**Apache Spark**

Apache Spark is an open-source cluster computing framework that is setting the world of Big Data on fire. When compared to Hadoop, Spark's performance is up to 100 times faster in memory and 10 times faster on disk.

**Spark & its Features**

Apache Spark is an open-source cluster computing framework for real-time data processing. The main feature of Apache Spark is its in-memory cluster computing that increases the processing speed of an application. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries, and streaming.

# Features of Apache Spark:

**Speed:**Spark runs up to 100 times faster than Hadoop MapReduce for large-scale data processing. It is also able to achieve this speed through controlled partitioning.

**Powerful Caching** Simple programming layer provides powerful caching and disk persistence capabilities.

**Deployment**  
It can be deployed through Mesos, Hadoop via YARN, or Spark’s own cluster manager.

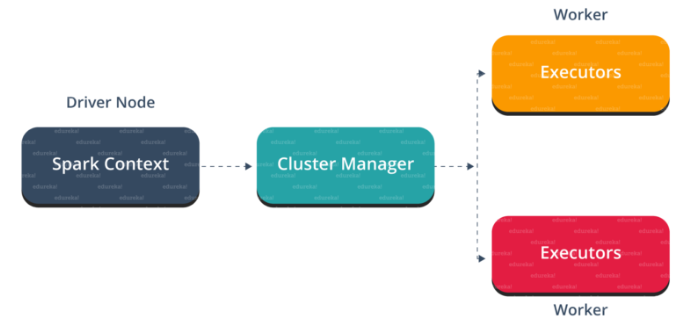
**Real-Time**  
It offers Real-time computation & low latency because of in-memory computation.

**Polyglot**  
Spark provides high-level APIs in Java, Scala, Python, and R. Spark code can be written in any of these four languages. It also provides a shell in Scala and Python.

# Spark Architecture Overview

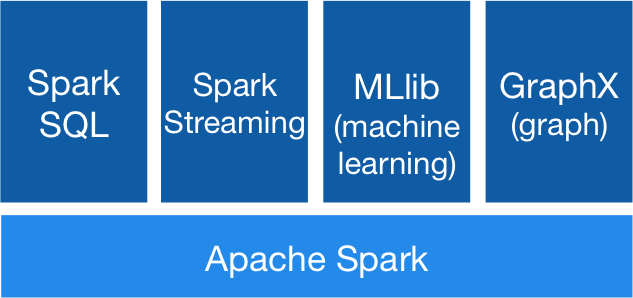
Apache Spark has a well-defined layered architecture where all the spark components and layers are loosely coupled. This architecture is further integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions:

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



# Spark Eco-System

The spark ecosystem is composed of various components like Spark SQL, Spark Streaming, MLlib, GraphX, and the Core API component.

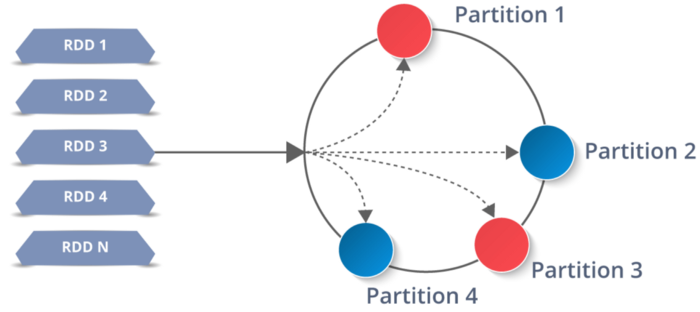


* Apache Spark: Sometimes also called Spark Core. The Spark Core implementation is a RDD (Resilient Distributed Dataset) which is a collection of distributed data across different nodes of the cluster that are processed in parallel.
* Spark SQL: The implementation here is DataFrame, which is a relational representation of the data. It provides functions with SQL like capabilities. Also, we can write SQL like queries for our data analysis.
* Spark Streaming: The implementation provided by this library is D-stream, also called Discretized Stream. This library provides capabilities to process/transform data in near real-time.
* MLlib: This is a Machine Learning library with commonly used algorithms including collaborative filtering, classification, clustering, and regression.
* GraphX: This library helps us to process Graphs, solving various problems (like Page Rank, Connected Components, etc) using Graph Theory.

**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

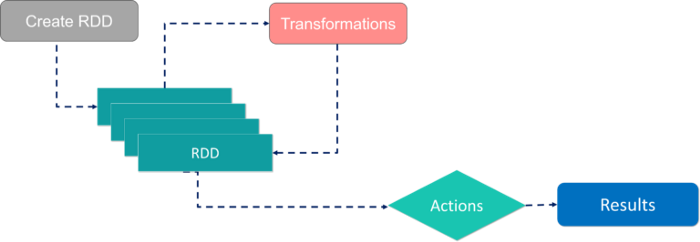


It is a layer of abstracted data over the distributed collection. It is immutable in nature and follows lazy transformations.

Now you might be wondering about its working. Well, the data in an RDD is split into chunks based on a key. RDDs are highly resilient, i.e., they are able to recover quickly from any issues as the same data chunks are replicated across multiple executor nodes. Thus, even if one executor node fails, another will still process the data. This allows you to perform your functional calculations against your dataset very quickly by harnessing the power of multiple nodes.

Moreover, 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.

Talking about the distributed environment, each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. Due to this, you can perform transformations or actions on the complete data parallelly. Also, you don’t have to worry about the distribution, because Spark takes care of that.



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.

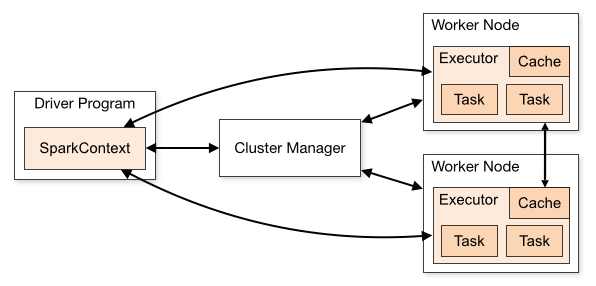
With RDDs, you can perform two types of operations:

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

# Working of Spark Architecture

As you have already seen the basic architectural overview of Apache Spark, now let’s dive deeper into its working.

In your **master node**, you have the driver program, which drives your application. The code you are writing behaves as a driver program or if you are using the interactive shell, the shell acts as the driver program.



Inside the driver program, the first thing you do is, you create a Spark Context. Assume that the Spark context is a gateway to all the Spark functionalities. It is similar to your database connection. Any command you execute in your database goes through the database connection. Likewise, anything you do on Spark goes through Spark context.

Now, this Spark context works with the cluster manager to manage various jobs. The driver program & Spark context takes care of the job execution within the cluster. A job is split into multiple tasks which are distributed over the worker node. Anytime an RDD is created in Spark context, it can be distributed across various nodes and can be cached there.

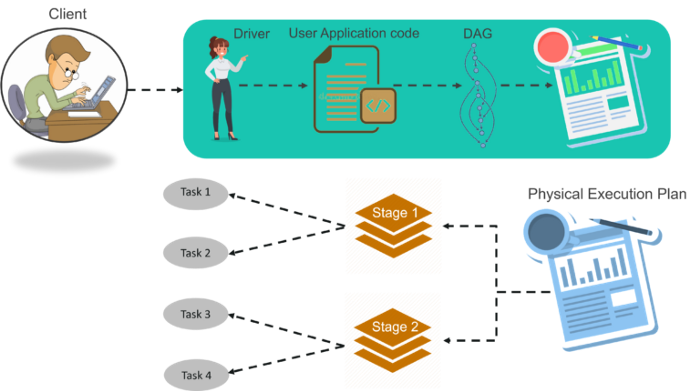
**W**orker nodes are the slave nodes whose job is to basically execute the tasks. These tasks are then executed on the partitioned RDDs in the worker node and hence returns back the result to the Spark Context.

Spark Context takes the job, breaks the job in tasks and distributes them to the worker nodes. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main Spark Context.

If you increase the number of workers, then you can divide jobs into more partitions and execute them parallel over multiple systems. It will be a lot faster.

With the increase in the number of workers, memory size will also increase & you can cache the jobs to execute it faster.

To know about the workflow of Spark Architecture, you can have a look at the **info graphic** below:



## ****STEP 1:****

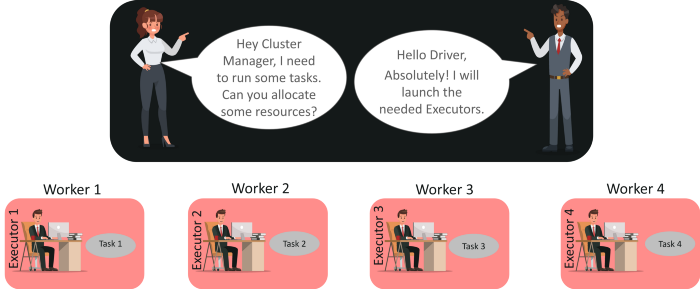
The client submits spark user application code. When application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically directed acyclic graph called DAG. At this stage, it also performs optimizations such as pipelining transformations.

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

* Create a RDD based on Python collection.

keywords = [‘Books’, ‘DVD’, ‘CD’, ‘PenDrive’]key\_rdd = sc.parallelize(keywords)

In the above code “keywords” is a python collection (List) and we are creating a RDD from a Python collection using the “parallelize” method of the Spark Context Class.

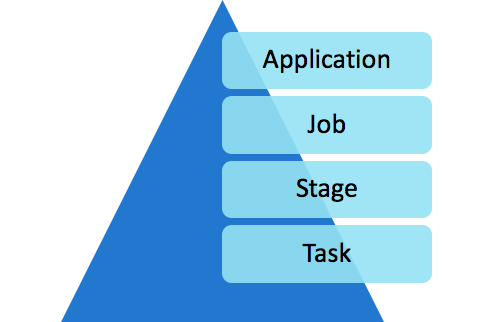
* Create a RDD from a file

file\_rdd = sc.textFile(“Path\_to\_File”)

There are two types of operations performed on a RDD :

* Transformations: These operations work in a lazy fashion. When you apply a transformation on a RDD it will not be evaluated immediately but will only be stored in a DAG (Directed Acyclic Graph) and will be evaluated at some later point of time after an action is executed. Some common transformations are map, filter, flatMap, groupByKey, reduceByKey, etc.
* Actions: These operations will be executed immediately. Some common actions are first, last, count, collect, etc.

Tip: RDDs are immutable in nature, you cannot change RDDs. However, you can transform one RDD to another using various Transformations.



* Application: When we submit the Spark code to a cluster it creates a Spark Application.
* Job: The Job is the top-level execution for any Spark application. A Job corresponds to an Action in a Spark application.
* Stage: Jobs will be divided into stages. The Transformations work in a lazy fashion and will not be executed until an Action is called. Actions might include one or many Transformations and the Transformations define the breakdown of jobs into stages, which corresponds to a shuffle dependency.
* Task: Stages will be further divided into various tasks. The task is the most granular unit in Spark applications. Each task represents a local computation on a particular node in the Spark Cluster.

Now we have a understanding of Spark, Spark Architecture, RDDs and the anatomy of a Spark Application. Let’s get our hands dirty with some hands-on exercises.

You can execute your Spark code by using a shell (Spark-shell or pyspark), Jupyter Notebooks, Zeppelin Notebooks, Spark-submit, etc.

Let’s create a RDD and understand some basic transformations.

* Create a RDD from a collection.

num = [1,2,3,4,5]num\_rdd = sc.parallelize(num)

Here num\_rdd is an RDD based on python collection (list).

Transformations

As we know, Transformations are lazy in nature and they will not be executed until an Action is executed on top of them. Let’s try to understand various available Transformations.

* Map: This will map your input to some output based on the function specified in the map function.

We already have “num\_rdd” created. Let’s try to double each number in RDD.

double\_rdd = num\_rdd.map(lambda x : x \* 2)

Note: The expression specified inside the map function is another function without any name which is called a lambda function or anonymous function.

* Filter: To filter the data based on a certain condition. Let’s try to find the even numbers from num\_rdd.

even\_rdd = num\_rdd.filter(lambda x : x % 2 == 0)

* flatMap: This function is very similar to map, but can return multiple elements for each input in the given RDD.

flat\_rdd = num\_rdd.flatMap(lambda x : range(1,x))

This will return the range object for each element in the input RDD (num\_rdd).

* distinct: This will return distinct elements from an RDD.

rdd1 = sc.parallelize([10, 11, 10, 11, 12, 11])dist\_rdd = rdd1.distinct()

The above Transformations are single-valued, where each element within a RDD contains a single scalar value. Let’s discuss some key-value pair RDDs, where each element of the RDD will be a (key, value) pair.

* reduceByKey: This function reduces the key values pairs based on the keys and a given function inside the reduceByKey. Here’s an example.

pairs = [ (“a”, 5), (“b”, 7), (“c”, 2), (“a”, 3), (“b”, 1), (“c”, 4)]pair\_rdd = sc.parallelize(pairs)

pair\_rdd is now key-value pair RDD.

output = pair\_rdd.reduceByKey(lambda x, y : x + y)

the output RDD will contain the pairs :

[ (“a”, 8), (“b”, 8), (“c”, 6) ]

Let’s try to understand the contents of the output RDD here. We can think of the reduceByKey function in 2 steps.

1. It will collect all the values for a given key. So the intermediate output will be as follows :

(“a”, <5,3>)(“b”, <7, 1>)(“c”, <2, 4>)

2. Now we have all the values corresponding to a particular key. Then the “values” collection will be reduced or aggregated based on the function mentioned inside the reduceByKey. In our case it is the sum function, so we are getting the sum of all the values for a given key. Hence the output is :

[ (“a”, 8), (“b”, 8), (“c”, 6) ]

* groupByKey: This function is another ByKey function which can operate on a (key, value) pair RDD but this will only group the values based on the keys. In other words, this will only perform the first step of reduceByKey.

grp\_out = pair\_rdd.groupByKey()

* sortByKey: This function will perform the sorting on a (key, value) pair RDD based on the keys. By default, sorting will be done in ascending order.

pairs = [ (“a”, 5), (“d”, 7), (“c”, 2), (“b”, 3)]raw\_rdd = sc.parallelize(pairs)sortkey\_rdd = raw\_rdd.sortByKey()

This will sort the pairs based on keys. So the output will be

[ (“a”, 5), (“b”, 3), (“c”, 2), (“d”, 7)]

Note: for sorting in descending order pass “ascending=False”.

* sortBy: sortBy is a more generalized function for sorting.

pairs = [ (“a”, 5, 10), (“d”, 7, 12), (“c”, 2, 11), (“b”, 3, 9)]raw\_rdd = sc.parallelize(pairs)

Now we have got a RDD of tuples where each tuple has 3 elements in it. Let’s try to do the sorting based on the 3rd element of the tuple.

sort\_out = raw\_rdd.sortBy(lambda x : x[2])

Note: for sorting in descending order pass “ascending=False”.

**Actions**

Actions are operations on RDDs which execute immediately. While Transformations return another RDD, Actions return language native data structures.

* Count: This will count the number of elements in the given RDD.

num = sc.parallelize([1,2,3,4,2])num.count() # output : 5

* First: This will return the first element from given RDD.

num.first() # output : 1

* Collect: This will return all the elements for the given RDD.

num.collect() # output : [1,2,3,4,2]

Note: We should not use the collect operation while working with large datasets. Because it will return all the data which is distributed across the different workers of your cluster to a driver. All the data will travel across the network from worker to driver and also the driver would need to hold all the data. This will hamper the performance of your application.

* Take: This will return the number of elements specified.

num.take(3) # output : [1, 2, 3]

Let’s do the comparison between **SparkSession vs SparkContext**.

### What Is SparkContext?

SparkContext is the primary point of entry for **Spark capabilities**. A SparkContext represents a Spark cluster’s connection that is useful in building RDDs, accumulators, and broadcast variables on the cluster. It enables your Spark Application to connect to the **Spark Cluster** using Resource Manager. Also, before the creation of SparkContext, SparkConf must be created.

After creating the SparkContext, you can use it to create RDDs, broadcast variables, and accumulators, as well as access Spark services and perform jobs. All of this can be done until SparkContext is terminated. Access to the other two contexts, SQLContext and HiveContext, is also possible through SparkContext. Since Spark 2.0, most **SparkContext functions** are also available in SparkSession. SparkContext’s default object sc is provided in Spark-Shell, and it can also be constructed programmatically using the SparkContext class. As a result, SparkContext provides numerous Spark functions. This includes getting the current status of the **Spark Application**, setting the configuration, canceling a task, canceling a stage, and more. It was a means to get started with all the Spark features prior to the introduction of SparkSession, as shown in this **SparkSession Vs SparkContext** comparison post.

### What Is SparkSession?

**Apache Spark 2.0** is the company’s next significant release. This is a significant shift in the degree of abstraction for the Spark API and libraries. Previously, as RDD was the major API, SparkContext was the entry point for Spark. It was constructed and modified with the help of context APIs. At that time, we have to use a distinct context for each API. We required StreamingContext for Streaming, SQLContext for SQL, and HiveContext for Hive. However, because the DataSet and DataFrame APIs are becoming new independent APIs, we require an entry-point construct for them. As a result, in Spark 2.0, we have a new entry point built for DataSet and DataFrame APIs called **SparkSession**.

It combines SQLContext, HiveContext, and StreamingContext. All of the APIs accessible in those contexts are likewise available in SparkSession, and SparkSession includes a **SparkContext** for real computation. It’s worth noting that the previous SQLContext and HiveContext are still present in updated versions, but only for backward compatibility. As a result, when comparing **SparkSession vs SparkContext**, as of Spark 2.0.0, it is better to use SparkSession because it provides access to all of the Spark features that the other three APIs do. Its Spark object comes by default in Spark-shell, and it can be generated programmatically using the SparkSession builder pattern.

## Why Should You Use SparkSession Over SparkContext?

From Spark 2.0, **SparkSession** provides a common entry point for a Spark application. It allows you to interface with Spark’s numerous features with a less amount of constructs. Instead of SparkContext, HiveContext, and SQLContext, everything is now within a SparkSession. One aspect of the explanation why SparkSession is preferable over SparkContext in SparkSession Vs SparkContext battle is that SparkSession unifies all of Spark’s numerous contexts, removing the developer’s need to worry about generating separate contexts

from pyspark.sql import SparkSession

spark=SparkSession.builder.appName("Name").getOrCreate()