BIG DATA ANALYTICS

UNIT-IV

SPARK BASICS: History of Spark, Spark Ecosystem, Spark Architecture, Spark Shell, Working with RDDs in Spark: RDD Basics, Creating RDDs in Spark, RDD Operations, Passing Functions to Spark, Transformations and Actions in Spark, Spark RDD Persistence

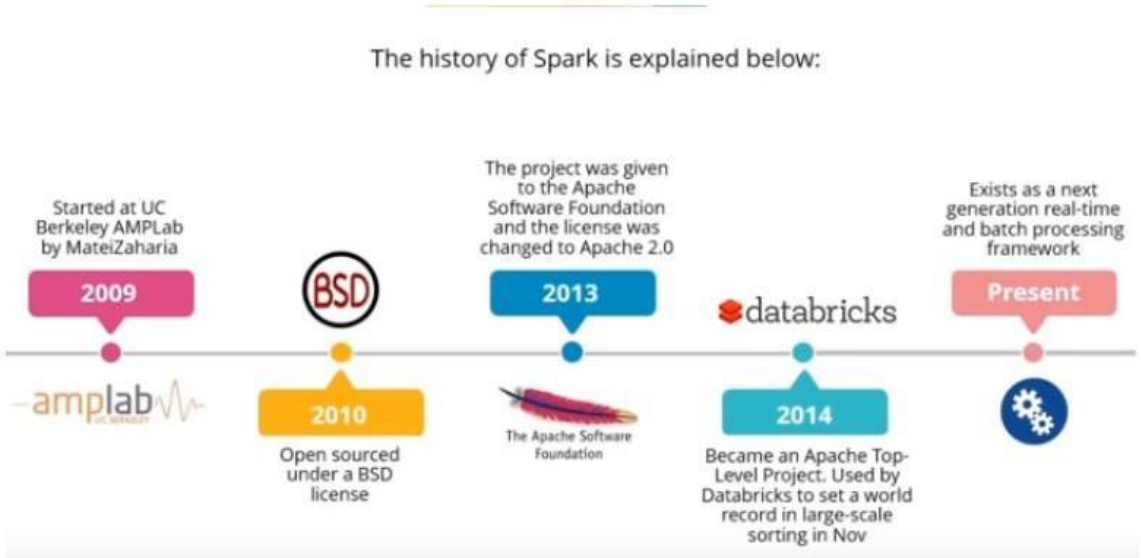
Working with Key/Value Pairs: Pair RDDs, Transformations on Pair RDDs, Actions Available on Pair RDDs

INTRODUCTION TO APACHE SPARK:

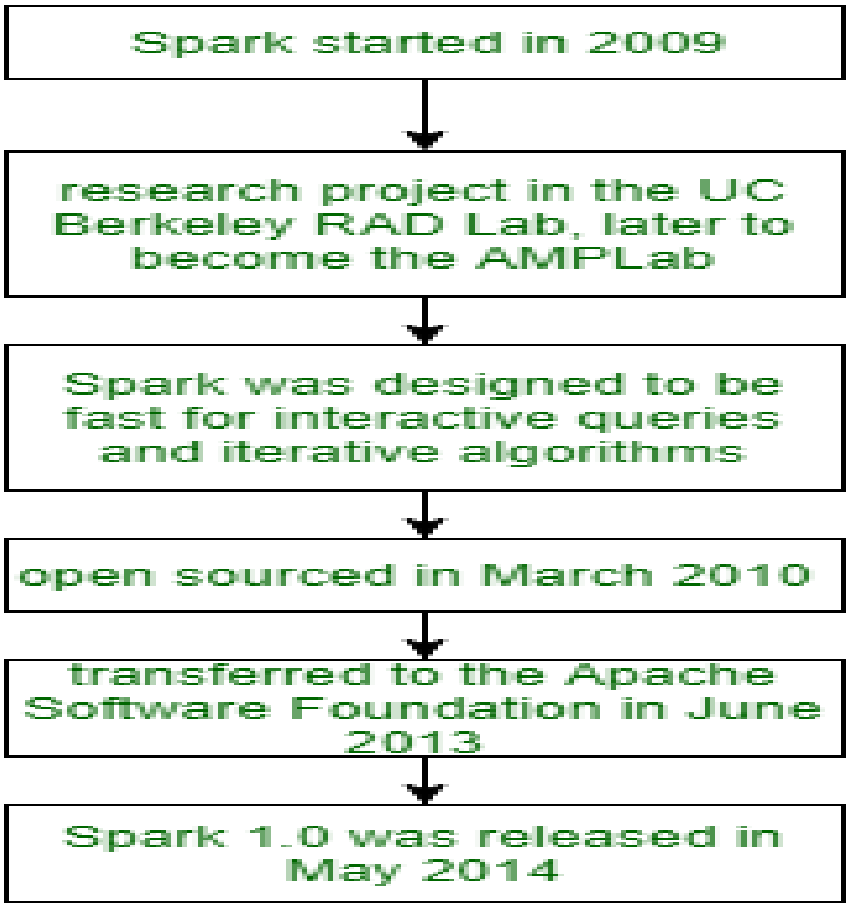
- Apache Spark is an open-source **in-memory cluster computing** framework, used for Storing & processing the Real time data .
- Its **primary purpose** is to handle the **real-time generated data**.
- Spark copies most of the data from a physical server on to a RAM . This is called the **"in memory "operation**. Due to its **in-memory cluster computing Feature** it increases the processing speed of an application.

- Spark applies Hadoop in two forms. The first form is STORAGE and another one is PROCESSING. Thus, spark includes its computation for cluster management and applies **Hadoop** for only storage purposes.
- Spark was built on the top of the Hadoop MapReduce.
- It was optimized to run in memory whereas alternative approaches like Hadoop's MapReduce writes data to and from computer hard drives. So, Spark processes the data much quicker than other alternatives.
- Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.
- Spark is simple due to it could be used for more than one thing such as working with data streams or graphs, machine learning algorithms, inhaling data into the database, building data pipelines, executing distributed sql, and others.

History of Spark:



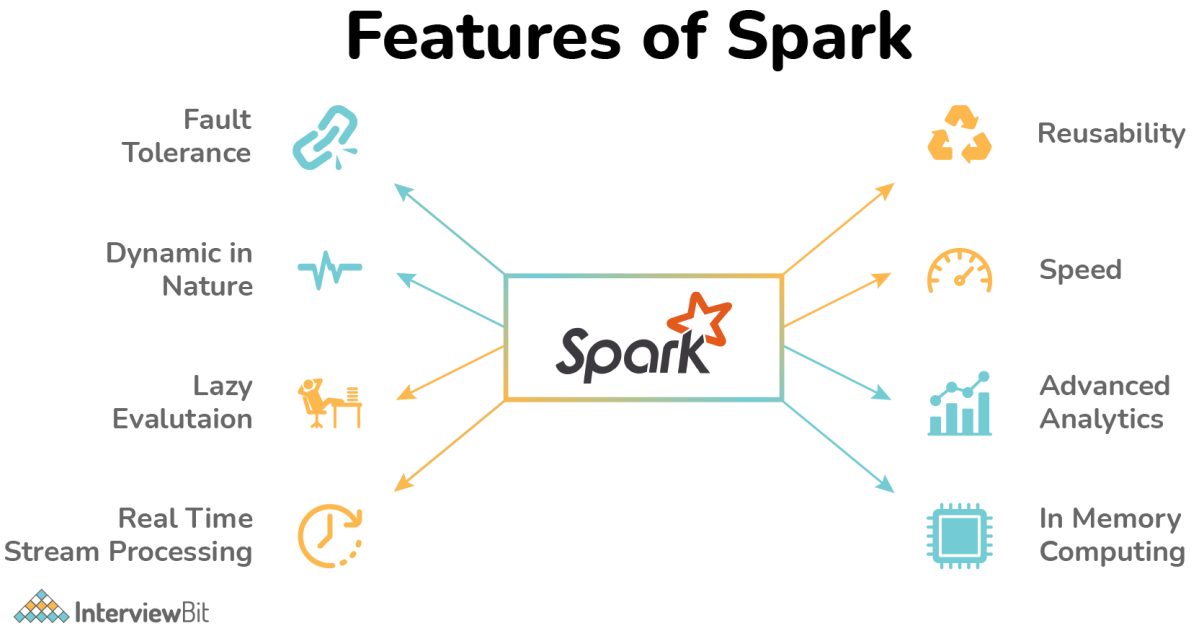
- Spark started in **2009** as a research project in the UC(University of California) Berkeley RAD Lab, later to become the **AMPLab.**
- The researchers in the lab had previously been working on Hadoop MapReduce, and observed that MapReduce was inefficient for iterative and interactive computing jobs. Thus, from the beginning, Spark was designed to be fast for interactive queries and iterative algorithms, bringing in ideas like support for in-memory storage and efficient fault recovery.
- Soon after its creation it was already 10–20× faster than MapReduce for certain jobs.
- Some of Spark's first users were other groups inside UC Berkeley, including machine learning researchers such as the Mobile Millennium project, which used Spark to monitor and predict traffic congestion in the San Francisco Bay Area.
- Spark was first **open sourced in March 2010** under a BSD license.
- In **2013,** the project was acquired by **Apache Software Foundation.**
- In **2014,** the Spark emerged as a Top-Level Apache Project.
- In addition to UC Berkeley, major contributors to Spark include **Databricks**, Yahoo!, and Intel.
- In **2014 it became the Apache Top level Project** and its used by **Databricks** to set a world record in Large scale Sorting.
- Internet powerhouses such as Netflix, Yahoo, and eBay have deployed Spark at massive scale, collectively processing multiple petabytes of data on clusters of over 8,000 nodes. It has quickly become the largest open source community in big data, with over 1000 contributors from 250+ organizations.



Why Spark?

- Spark is used to cache data in memory across multiple parallel operations.
- it is much faster than MapReduce, which requires more disc reading and writing. Spark has a faster startup time, higher parallelism, and better CPU utilization today.
- Spark uses a more robust functional programming method than MapReduce.
- Spark is particularly well-suited to iterative algorithms that process large amounts of data in parallel.

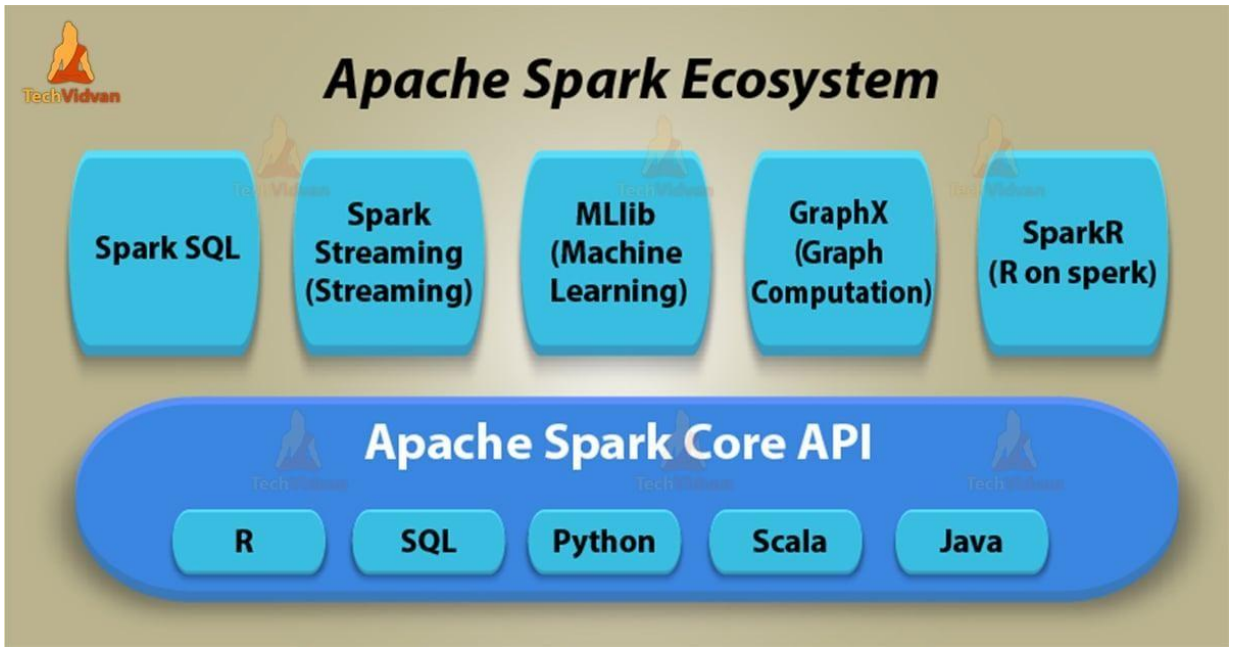
FEATURES OF SPARK:



- **High Processing Speed**: Apache Spark helps in the achievement of a very high processing speed of data by reducing read-write operations to disk. The speed is almost **100x faster** while performing **in-memory computation** and **10x** faster while performing disk computation.
- **Dynamic Nature**: Spark provides 80 high-level operators which help in the easy development of parallel applications.
- **In-Memory Computation**: The in-memory computation feature of Spark due to its DAG execution engine increases the speed of data processing. This also supports data caching and reduces the time required to fetch data from the disk.
- **Reusability**: Spark codes can be reused for batch-processing, data streaming, running ad-hoc queries, etc.
- **Fault Tolerance**: Spark supports fault tolerance using RDD. Spark RDDs are the abstractions designed to handle failures of worker nodes which ensures zero data loss.

- **Stream Processing**: Spark supports stream processing in real-time. The problem in the earlier MapReduce framework was that it could process only already existing data.
- **Lazy Evaluation**: Spark transformations done using Spark RDDs are lazy. Meaning, they do not generate results right away, but they create new RDDs from existing RDD. This lazy evaluation increases the system efficiency.
- **Support Multiple Languages**: Spark supports multiple languages like R, Scala, Python, Java which provides dynamicity and helps in overcoming the Hadoop limitation of application development only using Java.
- **Cost Efficiency**: Apache Spark is considered a better cost-efficient solution when compared to Hadoop as Hadoop required large storage and data centers while data processing and replication.
- **Real-Time:**
  It offers Real-time computation & low latency because of **in-memory computation.**

Spark Ecosystem/Components:



- Apache Spark ecosystem is built on top of the core execution engine that has extensible API's in different languages.

1) Scala

Spark framework is built on Scala, so programming in Scala for Spark can provide access to some of the latest and greatest features that might not be available in other supported programming spark languages.

2) Python

Python language has excellent libraries for data analysis like Pandas and Sci-Kit learn but is comparatively slower than Scala.

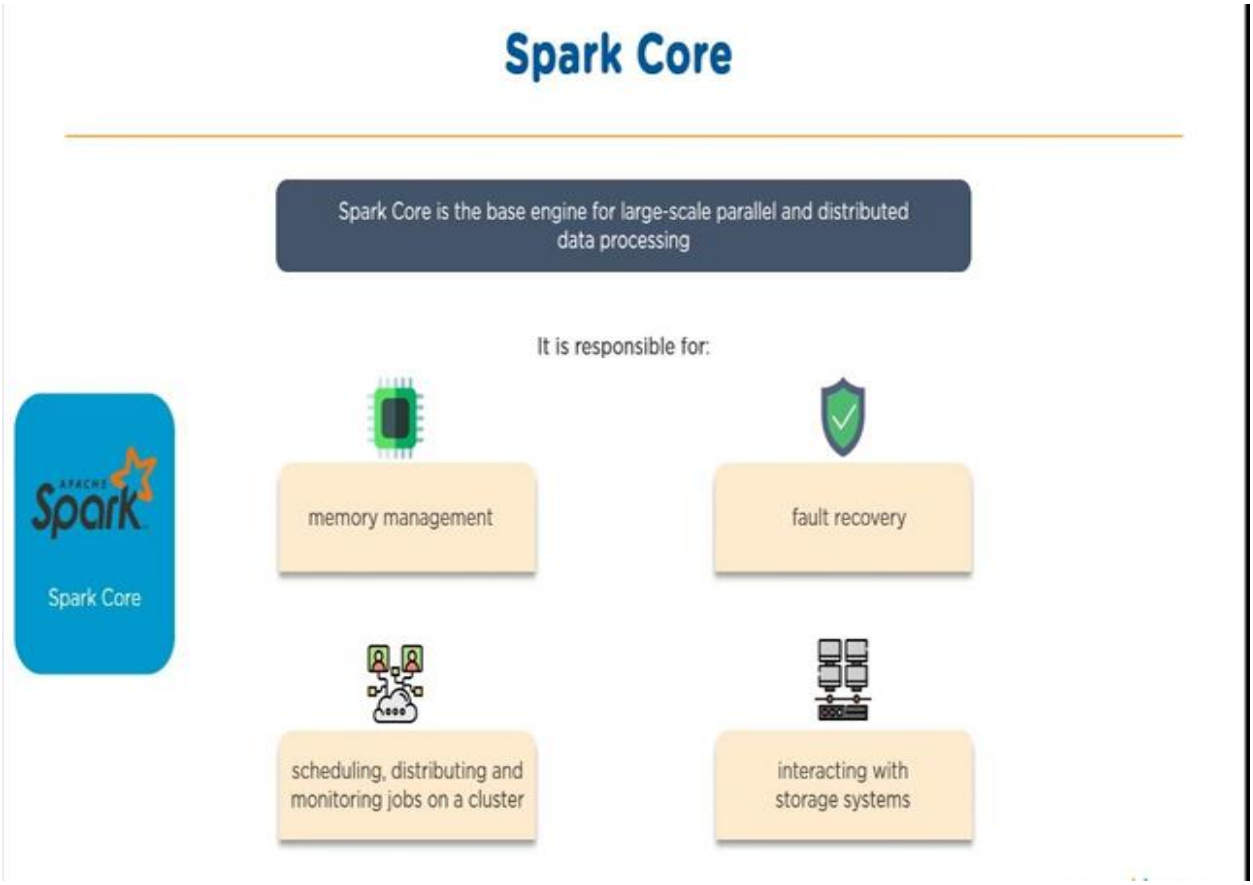
3) R Language

R programming language has rich environment for machine learning and statistical analysis which helps increase developer productivity. Data scientists can now use R language along with Spark through SparkR for processing data that cannot be handled by a single machine.

Spark Components

The Apache Spark components include:

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib(Machine learning library)
- GraphX
- Spark R

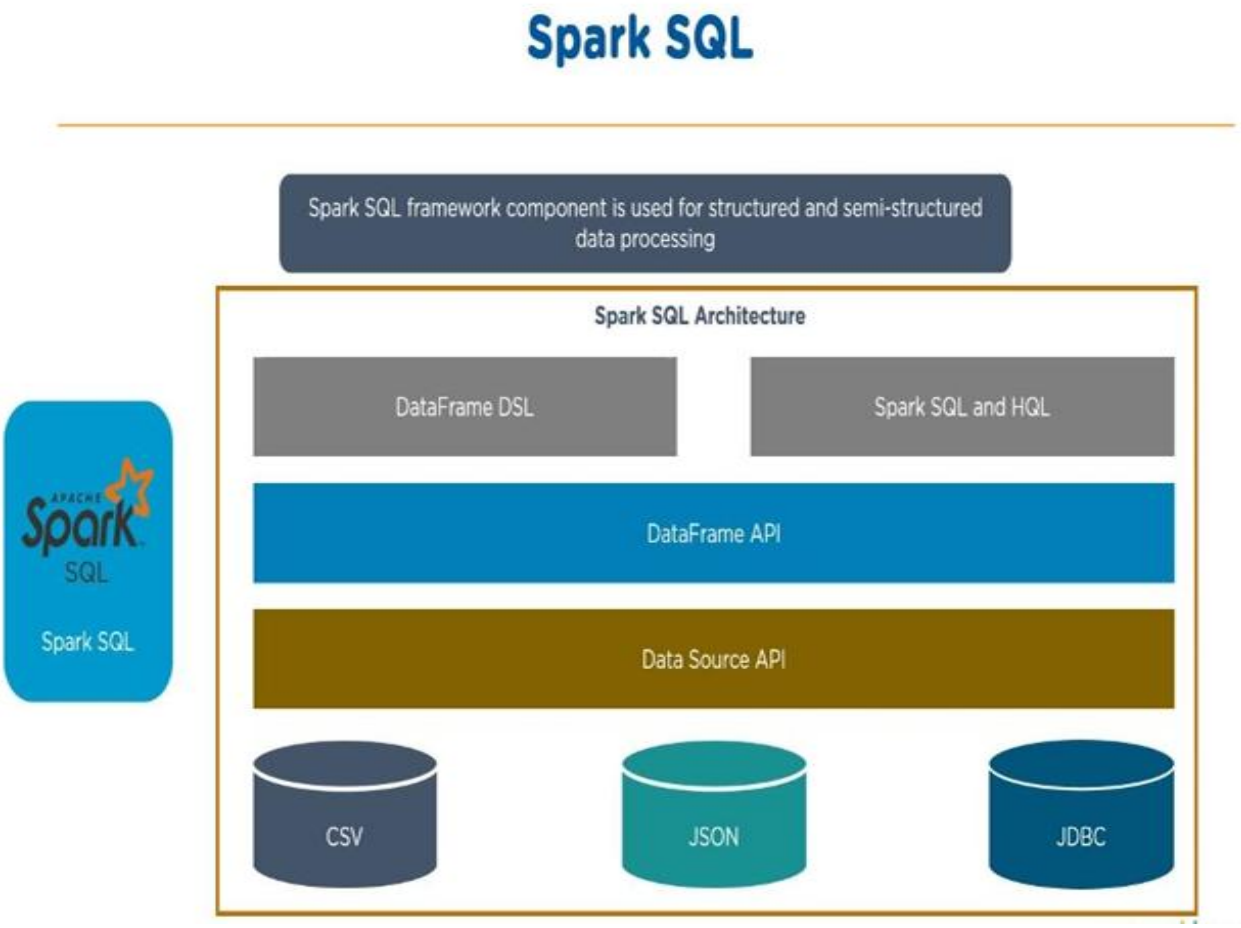


Spark core:

- Spark Core component is accountable for all the basic I/O Functionalities. It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster & interacting with storage systems.

Spark Core makes use of a special data structure known as RDD (Resilient Distributed Datasets). Data sharing or reuse in distributed computing systems like Hadoop MapReduce requires the data to be stored in intermediate stores like HDFS. This slows down the overall computation speed because of several replications, IO operations and serializations in storing the data in these intermediate stable data stores. Resilient Distributed Datasets overcome this drawback of Hadoop MapReduce by allowing - fault tolerant 'in-memory' computations.

Spark SQL:



- On the top of spark, spark SQL enables users to run SQL/HQL queries.
- Spark developers can leverage the power of declarative queries and optimized storage by running SQL like queries on Spark data, that is present in RDDs and other external sources.
- Users can perform, extract, transform and load functions on the data coming from various formats like JSON or Parquet or Hive and then run ad-hoc queries using Spark SQL.
- Spark SQL eases the process of extracting and merging various datasets so that the datasets are ready to use for machine learning.
- We can process structured as well as semi-structured data, by using spark SQL.
- Moreover, it offers to run unmodified queries up to 100 times faster on existing deployments.

Spark streaming:

- **Spark Streaming** is a light weight API that allows developers to perform batch processing and streaming of data with ease, in the same application.
- It makes use of a continuous stream of input data (Discretized Stream or Stream- a series of RDD's) to process data in real-time.
- Spark Streaming leverages the fast scheduling capacity of Apache Spark Core to perform streaming analytics by ingesting data in mini-batches.
- Transformations are applied on those mini batches of data.
- Data in Spark Streaming is ingested from various data sources and live streams like Twitter, IoT Sensors, Apache Flume, etc.
- Spark streaming is used in applications that require real-time statistics and rapid response like alarms, IoT sensors, diagnostics, cyber security, etc.
- Spark streaming finds great applications in Log processing, Intrusion Detection and Fraud Detection.

Spark MLlib:

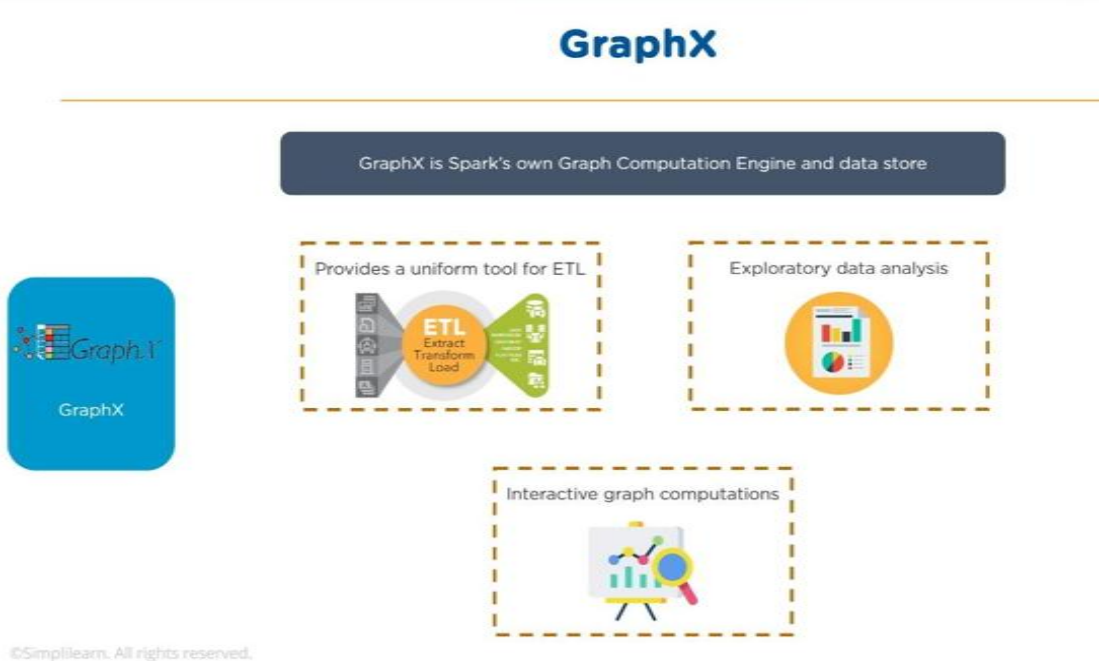
- Machine learning library delivers both efficiencies as well as the high-quality algorithms.
- MLlib is simple to use, scalable, compatible with various programming languages and can be easily integrated with other tools.
- MLlib eases the deployment and development of scalable machine learning pipelines.
- MLlib library has implementations for various common machine learning algorithms –

Clustering- K-means
Classification – naïve Bayes, logistic regression, SVM
Regression –Linear Regression

- Moreover, it is the best choice for a data scientist. Since it is capable of in-memory data processing, that improves the performance of iterative algorithm drastically.

Spark graphx:

- GraphX is the newest component in Spark.
- It is the graph computation engine built on top of apache spark that enables to process graph data at scale.
- It's a directed multigraph, which means it contains both edges and vertices and can be used to represent a wide range of data structures.
- It includes a growing collection of algorithms that help you analyze your data



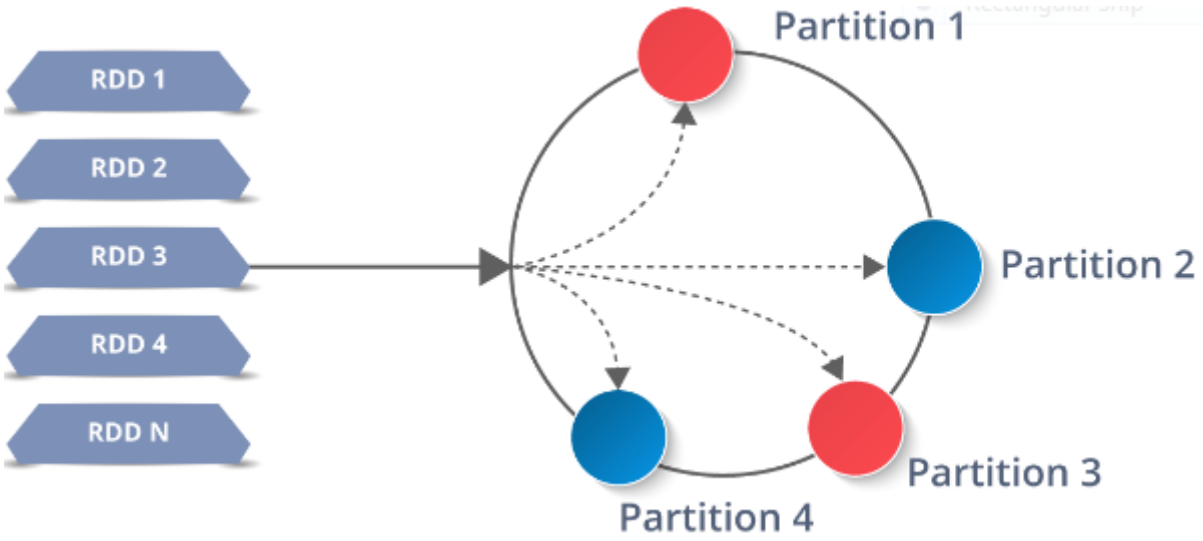
SparkR:

- SparkR is an R package that provides a light-weight frontend to use Apache Spark from R.
- SparkR exposes the Spark API through the RDD class and allows users to interactively run jobs from the R shell on a cluster.
- Moreover, it allows data scientists to analyze large datasets.
- Although, the main idea behind sparkr was to explore different techniques to integrate the usability of R with the scalability of spark.

Resilient Distributed Dataset(RDD)

RDDs are the building blocks of any Spark application. RDDs Stands for:

Resilient: Fault tolerant and is capable of rebuilding data on failure
Distributed: Distributed data among the multiple nodes in a cluster
Dataset: Collection of partitioned data with values



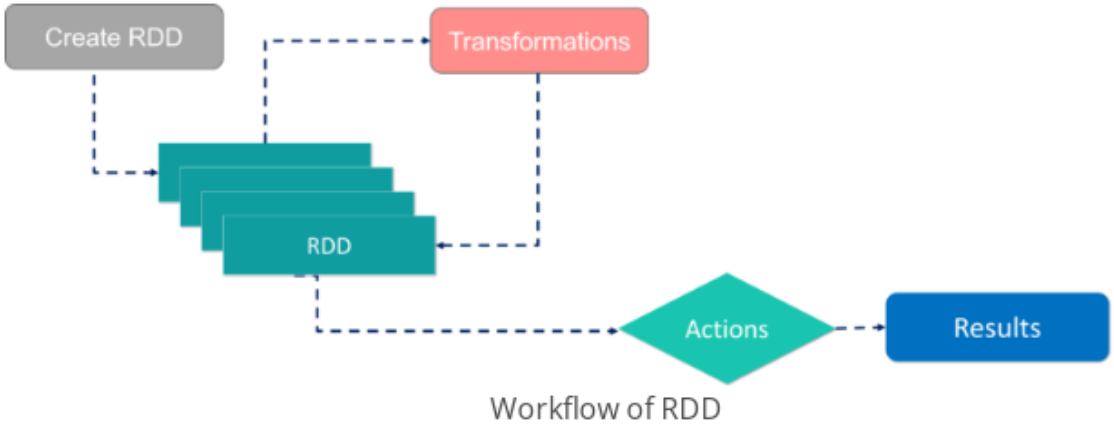
It is a layer of abstracted data over the distributed collection. It is immutable in nature and follows lazy transformations.

Spark Architecture:

- The Apache Spark framework uses a master-slave architecture.
- Its cluster consists of a single master and multiple slaves in which consists of a **driver,** which runs as a **master node,** and many **executors** that run across as **worker nodes(Slave nodes)** in the cluster.
- Apache Spark can be used for batch processing and real-time processing as well.
- The spark architecture depends upon two abstractions:

Resilient Distributed Dataset(RDD)
Directed Acyclic Graph (DAG)

- **Resilient distributed datasets (RDD):** **RDD** is an immutable (read-only), fundamental collection of elements or items that can be operated on many devices at the same time (spark parallel processing). once you create an RDD it becomes immutable. By immutable I mean, an object whose state cannot be modified after it is created, but they can surely be transformed. Each dataset in an RDD can be divided into logical portions, which are then executed on different nodes of a cluster.
- The resilient distributed datasets are the group of data items that can be stored in-memory on worker nodes. Here,
  - Resilient: restore the data on failure.
  - Distributed: data is distributed among different nodes.
  - Dataset: group of data



Workflow of RDD

There are two ways to create RDDs − parallelizing an existing collection in your driver program, or by referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, etc.

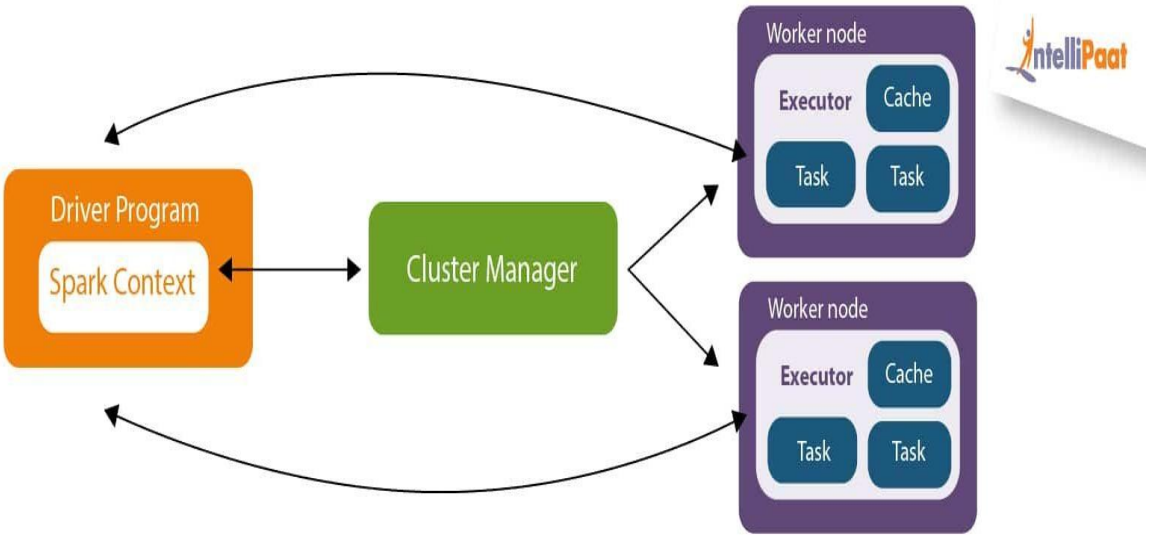
With RDDs, you can perform two types of operations:

Transformations: They are the operations that are applied to create a new RDD.
Actions: They are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.

Directed acyclic graph (DAG):

- DAG is a finite direct graph that performs a sequence of computations on data.
- (Directed Acyclic Graph) DAG in Apache Spark is a set of Vertices and Edges, where ***vertices*** represent the RDDs and the ***edges*** represent the Operation to be applied on RDD.
- In Spark DAG, every edge directs from earlier to later in the sequence.
- On the calling of ***Action,*** the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task.
- The Scheduler splits the Spark RDD into **stages** based on various transformation applied.
- Each stage is comprised of **tasks**, based on the partitions of the RDD, which will perform same computation in parallel.
- The graph here refers to navigation, and directed and acyclic refers to how it is done.

Working on the Apache Spark Architecture:



- Driver Program in the Apache Spark architecture calls the main program of an application and creates SparkContext.
- A SparkContext consists of all the basic functionalities. Spark Driver contains various other components such as DAG Scheduler, Task Scheduler, Backend Scheduler, and Block Manager, which are responsible for translating the user-written code into jobs that are actually executed on the cluster.
- Spark Driver and SparkContext collectively watch over the job execution within the cluster.

- Spark Driver works with the Cluster Manager to manage various other jobs.
- The cluster Manager does the resource allocating work. And then, the job is split into multiple smaller tasks which are further distributed to worker nodes.
- Whenever an RDD is created in the SparkContext, it can be distributed across many worker nodes and can also be cached there.
- Worker nodes execute the tasks assigned by the Cluster Manager and return it back to the Spark Context.
- An executor is responsible for the execution of these tasks.
- The lifetime of executors is the same as that of the Spark Application. If we want to increase the performance of the system, we can increase the number of workers so that the jobs can be divided into more logical portions.

Driver Program:

- The Driver Program is a process that runs the main() function of the application and creates the **SparkContext** object. The purpose of **SparkContext** is to coordinate the spark applications, running as independent sets of processes on a cluster.

- To run on a cluster, the **SparkContext** connects to a different type of cluster managers and then perform the following tasks: -
    - It acquires executors on nodes in the cluster.
    - Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
    - At last, the SparkContext sends tasks to the executors to run.

Cluster Manager:

- The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.
- It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
- Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node:

- The worker node is a slave node
- Its role is to run the application code in the cluster.

Executor:

- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and write data to the external sources.
- Every application contains its executor.

Task:

- A unit of work that will be sent to one executor.

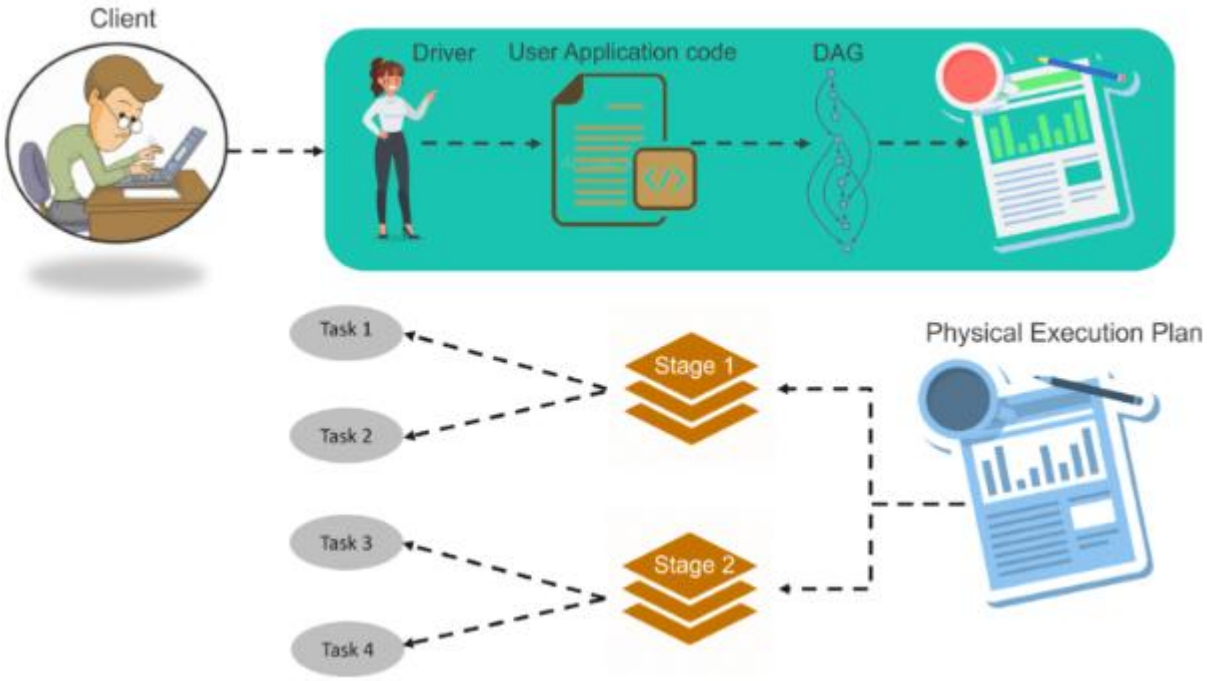


Fig: Spark Architecture Infographic

STEP 1:

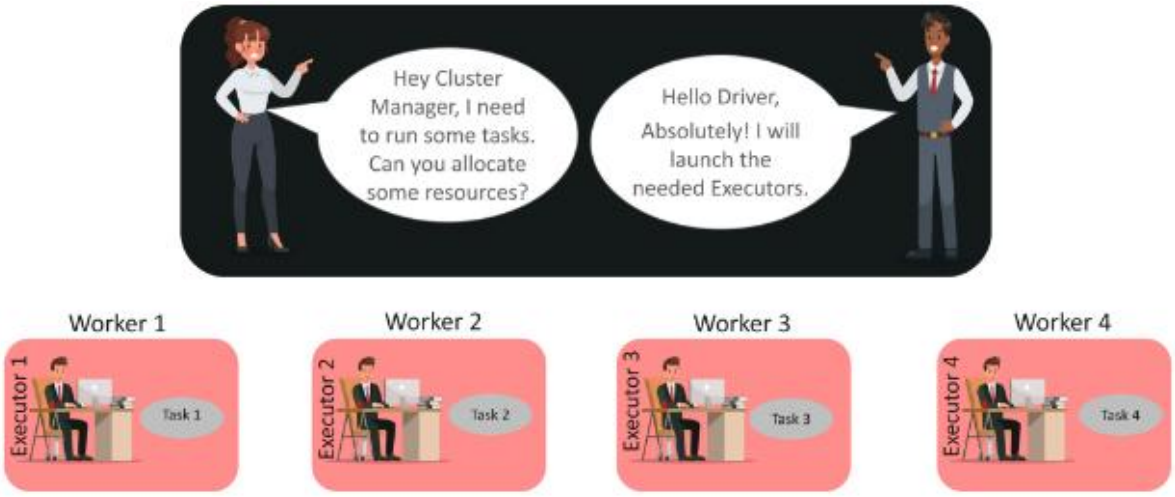
- The client submits spark user application code.
- When an application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically directed acyclic graph called DAG.
- At this stage, it also performs optimizations such as pipelining transformations.

STEP 2:

- After that, it converts the logical graph called DAG into physical execution plan with many stages.
- After converting into a physical execution plan, it creates physical execution units called tasks under each stage.
- Then the tasks are bundled and sent to the cluster.

STEP 3:

- Now the driver talks to the cluster manager and negotiates the resources.
- Cluster manager launches executors in worker nodes on behalf of the driver.
- At this point, the driver will send the tasks to the executors based on data placement.
- When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task.

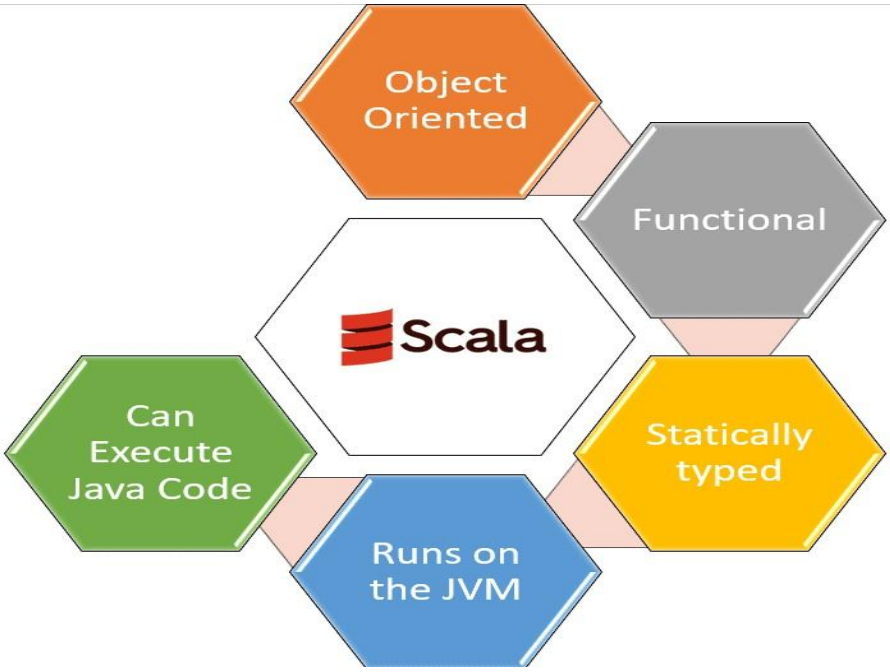


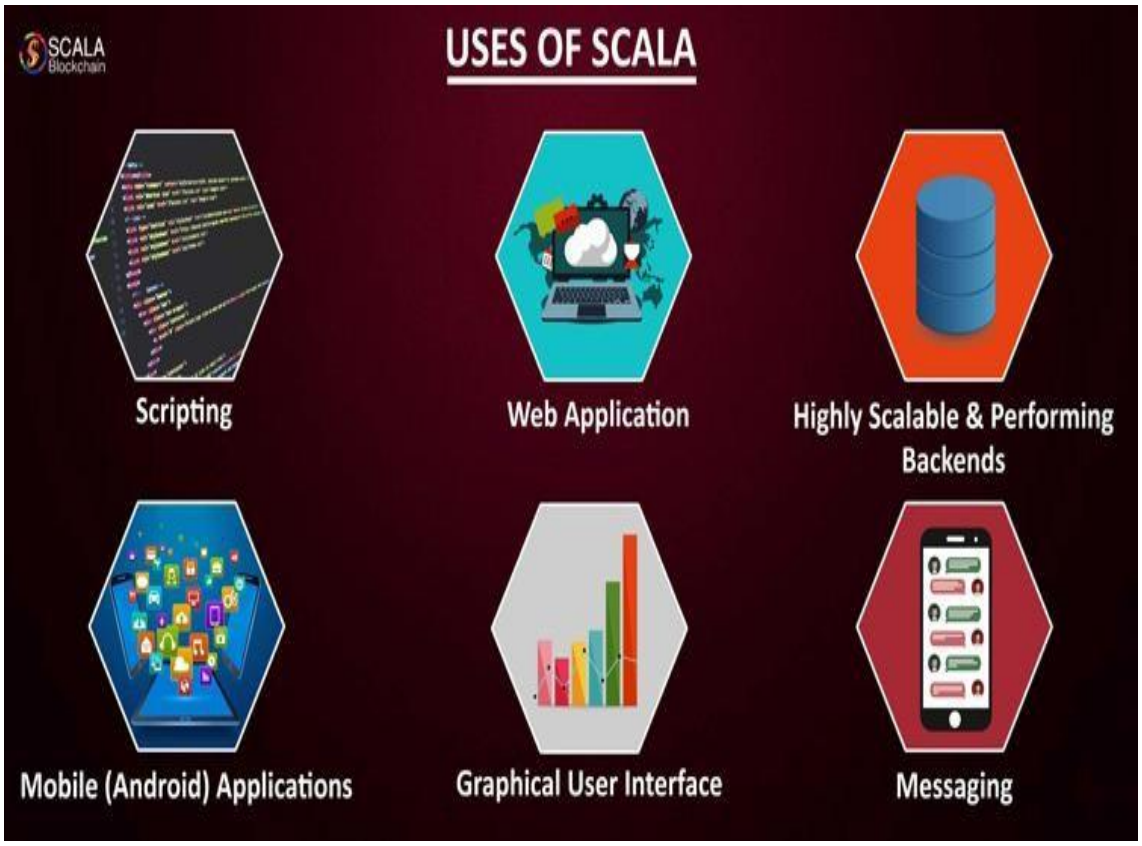
STEP 4:

- During the course of execution of tasks, driver program will monitor the set of executors that runs.
- Driver node also schedules future tasks based on data placement.

SCALA:

- **Scala** is a general-purpose, high-level, multi-paradigm programming language.
- It is a pure object-oriented programming language which also provides the support to the functional programming approach.
- There is no concept of primitive data as everything is an object in Scala.
- It is designed to express the general programming patterns in a refined, succinct, and type-safe way.
- Scala programs can convert to bytecodes and can run on the JVM**(Java Virtual Machine)**.
- Scala stands for ***Scalable*** **language.**


Object Oriented
Functional
Scala
Can Execute Java Code
Statically typed
Runs on the JVM


SCALA Blockchain
USES OF SCALA
Scripting
Web Application
Highly Scalable & Performing Backends
Mobile (Android) Applications
Graphical User Interface
Messaging

Spark Shell:

- The shell acts as an interface to access the operating system's service.
- **Apache Spark** is shipped with an interactive shell/scala prompt with the interactive shell we can run different commands to process the data.
- we can create **RDDs** and perform various transformations and actions like filter(), partitions(), cache(), count(), collect, etc.

Scala – Spark Shell Commands:

- **Apache Spark** is shipped with an interactive shell/scala prompt, as the spark is developed in **Scala**. Using the interactive shell we will run different commands (RDD transformation/action**)** to process the data.

- The command to start the Apache Spark Shell:

  $bin/spark-shell

- **Create a new RDD:**

  a) Read File from local filesystem and create an RDD.

  scala> val data = sc.textFile("data.txt")

- sc is the object of **SparkContext**

- **Create an RDD through Parallelized Collection:**

```
scala> val no = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val noData = sc.parallelize(no)
```

- **From Existing RDDs:**

```
scala> val newRDD = no.map(data => (data * 2))
```

These are **three methods to create the RDD**. We can use the first method, when data is already available with the external systems like a local filesystem, **HDFS**, **HBase**, etc. One can create an RDD by calling a **textFile** method of **Spark Context** with path / URL as the argument. The second approach can be used with the existing collections and the third one is a way to create new RDD from the existing one.

- **Number of Items in the RDD:**

Count the number of items available in the RDD. To count the items we need to call an Action:

scala> data.count()

- **Filter Operation:**

Filter the RDD and create new RDD of items which contain word "hello". To filter, we need to call transformation filter, which will return a new RDD with subset of items.

scala> val DFData = data.filter (line => line.contains("hello"))

- **Transformation and Action together:**

For complex requirements, we can chain multiple operations together like filter (transformation) and count (action) together:

scala> data.filter(line => line.contains(“Hello”)).count()[/php]

- **Read the first item from the RDD:**

To read the first item from the file, you can use the following command:

scala> data.first()

- **Read the first 5 item from the RDD:**

To read the first 5 item from the file, you can use the following command:

scala> data.take(5)

- **RDD Partitions:**

An RDD is made up of multiple partitions, to count the number of partitions:

scala> data.partitions.length

The Minimum no. of partitions in the RDD is **2 (by default).** When we create RDD from HDFS file then a number of blocks will be equals to the number of partitions.

- **Read Data from HDFS file:**

To read data from HDFS file we can specify complete hdfs URL like **hdfs://IP:PORT/PATH**

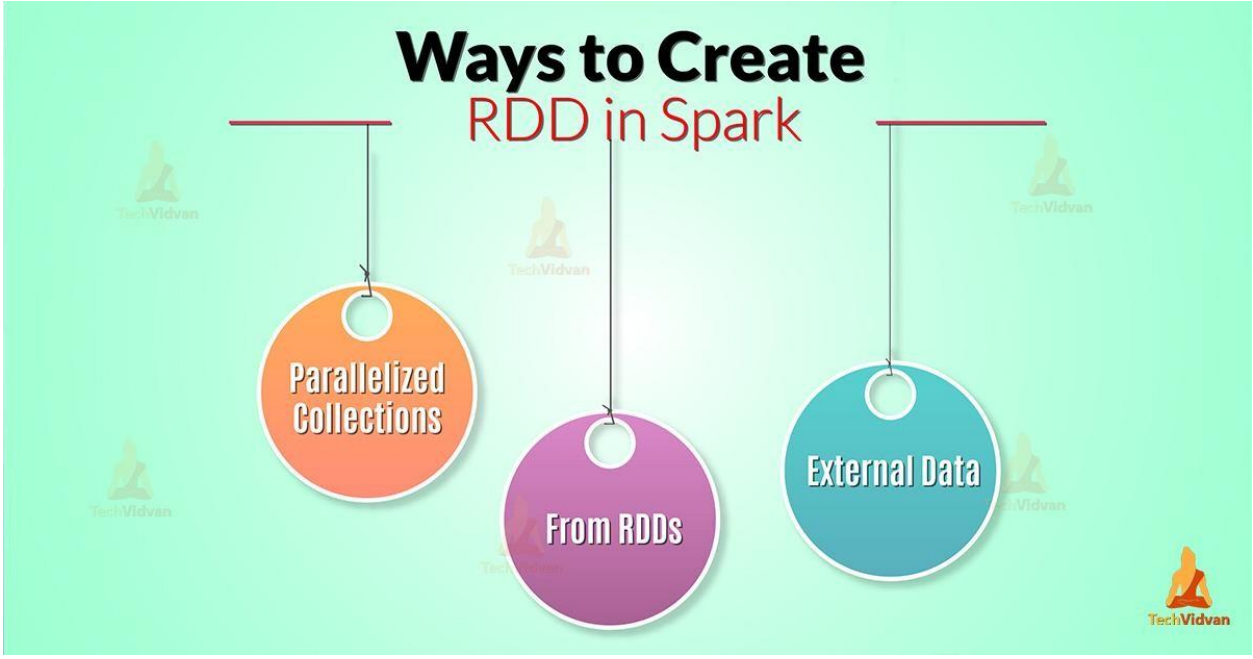
scala> var hFile = sc.textFile("hdfs://localhost:9000/inp")

- **write the data to HDFS file**

To write the data from HFDS:

[Php]scala> wc.Saveastextfile("hdfs://localhost:9000/out")[/php]

CREATION OF RDD in SPARK:



- **Parallelized Collection**
- **External Datasets**
- **From Existing RDD's**

Parallelized Collection:

- To create parallelized collection, call **sparkcontext's** parallelize method on an existing collection in the driver program. Each element of collection is copied to form a distributed dataset that can be operated on in parallel.
- RDDs can be created generally by the **parallelizing method.**

- It is possible by taking an existing collection from our driver program.

- Driver program such as Scala, Python, Java. Also by calling the sparkcontext's parallelize( ) method on it.
- This is a basic method to create RDD which is applied at the very initial stage of spark.
- It creates RDD very quickly. It also initializes further operations on them at the same time.
- To operate this method, we need entire dataset on one machine.

Example: Val info = array(1, 2, 3, 4)

Val distinfo = sc.Parallelize(info)

Now, we can operate the distributed dataset (distinfo) parallel such like distinfo.Reduce((a, b) => a + b).

➢ External Datasets

- In Spark, the distributed datasets can be created from any type of storage sources supported by Hadoop such as HDFS, Cassandra, HBase and even our local file system. Spark provides the support for text files, **SequenceFiles**, and other types of Hadoop **InputFormat**.
- **SparkContext's** textFile method can be used to create RDD's text file. This method takes a URI for the file (either a local path on the machine or a hdfs://) and reads the data of the file.
-

```
scala> val data=sc.textFile("sparkdata.txt");
data: org.apache.spark.rdd.RDD[String] = sparkdata.txt MapPartitionsRDD[1] at te
xtFile at <console>:24
```

  - If any storage source supported by Hadoop, including our local file system(HDFS,HBase,Hive) it can create RDDs from it.
  - Apache spark does support sequence files, textfiles, and any other Hadoop input format.
  - We can create textfile RDDs by sparkcontext's textfile method.
  - It also reads whole as a collection of lines.
  - Always be careful that the path of the local system and worker node should always be similar. The file should be available at the same place in the local file system and worker node.
  - To load a dataset from an external storage system, we can use data frame reader interface. External storage system such as file systems, key-value stores. It supports many file formats like(JSON,Text file,CSV file etc)

- **From Existing RDD's:**
- As RDD is immutable so, we can not change anything to it. So we can create different RDD from the existing RDDs.
- This process of creating another dataset from the existing ones means transformation.
- As a result, transformation always produces new RDD.
- As they are immutable, no changes take place in it if once created.
- This property maintains the consistency over the cluster.
- Some of the operations performed on RDD are map, filter, count, distinct, flatmap etc.

For Example:

In this example, we are providing a text file which returns Dataset of the string as a result.

```
val words=spark.sparkContext.parallelize(Seq("sun", "rises", "in", "the", "east", "and", "sets", "in", "the", "west"))

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

wordPair.foreach(println)
```

In the above example RDD “wordPair” is created from existing RDD “word” using map ( ) transformation. This result contains word and starting character together of the same word.

RDD Operations (Passing Functions To Spark, Transformations And Actions In Spark):

- The RDD provides the two types of operations:
  - **Transformation**
  - **Action**

Transformation

In spark, the role of transformation is to create a new dataset from an existing one. The transformations are considered lazy as they only computed when an action requires a result to be returned to the driver program.

Let's see some of the frequently used RDD transformations.

Transformation	Description
map(func)	It returns a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	It returns a new dataset formed by selecting those elements of the source on which func returns true.

flatMap(func)	Here, each input item can be mapped to zero or more output items, so func should return a sequence rather than a single item.
mapPartitions(func)	It is similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

mapPartitionsWithIndex(func)	It is similar to mapPartitions that provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	It samples the fraction fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	It returns a new dataset that contains the union of the elements in the source dataset and the argument.

intersection(otherDataset)	It returns a new RDD that contains the intersection of elements in the source dataset and the argument.

distinct([numPartitions]))	It returns a new dataset that contains the distinct elements of the source dataset.
groupByKey([numPartitions])	It returns a dataset of (K, Iterable) pairs when called on a dataset of (K, V) pairs.
reduceByKey(func, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.

sortByKey([ascending], [numPartitions])	It returns a dataset of key-value pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numPartitions])	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.
cogroup(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable, Iterable)) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script.
coalesce(numPartitions)	It decreases the number of partitions in the RDD to numPartitions.

repartition(numPartitions)	It reshuffles the data in the RDD randomly to create either more or fewer partitions and balance it across them.
repartitionAndSortWithinPartitions(partitioner)	It repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

Action:

In Spark, the role of action is to return a value to the driver program after running a computation on the dataset.

Action	Description
reduce(func)	It aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

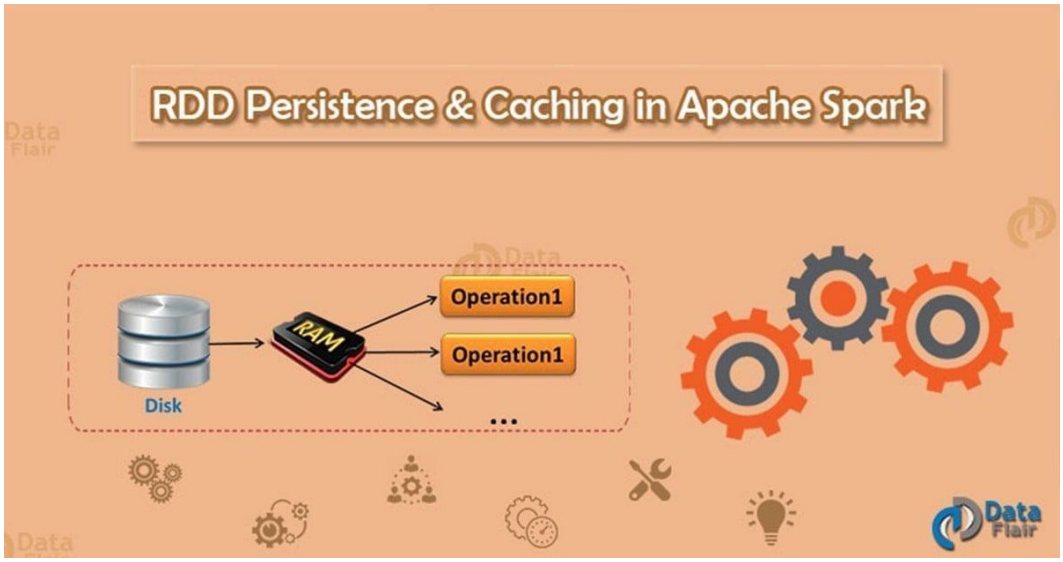
collect()	It returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	It returns the number of elements in the dataset.
first()	It returns the first element of the dataset (similar to take(1)).
take(n)	It returns an array with the first n elements of the dataset.

takeSample(withReplacement, num, [seed])	It returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	It returns the first n elements of the RDD using either their natural order or a custom

	comparator.
saveAsTextFile(path)	It is used to write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	It is used to write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system.
saveAsObjectFile(path) (Java and Scala)	It is used to write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().
countByKey()	It is only available on RDDs of type (K, V). Thus, it returns a hashmap of (K, Int) pairs with the count of each key.

foreach(func)	It runs a function func on each element of the dataset for side effects such as updating an Accumulator or interacting with external storage systems.

Spark RDD Persistence:



- Spark **RDD** persistence is an optimization technique in which saves the result of RDD evaluation.
- Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.
- We can make persisted RDD through **cache()** and **persist()** methods.
- When we use the cache() method we can store all the RDD in-memory.
- We can persist the RDD in memory and use it efficiently across parallel operations.

- The **difference between cache() and persist()** is that using cache() the default storage level is **MEMORY_ONLY** while using **persist()** we can use various storage levels .It is a key tool for an interactive algorithm. Because, when we persist RDD each node stores any partition of it that it computes in memory and makes it reusable for future use. This process speeds up the further computation ten times.

Storage levels of Persisted RDDs

Using **persist()** we can use various storage levels to Store Persisted RDDs in Apache Spark.

a. MEMORY_ONLY

- In this storage level, RDD is stored as deserialized Java object in the JVM.
- If the size of RDD is greater than memory, It will not cache some partition and recompute them next time whenever needed.
- In this level the space used for storage is very high, the CPU computation time is low, the data is stored in-memory. It does not make use of the disk.

b. MEMORY_AND_DISK

- In this level, RDD is stored as deserialized Java object in the JVM.
- When the size of RDD is greater than the size of memory, it stores the excess partition on the disk, and retrieve from disk whenever required.

- In this level the space used for storage is high, the CPU computation time is medium, it makes use of both in-memory and on disk storage.

c. MEMORY_ONLY_SER

- This level of Spark store the RDD as serialized Java object (one-byte array per partition). It is more space efficient as compared to deserialized objects,especially when it uses fast serializer.
- But it increases the overhead on CPU.
- In this level the storage space is low, the CPU computation time is high and the data is stored in-memory. It does not make use of the disk.

d. DISK_ONLY

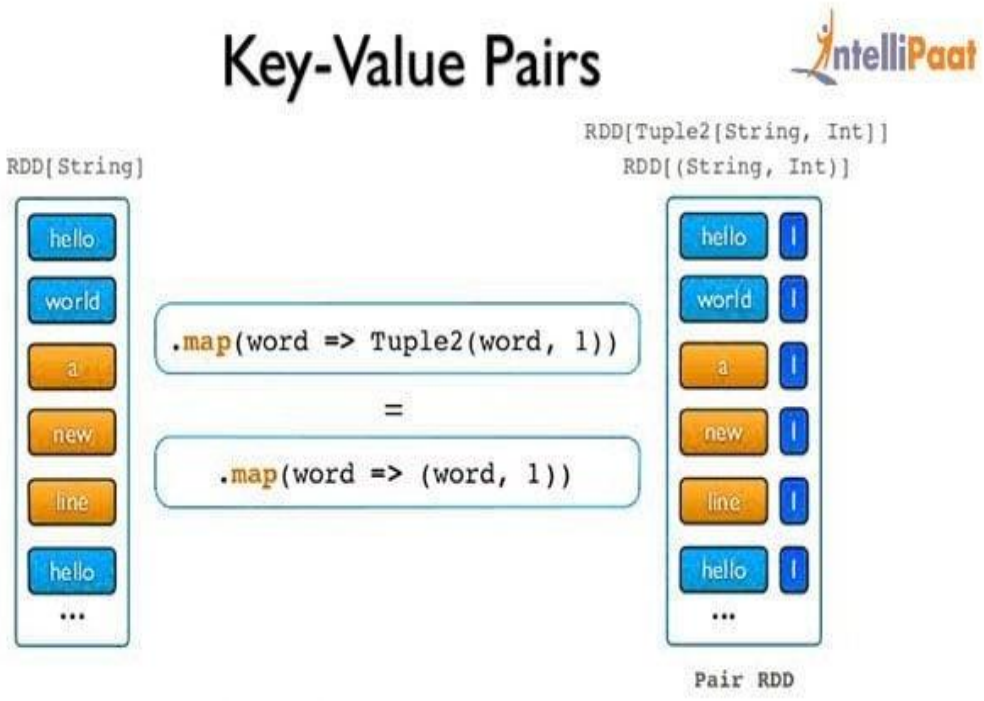
- In this storage level, RDD is stored only on disk.
- The space used for storage is low, the CPU computation time is high and it makes use of on disk storage.

How to Unpersist RDD in Spark

- Spark monitor the cache of each node automatically and drop out the old data partition in the LRU (least recently used) fashion.
- LRU is an algorithm which ensures the least frequently used data.
- It spills out that data from the cache.
- We can also remove the cache manually using **RDD.unpersist()** method.

Working with Key/Value Pairs: Pair RDDs, Transformations on Pair RDDs, Actions Available on Pair RDDs

- **Spark** provides special types of operations on RDDs that contain key/value pairs (Paired RDDs). These operations are called paired RDDs operations. Paired RDDs are a useful building block in many programming languages, as they expose operations that allow us to act on each key operation in parallel or re-group data across the network.
- Transformations and actions in spark RDD. Here transformation operations are groupByKey, reduceByKey, join, left outer join/right OuterJoin. Whereas actions like countByKey. However initially, we will learn a brief introduction to spark RDDs.
- So, let's start spark paired RDD tutorial.



- Spark Paired RDDs are nothing but RDDs containing a key-value pair. Basically, key-value pair (KVP) consists of a two linked data item in it.
- Here, the key is the identifier, whereas value is the data corresponding to the key value.
- Moreover, Spark operations work on RDDs containing any type of objects.
- However key-value pair RDDs attains few special operations in it. Such as, distributed "shuffle" operations, grouping or aggregating the elements by a key.
- In addition, on Spark Paired RDDs containing Tuple2 objects in Scala, these operations are automatically available. Basically, operations for the key-value pair are available in the Pair RDD functions class.
- However, that wraps around a Spark RDD of tuples.

How to Create Spark Paired RDD

- There are several ways to create Paired RDD in Spark, like by running a map() function that returns key-value pairs. However, language differs the procedure to build the key-value RDD. Such as

a. In Python language

It is a requirement to return an RDD composed of Tuples for the functions of keyed data to work. Moreover, in spark for creating a pair RDD, we use the first word as the key in python programming language.
pairs = lines.map(lambda x: (x.split(" ")[0], x))

b. In Scala language

As similar to the previous example here also we need to return tuples. Furthermore, this will make available the functions of keyed data. Also, to offer the extra key or value functions, an implicit conversion on Spark RDD of tuples exists.

- **Let's revise Data type Mapping between R and Spark**
  Afterward, again by using the first word as the keyword creating apache spark pair RDD.
  val pairs = lines.map(x => (x.split(" ")(0), x))

c. In Java language

- Basically, Java doesn't have a built-in function of tuple function. Therefore, we can use the Scala.
- It only sparks' Java API has users create tuples.Tuple2 class. However by, by writing new Tuple2(elem1, elem2) in Java, we can create a new tuple.
- Moreover, we can access its relevant elements with the _1() and _2() methods.
- Moreover, when we create paired RDDs in Spark, it is must to call special versions of spark's functions in java. As an example, we can use mapToPair () function in place of the basic map() function. Again, here using the first word as the keyword to create a Spark paired RDD,

```
PairFunction<String, String, String> keyData = new PairFunction<String, String, String>()

{

public Tuple2<String, String> call(String x)

{
```

return new Tuple2(x.split(" ")[0], x);

}

};

JavaPairRDD<String, String> pairs = lines.mapToPair(keyData)

Spark Paired RDD Operations:

A. Transformation Operations

- Paired RDD allows the same transformation those are available to standard RDDs.
- Moreover, here also same rules apply from "passing functions to spark".
- Also in Spark, there are tuples available in paired RDDs. Basically, we need to pass functions that operate on tuples, despite on individual elements.
- Let's discuss some of the transformation methods below, like

    - groupByKey
    - The groupbykey operation generally groups all the values with the same key. rdd.groupByKey()
    - reduceByKey(fun)

Here, the reduceByKey operation generally combines values with the samekey.
add.reduceByKey((x, y) => x + y)

- combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)

- CombineByKey uses a different result type, then combine those values with the same key.

- **mapValues(func)**

Even without changing the key, mapValues operation applies a function to each value of a paired RDD of spark. rdd.mapValues(x => x+1)

- **keys()**

Keys() operation generally returns a spark RDD of just the keys. rdd.keys()

- **values()**

values() operation generally returns an RDD of just the values. rdd.values()

- **sortByKey()**

Similarly, the sortByKey operation generally returns an RDD sorted by thekey.
rdd.sortByKey()

B. **Actionoperations**:

As similar as RDD transformations, there are same RDD actions available on spark pair RDD. However, paired rdds also attains some additional actions of spark. Basically, those leverages the advantage of data which is of keyvalue nature. Let's discuss some of the action methods below, like

- countByKey()

Through countByKey operation, we can count the number of elements for each key.
rdd.countByKey()

- collectAsMap()

Here, collectAsMap() operation helps to collect the result as a map to provide easy lookup.
rdd.collectAsMap()

- lookup(key)

Moreover, it returns all values associated with the provided key.
rdd.lookup()