair.training/blogs/hadoop-ecosystem-components/). Be it [**YARN (Yet Another Resource Negotiator)**](http://data-flair.training/blogs/category/yarn/) or SIMR (Spark in MapReduce). It can read anything existing Hadoop data that’s what makes it suitable for migration of pure Hadoop-MapReduce applications. It can run independently too.

[**Lazy Evaluation**](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/): Another outstanding feature of Spark which is call by need or memorization. It waits for instructions before providing final result which saves significant time.

**Active, progressive and expanding community:**Built by wide set of developers from over 100 companies. It has active mailing state and JIRA for issue tracking. It is most active component in Apache repository.

**3) What are the languages in which Apache Spark create API?**

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.

**4) Compare Apache Hadoop and Apache Spark.**

[**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/)

Stores data in local disk

Slow speed

Suitable for batch processing

External schedulers required

High latency

No in-built interactive mode.

[**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/)

Stores data in-memory

Faster speed

Suitable for batch and real-time processing

Schedules tasks itself

Low latency

Has interactive mode

**5) Can we run Apache Spark without Hadoop?**

Yes, [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) can run without [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), standalone, or in the cloud. Spark doesn't need a Hadoop cluster to work. Spark 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. Spark is a meant for distributed computing. In this case, the data is distributed 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.

**6) What are the benefits of Spark over MapReduce?**

[**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is easy to program and don't require that much hand coding whereas [**MapReduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) is not that easy in terms of programming and requires lots of hand coding

It has interactive mode whereas in MapReduce there is no built-in interactive mode, MapReduce is developed for batch processing.

For data processing Spark can use streaming, machine learning, and batch processing whereas Hadoop MapReduce can use the batch engine. Spark is general purpose cluster computation engine.

Spark executes batch processing jobs about 10 to 100 times faster than Hadoop MapReduce.

Spark uses an abstraction called [**RDD**](http://data-flair.training/blogs/apache-spark-rdd-tutorial/) which makes Spark feature rich, whereas map reduce doesn't have any abstraction

Spark uses lower latency by caching partial/complete results across distributed nodes whereas MapReduce is completely disk-based.

**7) Why is Apache Spark faster than Hadoop MapReduce?**

[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is faster than [**Apache Hadoop**](http://data-flair.training/forums/topic/why-apache-spark-is-faster-than-hadoop) due to below reasons:

1) Apache Spark provides [**in-Memory computating**](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**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) writes intermediate results back to Disk and reads it back.

2) 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

3) 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.

**8) What are the drawbacks of Apache Spark?**

The various disadvantages of [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) are:

There is no support for real-time processing in Spark. It supports near real-time processing of live data. The real time data is divided into batches of the predefined interval. And also the result of the computation is returned in batches.

Problem with small file comes when we use Spark with a large number of small files. As [**HDFS**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) allows a limited number of large files. Another place where Spark legs behind are we store the data gzipped in S3. This pattern is very nice except when there are lots of small gzipped files.

There is no dedicated file management system. It does not have its own file management system, so it relies on some other platform. For example, [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/) or another cloud-based platform.

It is expensive. Because to keep data in-memory is quite expensive. Also, the memory consumption is very high, and it is not handled in a user-friendly manner. Apache Spark requires lots of RAM to run in-memory, thus the cost of Spark is quite high.

Apache Spark lags behind in a number of algorithms. MLlib legs behind in a number of an available algorithm like Tanimoto distance.

The job requires being manually optimized and adequate to specific datasets. The partitioning and caching are controlled manually for an authentic solution.

In Spark, the data iterates in batches. Also, scheduling and execution of each iteration take place separately.

High latency than Apache Flink.

Spark does not support record based window criteria. It only has time-based window criteria.

Back pressure Handling - Back pressure is buildup of data at an input-output when the buffer is full and not able to receive the additional incoming data. No data is transferred until the buffer is empty. Apache Spark is not capable of handling pressure implicitly rather it is done manually.

**9) Explain the processing speed difference between Hadoop and Apache Spark.**

[**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**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) is batch oriented processing tool, it takes large dataset in the input, processes it and produces a result.  
Hadoop MapReduce adopted 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**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) 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 reduce because it caches input data in memory by [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/).

**10) Explain various Apache Spark ecosystem components. In which scenarios can we use these components?**

Below are the [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) ecosystem components:

**Spark Core**  
Spark Core is the base of Spark for parallel and distributed processing of huge datasets. It is in charge of all the essential I/O functionalities, programming, and observance the roles on spark clusters. It is also responsible for task dispatching, and networking with different storage systems, [**fault tolerance**](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/), and economical memory management. It uses special collection referred to as [**RDD (Resilient Distributed Datasets)**](http://data-flair.training/blogs/rdd-in-apache-spark/).

[**Spark SQL**](http://data-flair.training/blogs/spark-sql-tutorial/)  
SparkSQL is module/component in Apache Spark that is employed to access structured and semi-structured information. It is a distributed framework that is tightly integrated with varied spark programming like [**Scala**](http://data-flair.training/blogs/category/scala/), Python, and Java. It supports relative process in Spark programs via RDD further as on external data source. It is used for manipulating and taking information in varied formats. The means through that {we can|we will|we square measure able to} act with Spark SQL are SQL/HQL, [**DataFrame**](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/) API, and Datasets API. It provides higher improvement.  
The main abstraction in SparkSQL is information sets that act on structured data. It translates ancient SQL and HiveQL queries into Spark jobs creating Spark accessible wide. It supports real-time data analytics, data streaming SQL.

**SparkSQL defines 3 varieties of function:**

**Built-in perform or user-defined function:** Object comes with some functions for column manipulation. Using Scala we are able to outlined user outlined perform.

**Aggregate Function:** Operates on the cluster of rows and calculates one come back price per cluster.

**Window Aggregate:** Operates on cluster of rows and calculates one come back price every row in an exceeding cluster.

**Different type of APIs for accessing SparkSQL:  
SQL:**Executing SQL queries or [**Hive**](http://data-flair.training/blogs/category/hive/) queries, result are going to became in variety of DataFrame.

**DataFrame:**It is similar to relative table in SparkSQL. It is distributed the assortment of tabular information having rows and named column. It will perform filter, intersect, join, sort mixture and much more. It 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.

**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 be modeled in some data-structure. It 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 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 is accepted 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 is a distributed graph process framework on prime of Apache Spark. because it is predicated 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.

**11) Explain 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.

**12) Define Spark-SQL.**

Spark SQL is a Spark interface to work with Structured and Semi-Structured data (data that as defined fields i.e. tables). It provides abstraction layer called [**DataFrame**](https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/) and [**DataSet**](https://data-flair.training/blogs/apache-spark-dataset-tutorial/) through with we can work with data easily. One can say that DataFrame is like a table in a relational database. Spark SQL can read and write data in a variety of Structured and Semi-Structured formats like Parquets, JSON, Hive. Using SparkSQL inside Spark application is the best way to use it. This empowers us to load data and query it with SQL. we can also combine it with “regular” program code in Python, Java or [**Scala**](https://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/).

**13) How do we represent data in Spark?**

There are 3 ways to represent data in Apache Spark:  
RDD, DataFrame, and Dataset

[**RDD (Resilient Distributed Dataset)**](http://data-flair.training/blogs/rdd-in-apache-spark/): It is the fundamental data structure of [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/). It is an immutable collection of object. These object computes on diferent nodes of the cluster. There are three ways to create an RDD: by parallelizing already existing collection in dataset, from other RDD and from data in stable storage. RDD also provides two types of operation namely [**Transformation and Action**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

[**DataFrame**](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/): These are the dataset that are arranged in named columns. These are relational data items with good optimization technique. Although Spark DataFrame is above RDD, it possesses all the features of an RDD. DataFrames are ahead of RDD as it provides memory management plan.

[**Dataset**](http://data-flair.training/blogs/apache-spark-dataset-tutorial/): It is strongly typed data structure in [**SparkSQL**](http://data-flair.training/blogs/spark-sql-tutorial/). It maps to relational schema. Dataset gives the benefits of both type safety and Object oriented programming interface. Dataset clubs the property of both DataFrame and RDD. And thus, provides better functional programming interface.

**14) What is Resilient Distributed Dataset (RDD) in Apache Spark? How does it make spark operator rich?**

It is the fundamental data structure of Apache Spark and provides core abstraction. It is a collection of immutable objects which computes on different nodes of the cluster. It is resilient as well as lazy in nature apart from being statically typed.

RDDs support two kinds of operations.  
1. Transformations – It applies some function on a RDD and creates a new RDD. It does not modify the original RDD. Also, the new RDD keeps a pointer to it’s parent RDD. When a transformation is called Spark does not execute it immediately , instead it creates a lineage (a track of all the transformations that has to be applied on that RDD including from where it has to read the data)  
2. Actions – It is an operation that triggers computations and returns a value.

RDDs are divided into smaller chunks called partitions (logical chunks of data), when some actions are executed, a task is launched per partition. The number of partitions are directly responsible for parallelism. Spark automatically decides the number of partitions that an RDD has to be divided into but we can control it using repartition or coalesce transformations. These partitions are then distributed across the nodes in the cluster.

There are three ways to create an RDD in Spark – Data from an external file or stable storage, from an existing RDD and, parallelizing collections in the driver program.

One of the most important characteristics of RDD is Caching. We can cache RDD in memory by calling rdd.cache() , which then loads the partitions into the memory of the node that holds it. This improves the performance to a great extent.

**15) What are the major features/characteristics of RDD (Resilient Distributed Datasets)?**

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 block2. List of dependencies on parent RDD: None3. A function to compute each partition: read respective HDFS block4. Optional Preferred location: HDFS block location5. Optional partitioned info: None

**#FilteredRDD :**

Properties of FilteredRDD:

1. List or Set of partitions: No. of partitions same as parent RDD2. 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 it4. Optional Preferred location: None (Ask Parent)5. Optional partitioned info: None

**16) How is RDD in Apache 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).

**17) Explain the operation transformation and action in Apache Spark RDD.**

Transformations are lazy evaluated operations on RDD that create one or many new [**RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/), 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 be changed or modified 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 be executed 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.

**18) How to process data using Transformation operation in Spark?**

Transformations are lazy evaluated operations on RDD that create one or many new [**RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/), 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 be changed or modified 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 be executed 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.

**12) Explain briefly what is Action in Apache Spark? How is final result generated using an action?**

**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 be evaluated. 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).

**13) Compare Transformation and Action in Apache Spark.**

**Transformations**[**create new 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/)  
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 triggers 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()

**14) How to identify that the given operation is transformation or action?**

In order to identify the operation, one need to look at the return type of an operation.

**If operation returns a new RDD in that case an operation is 'Transformation'**

**If 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.

**15) What are the ways to create RDDs in Apache Spark? Explain.**

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

val rdd1 = Array(1,2,3,4,5)

val rdd2 = sc.parallelize(rdd1)

**OR**

val myList = sc.parallelize(List(1 to 1000), 5) where 5 is the number of partitions

[If we do not specify then default partition is 1

**Create by loading an external Dataset**

In Spark, distributed dataset can be formed from any data source supported by [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), including the local file system, [**HDFS**](http://data-flair.training/blogs/category/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)**

import org.apache.spark.sql.SparkSession

def main(args: Array[String]):Unit = {

object DataFormat {

val spark = SparkSession.builder.appName("AvgAnsTime").master("local").getOrCreate()

val dataRDD = spark.read.csv("path/of/csv/file").rdd

**ii) json (String path)**

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

**iii) textFile (String path)**

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.

val words=spark.sparkContext.parallelize(Seq("the", "quick", "brown", "fox", "jumps", "over", "the", "lazy",

"dog"))

val wordPair = words.map(w => (w.charAt(0), w))

wordPair.foreach(println)

**16) Explain benefits of lazy evaluation in RDD in Apache Spark?**

Lazy evaluation means that [**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) does not evaluate each [**transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) as they arrive, but instead queues them together and evaluate all at once, as an Action is called.

The benefit of this approach is that Spark can make optimization decisions after it had a chance to look at the [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) in entirety. This would not be possible if it were to execute everything as soon as it got it. As a result, a large volume of Network I/O can be avoided which, otherwise, could have caused a serious bottleneck.

**Example:**  
Suppose we have a file **words.txt** containing the following lines:

line1 word1

line2 word2 word1

line3 word3 word4

line4 word1

Next, we apply the following operations.

scala> val lines = sc.textFile("words.txt")

scala> val filtered = lines.filter(line => line.contains("word1"))

scala> filtered.first()

res0: String = line1 word1

If Spark were to evaluate each line immediately, it would end up reading the whole file, then applying a filter transformation and then displaying the first line from the filtered result. This would mean a lot of extra work and unnecessary memory utilization.

On the other hand, in Lazy evaluation mode, Spark first builds the entire DAG and then, using optimization techniques it understands that reading the entire file is not necessary. The same result can be achieved by just reading the first line of the file.

**17) Why is transformation lazy operation in Apache Spark RDD? How is it useful?**

Whenever a [**transformation operation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) is performed in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/), it is lazily evaluated. It won't be executed 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.

<stronh>For example, consider a 1TB of 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. 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 be loaded into memory, all of the data within the file will be splitted based 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 be optimized, the operation will be combined / merged 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.

**18) What is RDD lineage graph? How does it enable fault-tolerance in Spark?**

Lineage Graph is the graph of all the parent RDDs for an [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/).

By applying a different [**transformation on an RDD**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) results in lineage graph.

When one derives the new RDD from existing (previous) RDD using transformation, Spark keeps the track of all the dependencies between RDD is called lineage graph.

Lineage Graph is useful for scenarios mentioned below:

(1) When there is a demand for computing the new RDD.  
(2) To recover the lost data if part of [**persisted RDD**](http://data-flair.training/blogs/apache-spark-rdd-persistence-caching/) is lost.

In other words, Lineage Graph is a graph of all transformation operation that needs to execute when an action operation is called.

**19) What are the types of transformation in RDD in Apache Spark?**

There are two kinds of transformations:

Narrow transformations

Wide transformations

**Narrow transformations:**  
Narrow transformations are the result of map, filter and in which data to be transformed  
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**](http://data-flair.training/blogs/rdd-in-apache-spark/).

**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**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/)must execute RDD shuffle, which transfers data across cluster  
and results in a new stage with a new set of partitions.

**20) What is Map() operation 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.

**21) Explain the flatMap operation on Apache Spark RDD.**

When one want to produce multiple elements (values) for each input element, flatMap() is used.

As with map(), flatMap() also takes function as an input.

Output of the function is a List of the element through which we can iterate. (i.e. function can return 0 or more element for each input element)

Simple use of flatMap() is splittin up an input line (string) into words.

**Example**

val fm1 = sc.parallelize(List("Good Morning", "Data Flair", "Spark Batch"))

val fm2 = fm1.flatMap(y => y.split(" "))

fm2.foreach{println}

**Output is as follows:**

Good  
Morning  
Data  
Flair  
Spark  
Batch

**22) Describe the distnct(),union(),intersection() and substract() transformation in Apache Spark RDD.**

**distnct() transformation**

If one want only unique elements in a [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) in that case one can use d1.distnct() where d1 is RDD

**Example**

val d1 = sc.parallelize(List("c","c","p","m","t"))

val result = d1.distnct()

result.foreach{println}

**OutPut:**  
p  
t  
m  
c

**union() transformation**

Its simplest set operation.

rdd1.union(rdd2) which outputs a RDD which contains the data from both sources.

If the duplicates are present in the input RDD, output of union() transformation will contain duplicate also which can be fixed using distinct().

**Example**

val u1 = sc.parallelize(List("c","c","p","m","t"))

val u2 = sc.parallelize(List("c","m","k"))

val result = u1.union(u2)

result.foreach{println}

**Output:**  
c  
c  
p  
m  
t  
c  
m  
k

**intersection() transformation**

intersection(anotherrdd) returns the elements which are present in both the RDDs.

intersection(anotherrdd) remove all the duplicate including duplicated in single RDD

val is1 = sc.parallelize(List("c","c","p","m","t"))

val is2 = sc.parallelize(List("c","m","k"))

val result = is1.union(is2)

result.foreach{println}

**Output :**  
m  
c

**subtract() transformation**

Subtract(anotherrdd).

It returns an RDD that has only value present in the first RDD and not in second RDD.

**Example**

val s1 = sc.parallelize(List("c","c","p","m","t"))

val s2 = sc.parallelize(List("c","m","k"))

val result = s1.subtract(s2)

result.foreach{println}

**Output:**  
t  
p

**23) Explain join() operation in Apache Spark**

> join() is transformation.  
> It's in package **org.apache.spark.rdd.pairRDDFunction**  
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]Permalink

Return an RDD containing all pairs of elements with matching keys in this and other.  
Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in this and (k, v2) is in other. Performs a hash join across the cluster.

**From** :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#213_Join>

It is joining two datasets. When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Example1:

val rdd1 = sc.parallelize(Seq(("m",55),("m",56),("e",57),("e",58),("s",59),("s",54)))

val rdd2 = sc.parallelize(Seq(("m",60),("m",65),("s",61),("s",62),("h",63),("h",64)))

val joinrdd = rdd1.join(rdd2)

joinrdd.collect

**Output:**

Array[(String, (Int, Int))] = Array((m,(55,60)), (m,(55,65)), (m,(56,60)), (m,(56,65)), (s,(59,61)), (s,(59,62)), (s,(54,61)), (s,(54,62)))

**Example2:**

val myrdd1 = sc.parallelize(Seq((1,2),(3,4),(3,6)))

val myrdd2 = sc.parallelize(Seq((3,9)))

val myjoinedrdd = myrdd1.join(myrdd2)

myjoinedrdd.collect

**Output:**  
Array[(Int, (Int, Int))] = Array((3,(4,9)), (3,(6,9)))

**24) Explain leftOuterJoin() and rightOuterJoin() operation in Apache Spark.**

> Both leftOuterJoin() and rightOuterJoin() are transformation.  
> Both in package org.apache.spark.rdd.PairRDDFunctions

**leftOuterJoin() :**

def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]

Perform a left outer join of this and other. For each element (k, v) in this, the resulting RDD will either contain all pairs (k, (v, Some(w))) for w in other, or the pair (k, (v, None)) if no elements in other have key k. Hash-partitions the output using the existing partitioner/parallelism level.

leftOuterJoin() performs a join between two RDDs where the keys must be present in first RDD

**Example :**

val rdd1 = sc.parallelize(Seq(("m",55),("m",56),("e",57),("e",58),("s",59),("s",54)))

val rdd2 = sc.parallelize(Seq(("m",60),("m",65),("s",61),("s",62),("h",63),("h",64)))

val leftjoinrdd = rdd1.leftOuterJoin(rdd2)

leftjoinrdd.collect

**Output :**  
Array[(String, (Int, Option[Int]))] = Array((s,(59,Some(61))), (s,(59,Some(62))), (s,(54,Some(61))), (s,(54,Some(62))), (e,(57,None)), (e,(58,None)), (m,(55,Some(60))), (m,(55,Some(65))), (m,(56,Some(60))), (m,(56,Some(65))))

**rightOuterJoin():**  
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]

Perform a right outer join of this and other. For each element (k, w) in other, the resulting RDD will either contain all pairs (k, (Some(v), w)) for v in this, or the pair (k, (None, w)) if no elements in this have key k. Hash-partitions the resulting RDD using the existing partitioner/parallelism level.

It performs the join between two RDDs where the key must be present in other RDD

**Example:**

val rdd1 = sc.parallelize(Seq(("m",55),("m",56),("e",57),("e",58),("s",59),("s",54)))

val rdd2 = sc.parallelize(Seq(("m",60),("m",65),("s",61),("s",62),("h",63),("h",64)))

val rightjoinrdd = rdd1.rightOuterJoin(rdd2)

rightjoinrdd.collect

**Array[(String, (Option[Int], Int))] = Array((s,(Some(59),61)), (s,(Some(59),62)), (s,(Some(54),61)), (s,(Some(54),62)), (h,(None,63)), (h,(None,64)), (m,(Some(55),60)), (m,(Some(55),65)), (m,(Some(56),60)), (m,(Some(56),65)))**

**25) Define fold() operation in Apache Spark.**

fold() is an action. It is wide operation (i.e. shuffle data across multiple partitions and output a single value)

It takes function as an input which has two parameters of the same type and outputs a single value of the input type.

It is similar to reduce but has one more argument 'ZERO VALUE' (say initial value) which will be used in the initial call on each partition.

**def fold(zeroValue: T)(op: (T, T) ⇒ T): T**

Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value". The function op(t1, t2) is allowed to modify t1 and return it as its result value to avoid object allocation; however, it should not modify t2.

This behaves somewhat differently from fold operations implemented for non-distributed collections in functional languages like Scala. This fold operation may be applied to partitions individually, and then fold those results into the final result, rather than apply the fold to each element sequentially in some defined ordering. For functions that are not commutative, the result may differ from that of a fold applied to a non-distributed collection.

zeroValue: The initial value for the accumulated result of each partition for the op operator, and also the initial value for the combine results from different partitions for the op operator - this will typically be the neutral element (e.g. Nil for list concatenation or 0 for summation)  
Op: an operator used to both accumulate results within a partition and combine results from different partitions

**Example :**

val rdd1 = sc.parallelize(List(1,2,3,4,5),3)

rdd1.fold(5)(\_+\_)

**Output :  
Int = 35**

val rdd1 = sc.parallelize(List(1,2,3,4,5))

rdd1.fold(5)(\_+\_)

**Output :  
Int = 25**

val rdd1 = sc.parallelize(List(1,2,3,4,5),3)

rdd1.fold(3)(\_+\_)

**Int = 27**

**26) What are the exact differences between reduce and fold operation in Spark?**

**Reduce:**  
Reduce methods walk through the elements in a collection,  
applying your function to neighboring elements to yield a new result,  
which is then compared to the next element in the sequence to yield a new result

def reduce[T]((value1,value1) => res)

**Fold:**  
Fold also works similar to Reduce and aggregate over a collection by executing an operation  
but with a specified initial value

def fold[T](acc:T)((acc,value) => acc)

**Example:**

Finding max in a given [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/)

val employeeData = List(("Ram",1000.0),("Vishnu",2000.0),("Ravi",7000.0))

val employeeRDD = sc.makeRDD(employeeData)

val dummyEmployee = ("ABC",0.0);

val maxSalaryEmployee = employeeRDD.fold(dummyEmployee)((acc,employee) => {

if(acc.\_2 < employee.\_2) employee else acc})

println("employee with maximum salary is"+maxSalaryEmployee)

**27) Explain first() operation in Apache Spark.**

> It is an action.  
> It returns the first element of the [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/).

**Example** :

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt")

rdd1.count

rdd1.first

**Output :**  
Long: 20  
String : DataFlair is the leading technology training provider

**28) Explain coalesce operation in Apache Spark.**

> It is a [**transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).  
> 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 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.

**29) How does pipe operation writes the result to standard output in Apache Spark?**

It 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 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.

**30) List out the** **difference between textFile and wholeTextFile in Apache Spark.**

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 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

**31) Define Partition and Partitioner in Apache Spark.**

**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.

**32) How many partitions are created by default in Apache Spark RDD?**

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.

**Example1:**

No Partition is not specified

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt")

**Example2:**

Following code create the RDD of 10 partitions, since we specify the no. of partitions.

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt", 10)

One can query about the number of partitions in following way :

rdd1.partitions.length

<strong>

OR

</strong>

rdd1.getNumPartitions

Best case Scenario is that we should make RDD in following way:

**33) 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.

**34) Define paired RDD in Apache Spark?**

Pair RDD is a special type of [**RDD in Apache Spark**](http://data-flair.training/blogs/rdd-in-apache-spark/) which extends its capabilities from a normal RDD and adds its own set of [**transformations**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/). The elements in a Pair RDD are key-value pairs which are particularly very helpful where the user needs to perform similar operations on each key.

**For example**: reduceByKey(), aggregateByKey(), foldByKey(), sortByKey() etc.

val file = sc.textFile("/path/to/file")  
val words = file.flatMap(line => line.split(" ")) // words is a normal RDD of String  
val tuple = words.map(word => (word, 1)) // tuple is a Pair RDD of (string, 1)  
val wc = tuple.reduceByKey((a, b) => a + b) // performing sum of count for each word (i.e.) key

**35) What are the differences between Caching and Persistence method 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.

**36) Define the run-time architecture of Spark?**

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.

**37)****What is the use of Spark driver, where it gets executed on the cluster?**

A [**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) driver (aka an application’s driver process) is a JVM process that hosts [**SparkContext**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/) for a Spark application. It is the master node in a Spark application.  
It is the cockpit of jobs and tasks execution (using [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)Scheduler and Task Scheduler).  
It hosts Web UI for the environment.  
It splits a Spark application into tasks and schedules them to run on executors.  
A driver is where the task scheduler lives and spawns tasks across workers.  
A driver coordinates workers and overall execution of tasks.

**38) 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

**39) Define various running modes of Apache Spark.**

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 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

**40) What is the Standalone mode in Spark cluster?**

[**View Answer >>**](http://data-flair.training/forums/topic/what-is-standalone-mode)

**41) Write the command to start and stop the Spark in an interactive shell?**

**Command to start the interactive shell in Scala:**  
>>>>bin/spark-shell  
**First go the spark directory i.e.**

hdadmin@ubuntu:~$ cd spark-1.6.1-bin-hadoop2.6/

hdadmin@ubuntu:~/spark-1.6.1-bin-hadoop2.6$ bin/spark-shell

------------------------------------------------------------------------------------------------------------------------------  
**Command to stop the interactive shell in Scala:**  
scala>Press (Ctrl+D)  
**One can see the following message**  
scala> Stopping spark context.

**42) Define 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/)

**43) Define SparkSession in Apache Spark? Why is it needed?**

Starting from [**Apache Spark**](http://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.

**44) In what ways SparkSession different from SparkContext?**

[**Spark Context:**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/)  
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.

**Example:**  
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.

**45) List out the various advantages of DataFrame over RDD in Apache Spark.**

**Introduction**  
**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**](http://data-flair.training/forums/topic/what-are-the-advantages-of-dataframe-in-apache-spark), DataFrames are evaluated lazily([**Lazy Evaluation**](http://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**](http://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 be extended to support any third-party data formats or sources. Existing third-party extensions already include Avro, CSV, ElasticSearch, and Cassandra.

**46) 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.**

**47) What is 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.

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

**Introduction to DataSets**

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 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 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 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.

**49) What are the ways to run Spark over Hadoop?**

[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is open source cluster processing engine which handles several different data types and data sources at the same time. Spark is100 times faster than Hadoop.

For on-premise deployments, Apache Spark doesn’t have a storage engine. It must rest on top of a read-write storage platform like the Hadoop Distributed File System (HDFS).

Spark can read and write data from and to HDFS, and other storage systems. For example, HBase and Amazon’s S3. Hadoop Users can also enrich their processing capabilities by combining Spark with Hadoop MapReduce, HBase, and other big data frameworks.

There are three methods to run Spark in a Hadoop cluster:  
**standalone, YARN, and SIMR.**

**Standalone deployment**: In Standalone Deployment, one can statically allocate resources on all or a subset of machines in a Hadoop cluster and run Spark side by side with Hadoop MR. The user can run arbitrary Spark jobs on their HDFS data. Generally, Hadoop 1.x users will use this method.

**Hadoop Yarn deployment**: Hadoop users who have already deployed or are planning to deploy Hadoop Yarn can simply run Spark on YARN without any pre-installation or administrative access required. Thus users can easily integrate Spark in their Hadoop stack and take advantage of Spark, as well as of other components running on Spark.

**Spark In MapReduce (SIMR)**: For the Hadoop users that are not running YARN yet, another option, in addition to the standalone deployment, It uses SIMR to launch Spark jobs inside MapReduce. Using SIMR, one can experiment with Spark. And also uses its shell within a couple of minutes after downloading. It lowers the barrier of deployment and plays with Spark.

Spark interoperates not only with Hadoop but also with other popular big data technologies.

**Apache Hive**: Spark enables Apache Hive users to run their unmodified queries much faster. Hive is a data warehouse solution running on Hadoop, while Shark allows the Hive framework to run on Spark in place of Hadoop.

AWS EC2: Users can easily run Spark on top of Amazon’s EC2. Either using the scripts that come with Spark, or the hosted versions of Spark on Amazon’s Elastic MapReduce.

**Apache Mesos**: Spark runs on top of Mesos. Mesos is a cluster manager system that provides efficient resource isolation across distributed applications, including MPI and Hadoop. Mesos enables fine-grained sharing that allows a Spark job to dynamically take advantage of the ideal resources in the cluster during execution. This results in performance improvements, especially for long-running Spark jobs.

**50) Explain Apache Spark Streaming? How is the processing of streaming data achieved in Apache Spark?**

**Spark Streaming**  
Data arriving continuously, in an unbounded sequence is a data stream. Continuously flowing input data is divided into discrete units with the help of streaming for further processing. Through Stream processing analyzing of streaming data is possible. Also, it is a low latency processing.

In the year 2103 Spark Streaming was introduced to [**Apache Spark**](http://data-flair.training/forums/topic/what-is-spark-streaming). It is an extension of the core Spark API. Streaming offers scalable, high-throughput and [**fault-tolerant**](http://data-flair.training/forums/topic/what-is-spark-streaming) stream processing of live data streams. It is possible to do Data ingestion from many sources. For Example Apache Flume, Kafka, Amazon Kinesis or TCP sockets. And, By using complex algorithms that are expressed with high-level functions processing can be done. For example reduce, map, join and window. Afterwards, processed data can be pushed out to live dashboards, filesystems and databases.

Streaming's Key abstraction is Discretized Stream. It is also known as [**Spark DStream**](http://data-flair.training/forums/topic/what-is-spark-streaming). A stream of data divided into small batches is represented by it. DStreams are built on Spark’s core data abstraction"[**RDDs**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/)". Streaming allows integration with any other Apache Spark components like [**Spark SQL**](http://data-flair.training/forums/topic/what-is-spark-streaming) and [**Spark MLlib**](http://data-flair.training/forums/topic/what-is-spark-streaming).

**51) What is a DStream?**

A Discretized Stream (DStream), it's the fundamental abstraction in [**Spark Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/), is a continuous sequence of [**RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/) of constant kind representing a steady/nonstop stream of information. DStreams may be created from live data like information/data from TCP sockets, Kafka, [**Flume**](http://data-flair.training/blogs/category/flume/), etc employing a StreamingContext or it may be generated by working on existing DStreams exploitation functions like map, window, and reduceByKeyAndWindow. Periodically DStream [**create an RDD**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) which is generated by a parent DStream.

This category contains the fundamental operations offered on all DStreams, like map, filter, and window. additionally, PairDStreamFunctions contains operations offered solely on DStreams of key-value pairs, like groupByKeyAndWindow and be a part of. Through implicit conversions, these operations are offered on any DStream of pairs (e.g., DStream[(Int, Int)]. DStreams internally is characterized by basic properties: - a listing of alternative DStreams depends on - An amount at that the DStream generates an RDD - operate that's want to generate an RDD once on every occasion interval

Discretized Stream may be a sequence of Resilient Distributed Databases that represent a stream of information. DStreams may be created from varied sources like Apache Kafka, [**HDFS**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/), and Apache Flume

**52) Describe different transformations in DStream in Apache Spark Streaming.**

Different transformations in [**DStream**](http://data-flair.training/blogs/apache-spark-dstream-discretized-streams/) in [**Apache Spark Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/) are:

1-**map(func)** -- Return a new DStream by passing each element of the source DStream through a function func.

2-**flatMap(func)** -- Similar to map, but each input item can be mapped to 0 or more output items.

3-**filter(func)** -- Return a new DStream by selecting only the records of the source DStream on which func returns true.

4-**repartition(numPartitions)** -- Changes the level of parallelism in this DStream by creating more or fewer partitions.

5-**union(otherStream)** -- Return a new DStream that contains the union of the elements in the source DStream and  
otherDStream.

6-**count()** -- Return a new DStream of single-element [**RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/) by counting the number of elements in each RDD of the source DStream.

7-**reduce(func)**-- Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one).

8-**countByValue()** -- When called on a DStream of elements of type K, Return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

9-**reduceByKey(func, [numTasks])**-- When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

10-**join(otherStream, [numTasks])** -- When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

11-**cogroup(otherStream, [numTasks])** -- When called on DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

12-**transform(func)** -- Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream.

13-**updateStateByKey(func)** -- Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key.

**53) Explain 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.

**54) Define the level of parallelism and its need in Spark Streaming.**

> In order to reduce the processing time, one need to increase the parallelism.  
> In Spark Streaming, there are three ways to increase the parallelism :  
**(1) Increase the number of receivers :**If there are too many records for single receiver (single machine) to read in and distribute so that is bottleneck. So we can increase the no. of receiver depends on scenario.  
**(2) Re-partition the receive data :**If one is not in a position to increase the no. of receivers in that case redistribute the data by re-partitioning.  
**(3) Increase parallelism in aggregation :**

**55) Define Parquet file format? How to convert data to Parquet format?**

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. Parquet could be a columnar information storage format.

Parquet is created 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 be a format which will be processed by 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.

**56) Define the common faults of the developer while using Apache Spark?**

1) 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.

**57) What is Speculative Execution in 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.

**58) What are the various types of shared variable in Apache Spark?**

There are two types of shared variables available in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/):  
(1) **Accumulators**: used to Aggregate the Information.  
(2) **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)

**59) What are Broadcast Variables?**

Broadcast variables are variables that are shared throughout the cluster. Broadcast variables need to be able to slot in memory on one machine. which means that they mustn't be something super massive, sort of a massive table or large vector. Secondly, broadcast variables area cannot be changed, which means that they can't be modified. If you want to change or modify, accumulators are needed.  
The properties of Broadcast Variable in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) are:

Immutable

Distributed to the cluster

Fit in memory

**60) Describe Accumulator in detail in Apache Spark.**

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

**Introduction of Accumulator :**

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.  
**Examples :**

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

record: org.apache.spark.sql.Dataset[String] = [value: string]</p>

<p>scala> val emptylines = sc.accumulator(0)

warning: there were two deprecation warnings; re-run with -deprecation for details

emptylines: org.apache.spark.Accumulator[Int] = 0</p>

<p>scala> val processdata = record.flatMap(x =>

{

if(x == "")

emptylines += 1

x.split(" ")

})</p>

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

scala> processdata.collect

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

**Output :**  
res0: Array[String] = Array(DataFlair, provides, training, on, cutting, edge, technologies., "", DataFlair, is, the, leading, training, provider,, we, have, trained, 1000s, of, candidates., Training, focues, on, practical, aspects, which, industy, needs, rather, than, theoretical, knowledge., "", DataFlair, helps, the, organizations, to, solve, BigData, Problems., "", Javadoc, is, a, tool, for, generating, API, documentation, in, HTML, format, from, doc, comments, in, source, code., It, can, be, downloaded, only, as, part, of, the, Java, 2, SDK., To, see, documentation, generated, by, the, Javadoc, tool,, go, to, J2SE, 1.5.0, API, Documentation., "", Javadoc, FAQ, -, This, FAQ, covers, where, to, download, the, Javadoc, tool,, how, to, find, a, list, of, known, bugs, and, feature, reque...  
scala> println("No. of Empty Lines : " + emptylines.value)  
No. of Empty Lines : 10

**Explanation and Conclusion of Program :**

In above example, we create an Accumulator[Int] 'emptylines'

Here, we want to find the no. of blank lines during our processing.

After that, we applied flatMap() transformation to process our data but we want to find out no. of empty lines (blank lines) so in flatMap() function if we encounter any blank line, accumulator empty lines increase by 1 otherwise we split the line by space.

After that, we check the output as well as no. of blank lines.

We create the accumulator with the initial value in driver program, by calling sc.accumulator(0) i.e. spark Context.accumulator(initial Value) where the return type of type initalValue {org.apache.spark.Accumulator[T] where T is initalValue]

At the end we call the value() property on the accumulator to access its value.

Please note that, task(s) on worker nodes can not access the value property of accumulator so for in context of task(s), accumulator is write-only variable.

The value() property of accumulator is available only in the driver program.

We can also count the no. of blank lines with the help of transformation/actions but for that, we need an extra operation but with the help of accumulator, we can count the no. of blank lines (or events in broader terms) as we load /process our data.

**61) What are the ways in which 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.

**62) Define the roles of the file system in any framework?**

In order to manage data on computer, one has to interact with the File System directly or indirectly.

When we install Hadoop on our computer, actually there are two file system exists on machine  
*(1) Local File System*,  *(2) HDFS (Hadoop Distributed File System)*

HDFS is sits top on of Local File System.

Following are the genera functions of File System (be it Local or HDFS)

Control the data access mechanism (i.e how data stored and retrived)

Manages the metadata about the Files / Folders (i.e. created date, size etc)

Grants the access permission and manage the securities

Efficiently manage the storage space

**63) How do you parse data in XML? Which kind of class do you use with Java to parse data?**

--> One way to parse the XML data in Java is to use the JDOM library. One can download it and import the JDOM library in your project. You can get the help from Google. If still, required help post your problem in the forum. I will try to give you the solution.

--> For Scala, Scala has inbuilt library for xml parsing. Scala-xml\_2.11-1.0.2 jar (please check the for new version if available).

**64) List some commonly used Machine Learning Algorithm Apache Spark.**

> Basically, there are three types of Machine Learning Algorithms :  
**(1) Supervised Learning Algorithm**  
**(2) Unsupervised Learning Algorithm**  
**(3) Reinforcement Learning Algorithm**

> Most commonly used Machine Learning Algorithm are as follows :  
(1) Linear Regression  
(2) Logistic Regression  
(3) Decision Tree  
(4) K-Means  
(5) KNN  
(6) SVM  
(7) Random Forest  
(8) Naïve Bayes  
(9) Dimensionality Reduction Algorithm  
(10) Gradient Boost and Adaboost

**------------------- --------------------------------------------------------------------------**

**Que 1. What is Apache Spark?**

Apache Spark is a powerful open source flexible data processing framework built around speed, ease of use, and sophisticated analytics.Apache Spark is lightening fast in cluster computing system. Spark can run on [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), standalone or in the cloud and is capable of accessing data from various sources including [**HDFS**](http://data-flair.training/blogs/category/hdfs/), [**HBase**](http://data-flair.training/blogs/category/hbase/), Cassandra or others.

Because of in-cluster computing in Spark, it doesn’t require to keep shuffling things in and out of disk. This results in faster processing of data in spark.

Spark has several advantages compared to other big data and MapReduce technologies like Hadoop and Storm. Few of them are:  
**1.Speed**  
It can run program up to 100 times faster than Hadoop-MapReduce in memory, or 10 times faster on disk.  
**2.Ease of Use**  
Spark has easy-to-use APIs for operating on large data sets. This includes a collection of over 100 operators for  
transforming data and familiar data frame APIs for manipulating semi-structured data.  
We can write applications in Java, [**Scala**](http://data-flair.training/blogs/category/scala/), Python, [**R**](http://data-flair.training/blogs/category/r/).  
**3.A Unified Engine**  
Spark comes with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing.  
**4.Runs Everywhere**  
Spark can run on top of Hadoop, Mesos, standalone, or in the cloud.

[**Spark ecosystem**](https://data-flair.training/blogs/apache-spark-ecosystem-components/)

Below is the brief overview of Spark Ecosystem and its components.  
It consists of:  
[**Spark Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/)**:** Spark Streaming is used for processing the real-time streaming data.  
[**Spark SQL**](http://data-flair.training/blogs/spark-sql-tutorial/)**:** Spark SQL component is a library on top of Spark cluster, by using we can run SQL queries on Spark data.  
**Spark MLlib:** MLlib is Spark’s scalable machine learning library.  
**Spark GraphX:** GraphX is for graphs and graph-parallel computation.

**Que 2. Why 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.

**Que 3. 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.

**Que 4. 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.

**Que 5. Which all languages Apache Spark supports?**

**Apache Spark**is written in [**Scala**](https://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/) language. [**Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/) provides an API in Scala, Python, and Java in order to interact with Spark. It also provides APIs in R language.

**Que 6. 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.

**Que 7. 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

**Que 8. 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/)

**Que 9. 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.

**Que 10. SparkSession vs SparkContext in Apache Spark.**

[**Spark Context:**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/)  
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.

**Example:**  
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.

**Que 11. What are the abstractions of Apache Spark?**

There 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.

**Que 12. How can we create RDD in Apache Spark?**

**Resilient Distributed Datasets (RDD)** is spark's core abstraction which is a [**resilient distributed dataset**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/).  
It is an immutable (read-only) distributed collection of objects.  
Each [**dataset**](https://data-flair.training/blogs/apache-spark-dataset-tutorial/) in RDD is divided into logical partitions,  
which may be computed on different nodes of the cluster.  
Including user-defined classes, RDDs may contain any type of Python, Java, or [**Scala**](https://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/) objects.

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

1. **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")}  
}

**Que 13. 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.

**Que 14. Explain the term paired RDD in Apache Spark**

**Introduction**  
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())

**Que 15. 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).

**Que 16. 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.

**Que 17. What are the types of Apache Spark transformation?**

To understand the types of Transformation better, Let's begin with the brief introduction of Transformation in [**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/).

**Transformation in Spark**  
Spark Transformation is a function that produces new [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. As RDDs are immutable in nature, so input RDDs, cannot be changed.  
An RDD lineage, built by Applying transformation built with the entire parent RDDs of the final RDD(s). In other words, it is also known as RDD operator graph or RDD dependency graph. It is a logical execution plan i.e., it is [**Directed Acyclic Graph (DAG)**](https://data-flair.training/blogs/dag-in-apache-spark/) of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().

Resultant RDD is always dissimilar from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).

Now, let's focus on the question, there are fundamentally two types of transformations:

**1. Narrow transformation –**  
While talking about Narrow transformation, all the elements which are required to compute the records in single partition reside in the single partition of parent RDD. To calculate the result, a limited subset of partition is used. This Transformation are the result of map(), filter().

**2. Wide Transformations -**  
Wide transformation means all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. Partitions may reside in many different partitions of parent RDD. This Transformation is a result of groupbyKey() and reducebyKey().

**Que 18. 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 block2. List of dependencies on parent RDD: None3. A function to compute each partition: read respective HDFS block4. Optional Preferred location: HDFS block location5. Optional partitioned info: None

**#FilteredRDD :**

Properties of FilteredRDD:

1. List or Set of partitions: No. of partitions same as parent RDD2. 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 it4. Optional Preferred location: None (Ask Parent)5. Optional partitioned info: None

**Que 19. 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.

Adding few more points on lineage graph:  
You can check lineage between two RDDs using rdd0.toDebugString. This gives back you the lineage graph from current rdd to all the previous dependencies of RDDs. See below. Whenever you see "+-" symbol from the toDebugString output, it means there will be next stage from the next operation onwards. This is indicates to identify that how many stage are created.

scala> val rdd0 = sc.parallelize(List("Ashok Vengala","Ashok Vengala","DataFlair"))  
rdd0: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[10] at parallelize at <console>:31

scala> val count = rdd0.flatMap(rec => rec.split(" ")).map(word => (word,1)).reduceByKey(\_+\_)  
count: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[13] at reduceByKey at <console>:33

scala> count.toDebugString  
res24: String =  
(2) ShuffledRDD[13] at reduceByKey at <console>:33 []  
+-(2) MapPartitionsRDD[12] at map at <console>:33 []  
| MapPartitionsRDD[11] at flatMap at <console>:33 []  
| ParallelCollectionRDD[10] at parallelize at <console>:31 []

From down to up (i.e, last three rows): These will be performed in stage-0. And the first row(ShuffledRDD): this will operation will be performed in stage-1.

In toDebugString output, we are seeing something like ParallelCollectionRDD, MapPartitionsRDD and ShuffleRDD. These are all implementation of RDD abstract class.

**Que 20.  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.

**Que 21. 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.

**Example1:**

No Partition is not specified

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt")

**Example2:**

Following code create the RDD of 10 partitions, since we specify the no. of partitions.

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt", 10)

One can query about the number of partitions in following way :

rdd1.partitions.length

<strong>

OR

</strong>

rdd1.getNumPartitions

Best case Scenario is that we should make RDD in following way:

**Que 22. What is Spark DataFrames?**

**Introduction**  
**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.

**Que 23. 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.

**Que 24. 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).

**Que 25. 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

**Que 26. 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.

**Que 27. 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.

**Que 28.What is the difference between DAG and Lineage?**

To understand the difference better, we will discuss each topic one by one:

**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.

**Que 29. 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.

**Que 30. What are the limitations of Apache Spark?**

Now-a-days,[**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/) is considered as the next Gen [**Big data**](https://data-flair.training/blogs/what-is-big-data/) tool that is being widely used by industries. But there are certain limitations of Apache Spark. they are:

**Limitations of Apache Spark:**

**1. No File Management System**  
Apache Spark relies on other platforms like Hadoop or some another cloud-based Platform for file management system. This is one of the major issues with Apache Spark.

**2. Latency**  
While working with Apache Spark, it has higher latency.

**3. No support for Real-Time Processing**  
In Spark Streaming, the arriving live stream of data is divided into batches of the pre-defined interval, and each batch of data is treated like Spark [**Resilient Distributed Database**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) (RDDs). Then these RDDs are processed using the operations like map, reduce, join etc. The result of these operations is returned in batches. Thus, it is not real-time processing but Spark is near real-time processing of live data. Micro-batch processing takes place in [**Spark Streaming**](https://data-flair.training/blogs/apache-spark-streaming-tutorial/).

**4. Manual Optimization**  
Manual Optimization is required to optimize Spark jobs. Also, it is adequate to specific datasets. we need to control manually if we want to partition and cache in Spark to be correct.

**5. Less no. of Algorithm**  
Spark MLlib lags behind in terms of a number of available algorithms like Tanimoto distance.

**6. Window Criteria**  
Spark does not support record based window criteria. It only has time-based window criteria.

**7. Iterative Processing**  
In Spark, the data iterates in batches and each iteration is scheduled and executed separately.

**8. Expensive**  
when we want cost-efficient processing of big data [**In-memory**](https://data-flair.training/blogs/apache-spark-in-memory-computing/) capability can become a bottleneck as keeping data in memory is quite expensive. At that time the memory consumption is very high, and it is not handled in a user-friendly manner. The cost of Spark is quite high because Apache Spark requires lots of RAM to run in-memory.

**Que 31. Different Running Modes of Apache Spark**

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 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

**Que 32. What are the different ways of representing data in Spark?**

Basically, there are 3 different ways to represent data in [**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/). Either we can represent it through RDD, or we use DataFrames for same or we can also select DataSets to represent our data in Spark. let's discuss each of them in detail:

**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 data fields and expressions to a query planner.

**Que 33. 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.

**Que 34. 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.

**Que 35. What are shared variables in Apache Spark?**

Shared variables are nothing but the variables that can be used in parallel operations. By default, when [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: **broadcast variables**, which can be used to cache a value in memory on all nodes, and **accumulators**, which are variables that are only “added” to, such as counters and sums.

**Que 36. 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.

**Que 37. What is Apache Spark Machine learning library?**

**Machine learning library in Apache Spark**

It is a scalable Machine learning library that discusses both high speed and high-quality algorithm.

To make machine learning scalable and easy, MLlib is created. There are machine learning libraries that included an implementation of various machine learning algorithms. For example, clustering, regression, classification and collaborative filtering. Some lower level machine learning primitives like generic gradient descent optimization algorithm are also present in MLlib.

In [**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/) Version 2.0 the RDD-based API in spark. MLlib package entered in maintenance mode. In this release, the DataFrame-based API is the primary Machine Learning API for Spark.Therefore, MLlib will not add any new feature to the RDD based API.

[**DataFrame**](https://data-flair.training/forums/topic/what-is-apache-spark-machine-learning-library) based API is that it is more user-friendly than [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) therefore MLlib is switching to DataFrame API. Some of the benefits of using DataFrames are it includes Spark Data sources, [**Spark SQL**](https://data-flair.training/blogs/spark-sql-tutorial/) DataFrame queries Tungsten and [**Catalyst optimizations**](https://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/), and uniform APIs across languages. This Machine learning library also uses the linear algebra package Breeze. Breeze is a collection of libraries for numerical computing and machine learning.

**Que 38. List commonly used Machine Learning Algorithm.**

> Basically, there are three types of Machine Learning Algorithms :  
**(1) Supervised Learning Algorithm**  
**(2) Unsupervised Learning Algorithm**  
**(3) Reinforcement Learning Algorithm**

> Most commonly used Machine Learning Algorithm are as follows :  
(1) Linear Regression  
(2) Logistic Regression  
(3) Decision Tree  
(4) K-Means  
(5) KNN  
(6) SVM  
(7) Random Forest  
(8) Naïve Bayes  
(9) Dimensionality Reduction Algorithm  
(10) Gradient Boost and Adaboost

**Que 39. What is the difference between DSM and RDD?**

On the basis of several features, the difference between RDD and DSM is:

i. **Read**

RDD – The read operation in [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) is either coarse-grained or fine-grained. Coarse-grained meaning we can transform the whole dataset but not an individual element on the dataset. While fine-grained means we can transform individual element on the dataset.  
DSM – The read operation in Distributed shared memory is fine-grained.

ii. **Write**

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

iii. **Consistency**

RDD – The consistency of RDD is trivial meaning it is immutable in nature. We can not realtor the content of RDD i.e. any changes on RDD is permanent. Hence, The level of consistency is very high.  
DSM – The system guarantees that if the programmer follows the rules, the memory will be consistent. Also, the results of memory operations will be predictable.

iv. **Fault-Recovery Mechanism**

RDD – By using lineage graph at any moment, the lost data can be easily recovered in Spark RDD. Therefore, for each transformation, new RDD is formed. As RDDs are immutable in nature, hence, it is easy to recover.  
DSM – [**Fault tolerance**](https://data-flair.training/blogs/fault-tolerance-in-apache-spark/) is achieved by a checkpointing technique which allows applications to roll back to a recent checkpoint rather than restarting.

v. **Straggler Mitigation**

Stragglers, in general, are those that take more time to complete than their peers. This could happen due to many reasons such as load imbalance, I/O blocks, garbage collections, etc.  
An issue with the stragglers is that when the parallel computation is followed by synchronizations such as reductions that causes all the parallel tasks to wait for others.

RDD – It is possible to mitigate stragglers by using backup task, in RDDs.  
DSM – To achieve straggler mitigation, is quite difficult.

vi. **Behavior if not enough RAM**

RDD – As there is not enough space to store RDD in RAM, therefore, the RDDs are shifted to disk.  
DSM – If the RAM runs out of storage, the performance decreases, in this type of systems.

**Que 40. List the advantage of Parquet file in Apache Spark.**

Parquet is an open source file format for [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/). Parquet stores nested data structures in a flat columnar format compared to a traditional approach where data is stored in row-oriented approach, parquet is more efficient in terms of storage and performance.

There are several advantages to columnar formats:

1)Organizing by column allows for better compression, as data is more homogeneous. The space savings are very noticeable at the scale of a Hadoop cluster.  
2)I/O will be reduced as we can efficiently scan only a subset of the columns while reading the data. Better compression also reduces the bandwidth required to read the input.  
3)As we store data of the same type in each column, we can use encoding better suited to the modern processors’ pipeline by making instruction branching more predictable.

**Que 41. What is lazy evaluation in Spark?**

**Lazy evaluation** means the execution will not start until an[**action**](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/) is triggered. [**Transformations**](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/) are lazy in nature i.e. when we call some operation on [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/), it does not execute immediately. [**Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/) adds them to a [**DAG**](https://data-flair.training/blogs/dag-in-apache-spark/) of computation and only when driver requests some data, this DAG actually gets executed

Advantages of lazy evaluation.

1) It is an optimization technique i.e. it provides optimization by reducing the number of queries.  
2) It saves the round trips between driver and cluster, thus speeds up the process.

**Que 42. What are the benefits of Spark lazy evaluation?**

[**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/) uses lazy evaluation in order the benefits:

1) Apply [**Transformations operations on RDD**](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/) or "loading data into RDD" is not executed immediately until it sees an action. Transformations on RDDs and storing data in [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) are lazily evaluated. Resources will be utilized in a better way if Spark uses lazy evaluation.

2) Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together. In case [**MapReduce**](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/), user/developer has to spend a lot of time on how to group operations together in order to minimize the number of MapReduce passes. In spark, there is no benefit of writing a single complex map instead of chaining together many simple operations. The user can organize their spark program into smaller operations. Spark will be managed very efficiently of all the operations by using lazy evaluation

3) Lazy evaluation helps to optimize the disk and memory usage in Spark.

4) In general, when are doing computation on data, we have to consider two things, that is space and time complexities. Using spark lazy evaluation, we can overcome both. The actions are triggered only when the data is required. It reduces overhead.

5) It also saves computation and increases speed. Lazy evaluation will play a key role in saving calculation overhead.  
Only necessary values are computed instead of whole dataset (Its all depends on actions operations, and few  
transformations also)

**Que 43. How much faster is Apache spark than Hadoop?**

[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) works faster when the data fits into memory, Spark processes data in memory which makes it faster in processing while [**MapReduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) pushes data to disk after processing it. Usage of [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) helps to do a lot of optimization, it can optimize and do computations in a single stage, and it also avoids unwanted reducer tasks. Spark can cache partial or complete data in memory allowing to avoid a lot of disks I/O. Commercially Spark is said to 100x faster than [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/).

**Que 44. What are the ways to launch Apache Spark over YARN?**

[**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/) has two modes of running applications on YARN: cluster and client  
spark-submit or spark-shell --master yarn-cluster or --master yarn-client

**Que 45. Explain various cluster manager in Apache Spark?**

Apache Spark uses three types of Cluster Manager:

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.

**Que 46. 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 TaskSchedulerImp1withspark.speculation enabled. It executes periodically everyspark.speculation.intervalafter the initialspark.speculation.interval passes.

**Que 47. How can data transfer be minimized when working with Apache Spark?**

In [**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/), Data Transfer can be reduced by avoiding operation which results in data shuffle.  
Avoid operations like repartition and coalesce, ByKey operations like groupByKey and reduceByKey, and join operations like cogroup and join.

Spark Shared Variables help in reducing data transfer. There two types for shared variables-Broadcast variable and Accumulator.

**Broadcast variable:**

If we have a large dataset, instead of transferring a copy of data set for each task, we can use a broadcast variable which can be copied to each node at one time  
and share the same data for each task in that node. Broadcast variable help to give a large data set to each node.  
First, we need to create a broadcast variable using SparkContext.broadcast and then broadcast the same to all nodes from driver program. Value method  
can be used to access the shared value. The broadcast variable will be used only if tasks for multiple stages use the same data.

**Accumulator:**  
Spark functions used variables defined in the driver program and local copied of variables will be generated. Accumulator are shared variables which help to update  
variables in parallel during execution and share the results from workers to the driver.

**Que 48. What are the cases where Apache Spark surpasses Hadoop?**

The below points are the benefits of [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) over [**Apache Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/)

**1.Speed**

Apache Spark is lightning fast cluster computing tool.  
[**Map reduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) reads and writes from disk and that slows down the processing speed.

**2.Difficulty**

It is easy to program in Spark as it contains many high-level operators with [**RDD – Resilient Distributed Dataset**](http://data-flair.training/blogs/apache-spark-rdd-tutorial/).  
In MapReduce, developers need to hand code each and every operation which makes it very complicated.

**3.Easy to Manage**

Spark performs Batch, Interactive, Machine Learning and Streaming in the same cluster.  
but MapReduce only provides provision for batch processing.

**4.Latency**

Spark provides low latency computing  
Map Reduce is a high latency computing framework

**5.Interactive mode**

Spark can process data interactively  
MapReduce doesn’t have interactive mode

**6.Streaming**

Spark can process real time data through [**Spark streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/).  
With MapReduce, you can only process data in batch mode.

**7.Ease of use**

Spark is easier to use, its abstraction (RDD) enables the user to process data using high-level operators. It provides rich APIs in Java, Scala, Python, and R.  
Map Reduce is complex; we need to handle low-level APIs to process the data, which requires lots of hand coding.

**Que 49. What is action, how it process data in apache spark**

**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 be evaluated. 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).

**Que 50. How is fault tolerance achieved in Apache Spark?**

The basic semantics of fault tolerance in[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is, all the [**Spark RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/) are immutable. It remembers the dependencies between every RDD involved in the operations, through the lineage graph created in the [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/), and in the event of any failure, Spark refers to the lineage graph to apply the same operations to perform the tasks.

There are two types of failures - Worker or driver failure. In case if the worker fails, the executors in that worker node will be killed, along with the data in their memory. Using the lineage graph, those tasks will be accomplished in any other worker nodes. The data is also replicated to other worker nodes to achieve fault tolerance. There are two cases:

1.**Data received and replicated** - Data is received from the source, and replicated across worker nodes. In the case of any failure, the data replication will help achieve fault tolerance.

2.**Data received but not yet replicated**- Data is received from the source but buffered for replication. In the case of any failure, the data needs to be retrieved from the source.

For stream inputs based on receivers, the fault tolerance is based on the type of receiver:

**Reliable receiver** - Once the data is received and replicated, an acknowledgment is sent to the source. In case if the receiver fails, the source will not receive acknowledgment for the received data. When the receiver is restarted, the source will resend the data to achieve fault tolerance.

**Unreliable receiver** - The received data will not be acknowledged to the source. In this case of any failure, the source will not know if the data has been received or not, and it will nor resend the data, so there is data loss.

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:

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

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

**Que 51. What is the role of Spark Driver in spark applications?**

A [**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) driver (aka an application’s driver process) is a JVM process that hosts [**SparkContext**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/) for a Spark application. It is the master node in a Spark application.  
It is the cockpit of jobs and tasks execution (using [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)Scheduler and Task Scheduler).  
It hosts Web UI for the environment.  
It splits a Spark application into tasks and schedules them to run on executors.  
A driver is where the task scheduler lives and spawns tasks across workers.  
A driver coordinates workers and overall execution of tasks.

**Que 52. What is worker node in Apache Spark cluster?**

[**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

**Que 53. Why is Transformation lazy in Spark?**

Whenever a [**transformation operation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) is performed in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/), it is lazily evaluated. It won't be executed 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.

<stronh>For example, consider a 1TB of 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. 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 be loaded into memory, all of the data within the file will be splitted based 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 be optimized, the operation will be combined / merged 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.

**Que 54. Can I run Apache Spark without Hadoop?**

Yes, [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) can run without [**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/), standalone, or in the cloud. Spark doesn't need a Hadoop cluster to work. Spark 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. Spark is a meant for distributed computing. In this case, the data is distributed 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.

**Que 55. Explain Accumulator in Spark.**

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

**Introduction of Accumulator :**

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.  
**Examples :**

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

record: org.apache.spark.sql.Dataset[String] = [value: string]</p>

<p>scala> val emptylines = sc.accumulator(0)

warning: there were two deprecation warnings; re-run with -deprecation for details

emptylines: org.apache.spark.Accumulator[Int] = 0</p>

<p>scala> val processdata = record.flatMap(x =>

{

if(x == "")

emptylines += 1

x.split(" ")

})</p>

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

scala> processdata.collect

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

**Output :**  
res0: Array[String] = Array(DataFlair, provides, training, on, cutting, edge, technologies., "", DataFlair, is, the, leading, training, provider,, we, have, trained, 1000s, of, candidates., Training, focues, on, practical, aspects, which, industy, needs, rather, than, theoretical, knowledge., "", DataFlair, helps, the, organizations, to, solve, BigData, Problems., "", Javadoc, is, a, tool, for, generating, API, documentation, in, HTML, format, from, doc, comments, in, source, code., It, can, be, downloaded, only, as, part, of, the, Java, 2, SDK., To, see, documentation, generated, by, the, Javadoc, tool,, go, to, J2SE, 1.5.0, API, Documentation., "", Javadoc, FAQ, -, This, FAQ, covers, where, to, download, the, Javadoc, tool,, how, to, find, a, list, of, known, bugs, and, feature, reque...  
scala> println("No. of Empty Lines : " + emptylines.value)  
No. of Empty Lines : 10

**Explanation and Conclusion of Program :**

In above example, we create an Accumulator[Int] 'emptylines'

Here, we want to find the no. of blank lines during our processing.

After that, we applied flatMap() transformation to process our data but we want to find out no. of empty lines (blank lines) so in flatMap() function if we encounter any blank line, accumulator empty lines increase by 1 otherwise we split the line by space.

After that, we check the output as well as no. of blank lines.

We create the accumulator with the initial value in driver program, by calling sc.accumulator(0) i.e. spark Context.accumulator(initial Value) where the return type of type initalValue {org.apache.spark.Accumulator[T] where T is initalValue]

At the end we call the value() property on the accumulator to access its value.

Please note that, task(s) on worker nodes can not access the value property of accumulator so for in context of task(s), accumulator is write-only variable.

The value() property of accumulator is available only in the driver program.

We can also count the no. of blank lines with the help of transformation/actions but for that, we need an extra operation but with the help of accumulator, we can count the no. of blank lines (or events in broader terms) as we load /process our data.

**Que 56.  What is the role of Driver program in Spark Application?**

Driver program is responsible for launching various parallel operations on the cluster.

Driver program contains application's main() function.

It is the process which is running the user code which in turn create the SparkContext object, [**create RDDs**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) and performs [**transformation and action operation on RDD**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

Driver program access [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/)through a [**SparkContext**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/) object which represents a connection to computing cluster (From Spark 2.0 onwards we can access SparkContext object through SparkSession).

Driver program is responsible for converting user program into the unit of physical execution called task.

It also defines distributed datasets on the cluster and we can apply different operations on Dataset (transformation and action).

Spark program creates a logical plan called [**Directed Acyclic graph**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) which is converted to physical execution plan by the driver when driver program runs.

**Que 57. How to identify that given operation is Transformation/Action in your program?**

In order to identify the operation, one need to look at the return type of an operation.

**If operation returns a new RDD in that case an operation is 'Transformation'**

**If 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.

**Que 58. Name the two types of shared variable available in Apache Spark.**

There are two types of shared variables available in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/):  
(1) **Accumulators**: used to Aggregate the Information.  
(2) **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)

**Que 59. What are the common faults of the developer while using Apache Spark?**

1) 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.

**Que 60. 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.

**Example1:**

No Partition is not specified

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt")

**Example2:**

Following code create the RDD of 10 partitions, since we specify the no. of partitions.

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt", 10)

One can query about the number of partitions in following way :

rdd1.partitions.length

<strong>

OR

</strong>

rdd1.getNumPartitions

Best case Scenario is that we should make RDD in following way:  
 **numbers of cores in Cluster = no. of partitions**

**Que 61. 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

**Que 62. Explain the filter transformation.**

filter() transformation in Apache Spark takes function as input.

It returns an [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) that only has element that pass the condition mentioned in input function.

**Example**:

val rdd1 = sc.parallelize(List(10,20,40,60))

val rdd2 = rdd2.filter(x => x !=10)

println(rdd2.collect())

**Output**

10

**Que 63. How to start and stop spark in interactive shell?**

**Command to start the interactive shell in Scala:**  
>>>>bin/spark-shell  
**First go the spark directory i.e.**

hdadmin@ubuntu:~$ cd spark-1.6.1-bin-hadoop2.6/

hdadmin@ubuntu:~/spark-1.6.1-bin-hadoop2.6$ bin/spark-shell

------------------------------------------------------------------------------------------------------------------------------  
**Command to stop the interactive shell in Scala:**  
scala>Press (Ctrl+D)  
**One can see the following message**  
scala> Stopping spark context.

**Que 64. Explain sortByKey() operation.**

> **sortByKey()** is a transformation.  
> It returns an RDD sorted by Key.  
> Sorting can be done in (1) Ascending **OR**(2) Descending **OR**(3) custom sorting  
From :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#212_SortByKey>  
They will work with any key type K that has an implicit Ordering[K] in scope. Ordering objects already exist for all of the standard primitive types. Users can also define their own orderings for custom types, or to override the default ordering. The implicit ordering that is in the closest scope will be used.

When called on [**Dataset**](https://data-flair.training/blogs/apache-spark-dataset-tutorial/)  
of (K, V) where k is Ordered returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the ascending argument.

Example :

<br />

val rdd1 = sc.parallelize(Seq(("India",91),("USA",1),("Brazil",55),("Greece",30),("China",86),("Sweden",46),("Turkey",90),("Nepal",977)))<br />

val rdd2 = rdd1.sortByKey()<br />

rdd2.collect();<br />

**Output:  
Array[(String,Int)] = (Array(Brazil,55),(China,86),(Greece,30),(India,91),(Nepal,977),(Sweden,46),(Turkey,90),(USA,1)**

<br />

val rdd1 = sc.parallelize(Seq(("India",91),("USA",1),("Brazil",55),("Greece",30),("China",86),("Sweden",46),("Turkey",90),("Nepal",977)))<br />

val rdd2 = rdd1.sortByKey(false)<br />

rdd2.collect();<br />

**Output:  
Array[(String,Int)] = (Array(USA,1),(Turkey,90),(Sweden,46),(Nepal,977),(India,91),(Greece,30),(China,86),(Brazil,55)**

**Que 65. Explain distnct(),union(),intersection() and substract() transformation in Spark**

**distnct() transformation**

If one want only unique elements in a [**RDD**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) in that case one can use d1.distnct() where d1 is RDD

**Example**

val d1 = sc.parallelize(List("c","c","p","m","t"))

val result = d1.distnct()

result.foreach{println}

**OutPut:**  
p  
t  
m  
c

**union() transformation**

Its simplest set operation.

rdd1.union(rdd2) which outputs a RDD which contains the data from both sources.

If the duplicates are present in the input RDD, output of union() transformation will contain duplicate also which can be fixed using distinct().

**Example**

val u1 = sc.parallelize(List("c","c","p","m","t"))

val u2 = sc.parallelize(List("c","m","k"))

val result = u1.union(u2)

result.foreach{println}

**Output:**  
c  
c  
p  
m  
t  
c

**intersection() transformation**

intersection(anotherrdd) returns the elements which are present in both the RDDs.

intersection(anotherrdd) remove all the duplicate including duplicated in single RDD

val is1 = sc.parallelize(List("c","c","p","m","t"))

val is2 = sc.parallelize(List("c","m","k"))

val result = is1.union(is2)

result.foreach{println}

**Output :**  
m  
c

**subtract() transformation**

Subtract(anotherrdd).

It returns an RDD that has only value present in the first RDD and not in second RDD.

**Example**

val s1 = sc.parallelize(List("c","c","p","m","t"))

val s2 = sc.parallelize(List("c","m","k"))

val result = s1.subtract(s2)

result.foreach{println}

**Output:**  
t  
p

**Que 66.Explain foreach() operation in apache spark**

> foreach() operation is an action.  
> It do not return any value.  
> It executes input function on each element of an RDD.

From :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#39_Foreach>

It executes the function on each item in [**RDD**](https://data-flair.training/forums/topic/explain-foreach-operation). It is good for writing database or publishing to web services. It executes parameter less function for each data items.

**Example:**

val mydata = Array(1,2,3,4,5,6,7,8,9,10)

val rdd1 = sc.parallelize(mydata)

rdd1.foreach{x=>println(x)}

OR

rdd1.foreach{println}

**Que 67.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.

**Example:**

val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)

val group = data.groupByKey().collect()

group.foreach(println)

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

**Example:**

val words = Array("one","two","two","four","five","six","six","eight","nine","ten")

val data = spark.sparkContext.parallelize(words).map(w => (w,1)).reduceByKey(\_+\_)

data.foreach(println)

**Que 68. 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.

From :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#24_MapPartitions>

2.4. 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>).

**Que 69. 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.

**Que 70. 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.

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

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

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)

scala>

scala> val data1 = data.map(line => line.split(" "))  
data1: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:26

scala>

scala> val data2 = data.flatMap(line => line.split(" "))  
data2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at flatMap at <console>:26

scala>

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))

scala>

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)

**Que 71.Explain fold() operation in Spark.**

fold() is an action. It is wide operation (i.e. shuffle data across multiple partitions and output a single value)

It takes function as an input which has two parameters of the same type and outputs a single value of the input type.

It is similar to reduce but has one more argument 'ZERO VALUE' (say initial value) which will be used in the initial call on each partition.

**def fold(zeroValue: T)(op: (T, T) ⇒ T): T**

Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value". The function op(t1, t2) is allowed to modify t1 and return it as its result value to avoid object allocation; however, it should not modify t2.

This behaves somewhat differently from fold operations implemented for non-distributed collections in functional languages like Scala. This fold operation may be applied to partitions individually, and then fold those results into the final result, rather than apply the fold to each element sequentially in some defined ordering. For functions that are not commutative, the result may differ from that of a fold applied to a non-distributed collection.

zeroValue: The initial value for the accumulated result of each partition for the op operator, and also the initial value for the combine results from different partitions for the op operator - this will typically be the neutral element (e.g. Nil for list concatenation or 0 for summation)  
Op: an operator used to both accumulate results within a partition and combine results from different partitions

**Example :**

val rdd1 = sc.parallelize(List(1,2,3,4,5),3)

rdd1.fold(5)(\_+\_)

**Que 72. 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.**

**Que 73. Explain values() operation in Apache Spark.**

values() is a transformation.

It returns an RDD of values only.

<br />

val rdd1 = sc.parallelize(Seq((2,4),(3,6),(4,8),(5,10),(6,12),(7,14),(8,16),(9,18),(10,20)))<br />

val rdd2 = rdd1.values<br />

rdd2.collect<br />

**Que 74. Explain keys() operation in Apache spark.**

keys() is a [**transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

It returns an [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) of keys.

val rdd1 = sc.parallelize(Seq((2,4),(3,6),(4,8),(5,10),(6,12),(7,14),(8,16),(9,18),(10,20)))

val rdd2 = rdd1.keys

rdd2.collect

**Output:**

Array[Int] = Array(2, 3, 4, 5, 6, 7, 8, 9, 10)

**Que 75. Explain textFile Vs wholeTextFile in Spark**

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 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

**Que 76. Explain cogroup() operation in Spark**

> It's a transformation.  
> It's in package **org.apache.spark.rdd.PairRDDFunctions**

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.

**Example:**

val myrdd1 = sc.parallelize(List((1,"spark"),(2,"HDFS"),(3,"Hive"),(4,"Flink"),(6,"HBase")))

val myrdd2 = sc.parallelize(List((4,"RealTime"),(5,"Kafka"),(6,"NOSQL"),(1,"stream"),(1,"MLlib")))

val result = myrdd1.cogroup(myrdd2)

result.collect

**Output :**  
Array[(Int, (Iterable[String], Iterable[String]))] =  
Array((4,(CompactBuffer(Flink),CompactBuffer(RealTime))),  
(1,(CompactBuffer(spark),CompactBuffer(stream, MLlib))),  
(6,(CompactBuffer(HBase),CompactBuffer(NOSQL))),  
(3,(CompactBuffer(Hive),CompactBuffer())),  
(5,(CompactBuffer(),CompactBuffer(Kafka))),  
(2,(CompactBuffer(HDFS),CompactBuffer())))

**Que 77. Explain pipe() operation in Apache Spark**

It 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 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.

**Que 78. Explain Spark coalesce() operation**

> It is a [**transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).  
> 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 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.

**Que 79.Explain the repartition() operation in Spark**

> repartition() is a transformation.  
> This function changes the number of partitions mentioned in parameter numPartitions(numPartitions : Int)  
> It's in package **org.apache.spark.rdd.ShuffledRDD**

**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.

**Que 80. Explain fullOuterJoin() operation in Apache Spark**

> It is transformation.  
> It's in package org.apache.spark.rdd.PairRDDFunctions

def fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))]

Perform a full outer join of this and other.  
For each element (k, v) in this, the resulting RDD will either contain all pairs (k, (Some(v), Some(w))) for w in other,  
or the pair (k, (Some(v), None)) if no elements in other have key k.  
Similarly, for each element (k, w) in other, the resulting RDD will either contain all pairs (k, (Some(v), Some(w))) for v in this,  
or the pair (k, (None, Some(w))) if no elements in this have key k.  
Hash-partitions the resulting RDD using the existing partitioner/ parallelism level.

**Example :**

val frdd1 = sc.parallelize(Seq(("Spark",35),("Hive",23),("Spark",45),("HBase",89)))

val frdd2 = sc.parallelize(Seq(("Spark",74),("Flume",12),("Hive",14),("Kafka",25)))

val fullouterjoinrdd = frdd1.fullOuterJoin(frdd2)

fullouterjoinrdd.collect

**Output :**  
Array[(String, (Option[Int], Option[Int]))] = Array((Spark,(Some(35),Some(74))), (Spark,(Some(45),Some(74))), (Kafka,(None,Some(25))), (Flume,(None,Some(12))), (Hive,(Some(23),Some(14))), (HBase,(Some(89),None)))

**Que 81. Expain Spark leftOuterJoin() and rightOuterJoin() operation**

> Both leftOuterJoin() and rightOuterJoin() are transformation.  
> Both in package org.apache.spark.rdd.PairRDDFunctions

**leftOuterJoin() :**

def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]

Perform a left outer join of this and other. For each element (k, v) in this, the resulting RDD will either contain all pairs (k, (v, Some(w))) for w in other, or the pair (k, (v, None)) if no elements in other have key k. Hash-partitions the output using the existing partitioner/parallelism level.

leftOuterJoin() performs a join between two RDDs where the keys must be present in first RDD

**Example :**

val rdd1 = sc.parallelize(Seq(("m",55),("m",56),("e",57),("e",58),("s",59),("s",54)))

val rdd2 = sc.parallelize(Seq(("m",60),("m",65),("s",61),("s",62),("h",63),("h",64)))

val leftjoinrdd = rdd1.leftOuterJoin(rdd2)

leftjoinrdd.collect

**Output :**  
Array[(String, (Int, Option[Int]))] = Array((s,(59,Some(61))), (s,(59,Some(62))), (s,(54,Some(61))), (s,(54,Some(62))), (e,(57,None)), (e,(58,None)), (m,(55,Some(60))), (m,(55,Some(65))), (m,(56,Some(60))), (m,(56,Some(65))))

**rightOuterJoin():**  
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]

Perform a right outer join of this and other. For each element (k, w) in other, the resulting RDD will either contain all pairs (k, (Some(v), w)) for v in this, or the pair (k, (None, w)) if no elements in this have key k. Hash-partitions the resulting RDD using the existing partitioner/parallelism level.

It performs the join between two RDDs where the key must be present in other RDD

**Example:**

val rdd1 = sc.parallelize(Seq(("m",55),("m",56),("e",57),("e",58),("s",59),("s",54)))

val rdd2 = sc.parallelize(Seq(("m",60),("m",65),("s",61),("s",62),("h",63),("h",64)))

val rightjoinrdd = rdd1.rightOuterJoin(rdd2)

rightjoinrdd.collect

**Que 82. Explain Spark join() operation**

> join() is transformation.  
> It's in package **org.apache.spark.rdd.pairRDDFunction**  
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]Permalink

Return an RDD containing all pairs of elements with matching keys in this and other.  
Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in this and (k, v2) is in other. Performs a hash join across the cluster.

**From** :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#213_Join>

It is joining two datasets. When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Example1:

val rdd1 = sc.parallelize(Seq(("m",55),("m",56),("e",57),("e",58),("s",59),("s",54)))

val rdd2 = sc.parallelize(Seq(("m",60),("m",65),("s",61),("s",62),("h",63),("h",64)))

val joinrdd = rdd1.join(rdd2)

joinrdd.collect

**Output:**

Array[(String, (Int, Int))] = Array((m,(55,60)), (m,(55,65)), (m,(56,60)), (m,(56,65)), (s,(59,61)), (s,(59,62)), (s,(54,61)), (s,(54,62)))

**Example2:**

val myrdd1 = sc.parallelize(Seq((1,2),(3,4),(3,6)))

val myrdd2 = sc.parallelize(Seq((3,9)))

val myjoinedrdd = myrdd1.join(myrdd2)

myjoinedrdd.collect

**Output:**  
Array[(Int, (Int, Int))] = Array((3,(4,9)), (3,(6,9)))

**Que 83. Explain the top() and takeOrdered() operation**

Both top() and takeOrdered() are actions.

Both returns then elements of [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/) based on default ordering or based on custom ordering provided by user.

def top(num: Int)(implicit ord: Ordering[T]): Array[T]

Returns the top k (largest) elements from this RDD as defined by the specified implicit Ordering[T] and maintains the ordering. This does the opposite of takeOrdered.

def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]

Returns the first k (smallest) elements from this RDD as defined by the specified implicit Ordering[T] and maintains the ordering. This does the opposite of top.

**Example :**

val myrdd1 = sc.parallelize(List(5,7,9,13,51,89))

myrdd1.top(3)

myrdd1.takeOrdered(3)

myrdd1.top(3)

**Output :**

Array[Int] = Array(89, 51, 13)

Array[Int] = Array(5, 7, 9)

Array[Int] = Array(89, 51, 13)

**Que 84. Explain first() operation in Spark**

> It is an action.  
> It returns the first element of the [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/).

**Example** :

val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt")

rdd1.count

rdd1.first

**Output :**  
Long: 20  
String : DataFlair is the leading technology training provider

**Que 85. Explain sum(), max(), min() operation in Apache Spark**

**sum() :**  
> It adds up the value in an [**RDD.**](http://data-flair.training/blogs/rdd-in-apache-spark/)  
> It is an package **org.apache.spark.rdd.DoubleRDDFunctions.**  
> Its return type is Double

**Example:**

val rdd1 = sc.parallelize(1 to 20)

rdd1.sum

**Output:**  
Double = 210.0

**max() :**  
> It returns a max value from RDD element defined by implicit ordering (element order)  
> It is an package org.apache.spark.rdd

**Example:**

val rdd1 = sc.parallelize(List(1,5,9,0,23,56,99,87))

rdd1.max

**Output:**  
Int = 99

**min() :**  
> It returns a **min** value from RDD element defined by implicit ordering (element order)  
> It is an package **org.apache.spark.rdd**

**Example:**

val rdd1 = sc.parallelize(List(1,5,9,0,23,56,99,87))

rdd1.min

**Output:**  
Int = 0

**Que 86. Explain countByValue() operation in Apache Spark RDD**

It is an action

It returns the count of each unique value in an RDD as a local Map (as a Map to driver program) **(value, countofvalues)**pair

Care must be taken to use this API since it returns the value to driver program so it's suitable only for small values.

**Example:**

val rdd1 = sc.parallelize(Seq(("HR",5),("RD",4),("ADMIN",5),("SALES",4),("SER",6),("MAN",8)))

rdd1.countByValue

**Output:**  
scala.collection.Map[(String, Int),Long] = Map((HR,5) -> 1, (RD,4) -> 1, (SALES,4) -> 1, (ADMIN,5) -> 1, (MAN,8) -> 1, (SER,6) -> 1)

val rdd2 = sc.parallelize{Seq(10,4,3,3)}

rdd2.countByValue

**Output:**  
scala.collection.Map[Int,Long] = Map(4 -> 1, 3 -> 2, 10 -> 1)

**Que 87. Explain the lookup() operation in Spark**

> It is an action  
> It returns the list of values in the RDD for key 'key'

val rdd1 = sc.parallelize(Seq(("Spark",78),("Hive",95),("spark",15),("HBase",25),("spark",39),("BigData",78),("spark",49)))  
rdd1.lookup("spark")  
rdd1.lookup("Hive")  
rdd1.lookup("BigData")

**Output:  
Seq[Int] = WrappedArray(15, 39, 49)  
Seq[Int] = WrappedArray(95)  
Seq[Int] = WrappedArray(78)**

**Que 88. Explain Spark countByKey() operation**

>It is an action operation  
> Returns (key, noofkeycount) pairs.

From :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#38_CountByKey>

It counts the value of **RDD** consisting of two components tuple for each distinct key. It actually counts the number of elements for each key and return the result to the master as lists of (key, count) pairs.

val rdd1 = sc.parallelize(Seq(("Spark",78),("Hive",95),("spark",15),("HBase",25),("spark",39),("BigData",78),("spark",49)))

rdd1.countByKey

**Output:**  
scala.collection.Map[String,Long] = Map(Hive -> 1, BigData -> 1, HBase -> 1, spark -> 3, Spark -> 1)

**Que 89. Explain Spark saveAsTextFile() operation**

From :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#310_SaveAsTextfile>

It writes the content of RDD to text file or saves the RDD as a text file in file path directory using string representation.

**Que 90. Explain reduceByKey() Spark operation**

> reduceByKey() is transformation which operate on pairRDD (which contains Key/Value).  
> PairRDD contains tuple, hence we need to pass the function that operator on tuple instead of each element.  
> It merges the values with the same key using associative reduce function.  
> It is wide operation because data shuffles may happen across multiple partitions.  
> It merges data locally before sending data across partitions for optimize data shuffling.  
> It takes function as an input which has two parameter of the same type (values associated with same key) and one element output of the input type(value)  
> We can say that it has three overloaded functions :  
 **reduceBykey(function)  
reduceByKey(function, numberofpartition)  
reduceBykey(partitioner, function)**  
From :  
<http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/#210_ReduceByKey>  
It uses associative reduce function, where it merges value of each key. It can be used with Rdd only in key value pair. It’s wide operation which shuffles data from multiple partitions/divisions and creates another RDD. It merges data locally using associative function for optimized data shuffling. Result of the combination (e.g. a sum) is of the same type that the values, and that the operation when combined from different partitions is also the same as the operation when combining values inside a partition.

Example :  
  
val rdd1 = sc.parallelize(Seq(5,10),(5,15),(4,8),(4,12),(5,20),(10,50)))  
val rdd2 = rdd1.reduceByKey((x,y)=>x+y)  
 **OR**  
rdd2.collect()

**Output:  
Array[(Int, Int)] = Array((4,20),(10,50),(5,45))**

**Que 91. Explain the operation reduce() in Spark**

> reduce() is an action. It is wide operation (i.e. shuffle data across multiple partitions and output a single value)  
> It takes function as an input which has two parameter of the same type and output a single value of the input type.  
> i.e. combine the elements of RDD together.

Example 1 :  
val rdd1 = sc.parallelize(1 to 100)  
val rdd2 = rdd1.reduce((x,y) => x+y)  
 **OR**  
val rdd2 = rdd1.reduce(\_ + \_)

**Output :  
rdd2: Int = 5050**

Example 2:  
val rdd1 = sc.parallelize(1 to 5)  
val rdd2 = rdd1.reduce(\_\*\_)

**Output :  
rdd2: Int = 120**

**Que 92.Explain the action count() in Spark RDD**

count() is an action in [**Apache Spark RDD operation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/)

count() returns the number of elements in [**RDD.**](http://data-flair.training/blogs/rdd-in-apache-spark/)

**Example:**  
val rdd1 = sc.parallelize(List(10,20,30,40))  
println(rdd1.count())  
 **Output:  
4**  
It returns a number of elements or items in RDD. So it basically counts the number of items present in the dataset and returns a number after the count.

**Que 93. Explain Spark map() transformation**

> map() [**transformation**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) takes a function as input and apply that function to each element in the [**RDD**](http://data-flair.training/blogs/rdd-in-apache-spark/).  
> Output of the function will be a new element (value) for each input element.  
Ex.  
val rdd1 = sc.parallelize(List(10,20,30,40))  
val rdd2 = rdd1.map(x=>x\*x)  
println(rdd2.collect().mkString(","))

**Que 94. Explain the flatMap() transformation in Apache Spark**

When one want to produce multiple elements (values) for each input element, flatMap() is used.

As with map(), flatMap() also takes function as an input.

Output of the function is a List of the element through which we can iterate. (i.e. function can return 0 or more element for each input element)

Simple use of flatMap() is splittin up an input line (string) into words.

**Example**

val fm1 = sc.parallelize(List("Good Morning", "Data Flair", "Spark Batch"))

val fm2 = fm1.flatMap(y => y.split(" "))

fm2.foreach{println}

**Output is as follows:**

Good  
Morning  
Data  
Flair  
Spark  
Batch

**Que 95. What are the limitations of Apache Spark?**

**Limitations of Apache Spark:**

**1. No File Management System**  
Apache Spark relies on other platforms like Hadoop or some another cloud-based Platform for file management system. This is one of the major issues with Apache Spark.

**2. Latency**  
While working with Apache Spark, it has higher latency.

**3. No support for Real-Time Processing**  
In Spark Streaming, the arriving live stream of data is divided into batches of the pre-defined interval, and each batch of data is treated like Spark [**Resilient Distributed Database**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) (RDDs). Then these RDDs are processed using the operations like map, reduce, join etc. The result of these operations is returned in batches. Thus, it is not real-time processing but Spark is near real-time processing of live data. Micro-batch processing takes place in [**Spark Streaming**](https://data-flair.training/blogs/apache-spark-streaming-tutorial/).

**4. Manual Optimization**  
Manual Optimization is required to optimize Spark jobs. Also, it is adequate to specific datasets. we need to control manually if we want to partition and cache in Spark to be correct.

**5. Less no. of Algorithm**  
Spark MLlib lags behind in terms of a number of available algorithms like Tanimoto distance.

**6. Window Criteria**  
Spark does not support record based window criteria. It only has time-based window criteria.

**7. Iterative Processing**  
In Spark, the data iterates in batches and each iteration is scheduled and executed separately.

**8. Expensive**  
when we want cost-efficient processing of big data [**In-memory**](https://data-flair.training/blogs/apache-spark-in-memory-computing/) capability can become a bottleneck as keeping data in memory is quite expensive. At that time the memory consumption is very high, and it is not handled in a user-friendly manner. The cost of Spark is quite high because Apache Spark requires lots of RAM to run in-memory.

**Que 96. What is Spark SQL?**

Spark SQL is a Spark interface to work with Structured and Semi-Structured data (data that as defined fields i.e. tables). It provides abstraction layer called [**DataFrame**](https://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/) and [**DataSet**](https://data-flair.training/blogs/apache-spark-dataset-tutorial/) through with we can work with data easily. One can say that DataFrame is like a table in a relational database. Spark SQL can read and write data in a variety of Structured and Semi-Structured formats like Parquets, JSON, Hive. Using SparkSQL inside Spark application is the best way to use it. This empowers us to load data and query it with SQL. we can also combine it with “regular” program code in Python, Java or [**Scala**](https://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/).

**Que 97. Explain Spark SQL caching and uncaching**

[**FIND**](https://data-flair.training/forums/topic/spark-sql-caching-and-uncaching) **ANSWER**

**Que 98. Explain Spark streaming**

**Spark Streaming**  
A data stream defines as a data arriving continuously in the form of an unbounded sequence. For further processing, Streaming separates continuously flowing input data into discrete units. It is a low latency processing and analyzing of streaming data.

In the year 2013, Apache Spark Streaming was added to [**Apache Spark**](https://data-flair.training/blogs/apache-spark-for-beginners/). Through Streaming, we can do [**fault-tolerant**](https://data-flair.training/blogs/fault-tolerance-in-apache-spark/),scalable stream processing of live data streams. From many sources like Kafka, Apache Flume, Amazon Kinesis or TCP sockets, Data ingestion can be possible. Also, by using complex algorithms, processing is possible. That are expressed with high-level functions such as map, reduce, join and window. By the end, processed data can be pushed out to filesystems, databases and live dashboards.

Internally, By Spark streaming, Live input data streams are received and divided into batches. Afterwards, these batches are then processed by the Spark engine to generate the final stream of results in batches.

Discretized Stream or, in short, a Spark [**DStream**](https://data-flair.training/blogs/apache-spark-dstream-discretized-streams/) is its basic abstraction. That also represents a stream of data divided into small batches. DStreams are built on Spark RDDs, Spark’s core data abstraction. Streaming can aslo integrate with any other [**Apache Spark components**](https://data-flair.training/blogs/apache-spark-ecosystem-components/) like [**Spark MLlib**](https://data-flair.training/blogs/apache-spark-ecosystem-components/) and [**Spark SQL**](https://data-flair.training/blogs/spark-sql-tutorial/).

**Que 99. What is DStream in Apache Spark Streaming?**

A **Discretized Stream (DStream)**, the basic abstraction in [**Spark Streaming**](https://data-flair.training/blogs/apache-spark-streaming-tutorial/), is a continuous sequence of [**RDDs**](https://data-flair.training/blogs/apache-spark-rdd-tutorial/) representing a continuous stream of data. DStreams can either be created from live data (such as, data from [**HDFS**](https://data-flair.training/blogs/hadoop-hdfs-tutorial/), Kafka or Flume) or it can be generated by [**transformation**](https://data-flair.training/blogs/apache-spark-streaming-transformation-operations/) existing DStreams using operations such as map, window and reduceByKeyAndWindow.

Internally, there are few basic properties by which DStreams is characterized:

1. DStream depends on the list of other DStreams.  
2. A time interval at which the DStream generates an RDD  
3. A function that is used to generate an RDD after each time interval

**Que 100. Explain different transformations in DStream in Apache Spark Streaming**

Different transformations in [**DStream**](http://data-flair.training/blogs/apache-spark-dstream-discretized-streams/) in [**Apache Spark Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/) are:

1-**map(func)** -- Return a new DStream by passing each element of the source DStream through a function func.

2-**flatMap(func)** -- Similar to map, but each input item can be mapped to 0 or more output items.

3-**filter(func)** -- Return a new DStream by selecting only the records of the source DStream on which func returns true.

4-**repartition(numPartitions)** -- Changes the level of parallelism in this DStream by creating more or fewer partitions.

5-**union(otherStream)** -- Return a new DStream that contains the union of the elements in the source DStream and  
otherDStream.

6-**count()** -- Return a new DStream of single-element [**RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/) by counting the number of elements in each RDD of the source DStream.

7-**reduce(func)**-- Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one).

8-**countByValue()** -- When called on a DStream of elements of type K, Return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.

9-**reduceByKey(func, [numTasks])**-- When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

10-**join(otherStream, [numTasks])** -- When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

11-**cogroup(otherStream, [numTasks])** -- When called on DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

12-**transform(func)** -- Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream.

13-**updateStateByKey(func)** -- Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key.

**Que 101. What is Starvation scenario in spark streaming**

In the famous word count example for spark streaming, the spark configuration object is initialized as follows:

/\* Create a local StreamingContext

with two working thread and batch

interval of 1 second.

The master requires 2 cores

to prevent from a starvation scenario. \*/

val sparkConf = new SparkConf().

setMaster("local[2]").

setAppName("WordCount")

Here if I change the master from local[2] to local or does not set the Master, I do not get the expected output and in fact word counting doesn't happen at all.

**The comment says "The master requires 2 cores to prevent from a starvation scenario" that's why they have done setMaster("local[2]").**

**Can somebody explain me why it requires 2 cores and what is starvation scenario ?**

**Que 102.Explain the level of parallelism in spark streaming**

> In order to reduce the processing time, one need to increase the parallelism.  
> In Spark Streaming, there are three ways to increase the parallelism :  
**(1) Increase the number of receivers :**If there are too many records for single receiver (single machine) to read in and distribute so that is bottleneck. So we can increase the no. of receiver depends on scenario.  
**(2) Re-partition the receive data :**If one is not in a position to increase the no. of receivers in that case redistribute the data by re-partitioning.  
**(3) Increase parallelism in aggregation :**

**Que 103. What are the different input sources for Spark Streaming**

> TCP Sockets  
> Stream of files  
> ActorStream  
> Apache Kafka  
> Apache Flume (Push-based receiver or Pull-based receiver)  
> Kinesis  
> Above are the various input source (receiver) through which data stream can make their way to the streaming application.

**Que 104. Explain Spark Streaming with Socket**

I'm trying to get the example spark streaming example to run but instead of getting input from the console created with netcat: nc -lk 9999, I want to send messages to this console and have spark listen. How can I do this.

So the idea is: Spark Listens from nc -l 9999 (as example) and we want send a message from a socket. How can we do this? Is there a way of avoiding to have a server here and can we make do with netcat?

Other system -> Socket message -> nc -> spark

**Que 105.  Define the roles of the file system in any framework?**

In order to manage data on computer, one has to interact with the File System directly or indirectly.

When we install Hadoop on our computer, actually there are two file system exists on machine  
*(1) Local File System*,  *(2) HDFS (Hadoop Distributed File System)*

HDFS is sits top on of Local File System.

Following are the genera functions of File System (be it Local or HDFS)

Control the data access mechanism (i.e how data stored and retrived)

Manages the metadata about the Files / Folders (i.e. created date, size etc)

Grants the access permission and manage the securities

Efficiently manage the storage space

**Que 106. How do you parse data in XML? Which kind of class do you use with Java to parse data?**

--> One way to parse the XML data in Java is to use the JDOM library. One can download it and import the JDOM library in your project. You can get the help from Google. If still, required help post your problem in the forum. I will try to give you the solution.

--> For Scala, Scala has inbuilt library for xml parsing. Scala-xml\_2.11-1.0.2 jar (please check the for new version if available).

**Que 107. What is PageRank in Spark?**

**FIND THE ANSWER**

**Que 108. 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

**Que 109. 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.

**Que 110. On what all basis can you differentiate RDD, DataFrame, and DataSet?**

**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. It is a data abstraction and domain-specific language (DSL) applicable on structure and semi-structured data. It is distributed 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 two-dimensional array like structure where each column contains the value of one variable and row contains one set of values for each column. It 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 set of records, immutable collection of objects with distributed computing. RDD is large collection of data or RDD is an array of reference of partitioned objects. Each and every datasets in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. self-recovered/recomputed in the case of failure. The dataset could be data loaded 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.