***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.

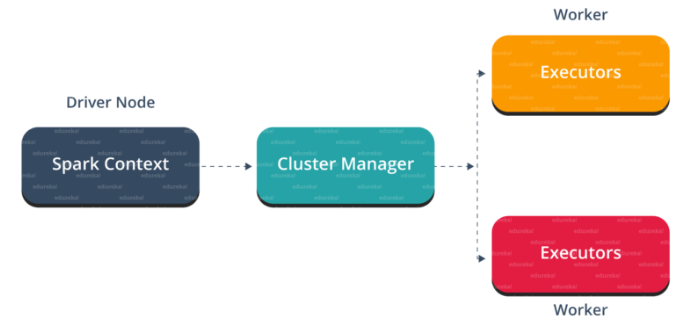
### ****Features of Apache Spark:****

1. **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.
2. **Powerful Caching**Simple programming layer provides powerful caching and disk persistence capabilities.
3. **Deployment**It can be deployed through ***Mesos, Hadoop via YARN, or Spark’s own cluster manager.***
4. **Real-Time**  
   It offers Real-time computation & low latency because of ***in-memory computation.***
5. **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**

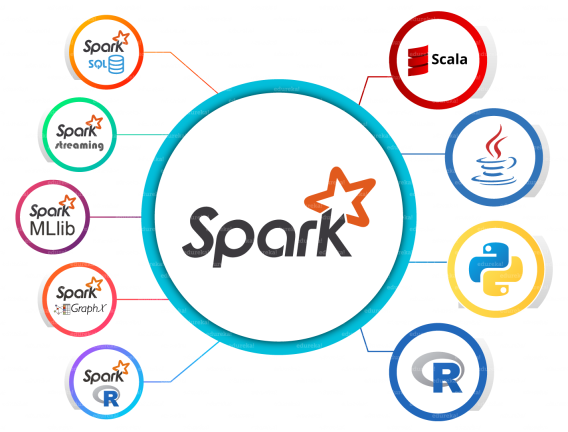
Apache Spark has a well-defined layered architecture where all the spark components and layers are loosely coupled. This architecture is further integrated with various extensions and libraries.Apache Spark Architecture is based on two main abstractions:

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



But before diving any deeper into the Spark architecture, let me explain few fundamental concepts of Spark like Spark Eco-system and RDD. This will help you in gaining better insights.

## ****Spark Eco-System****

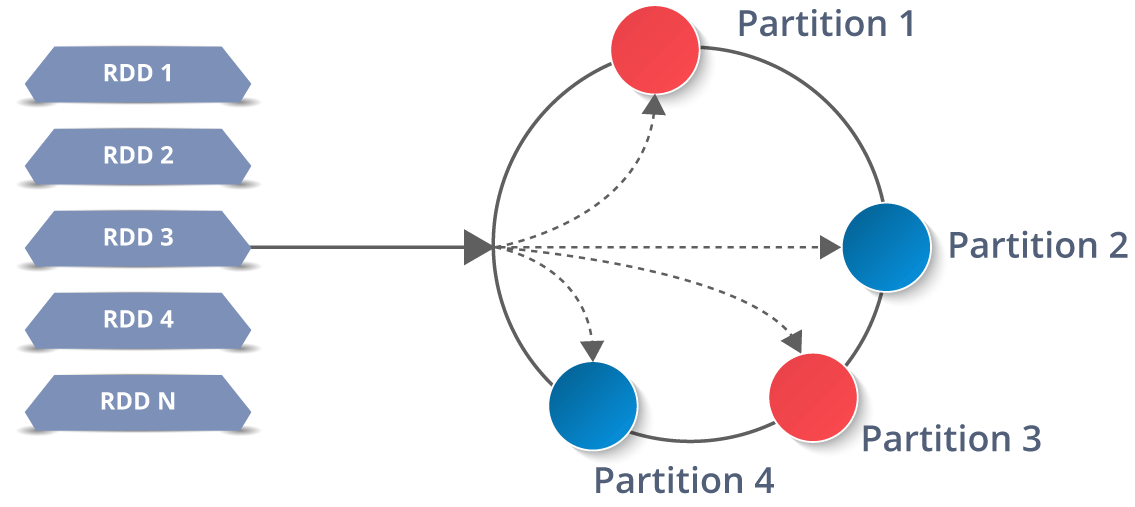
As you can see from the below image, the spark ecosystem is composed of various components like Spark SQL, Spark Streaming, MLlib, GraphX, and the Core API component.

1. **Spark Core**Spark Core is the base engine for large-scale parallel and distributed data processing. Further, additional libraries which are built on the top of the core allows diverse workloads for streaming, SQL, and machine learning. It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster & interacting with storage systems.
2. **Spark Streaming**Spark Streaming is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams.
3. **Spark SQL**Spark SQL is a new module in Spark which integrates relational processing with Spark’s functional programming API. It supports querying data either via SQL or via the Hive Query Language. For those of you familiar with RDBMS, Spark SQL will be an easy transition from your earlier tools where you can extend the boundaries of traditional relational data processing.
4. **GraphX**GraphX is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph (a directed multigraph with properties attached to each vertex and edge).
5. **MLlib (Machine Learning)**  
   MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.
6. ***SparkR***It is an R package that provides a distributed data frame implementation. It also supports operations like selection, filtering, aggregation but on large data-sets.

As you can see, Spark comes packed with high-level libraries, including support for R, SQL, Python, Scala, Java etc. These standard libraries increase the seamless integrations in a complex workflow. Over this, it also allows various sets of services to integrate with it like MLlib, GraphX, SQL + Data Frames, Streaming services etc. to increase its capabilities.

**Resilient Distributed Dataset(RDD)**

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

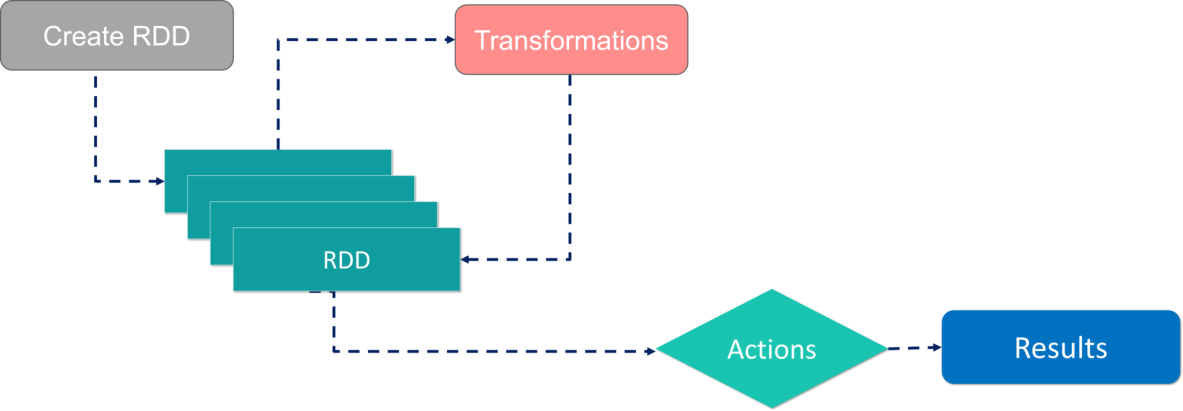
* ***Resilient:*** Fault tolerant and is capable of rebuilding data on failure
* ***Distributed:*** Distributed data among the multiple nodes in a cluster
* ***Dataset:*** Collection of partitioned data with values

It is a layer of abstracted data over the distributed collection. It is immutable in nature and follows [*lazy transformations*](https://www.edureka.co/blog/spark-tutorial/#Spark_Features).

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.

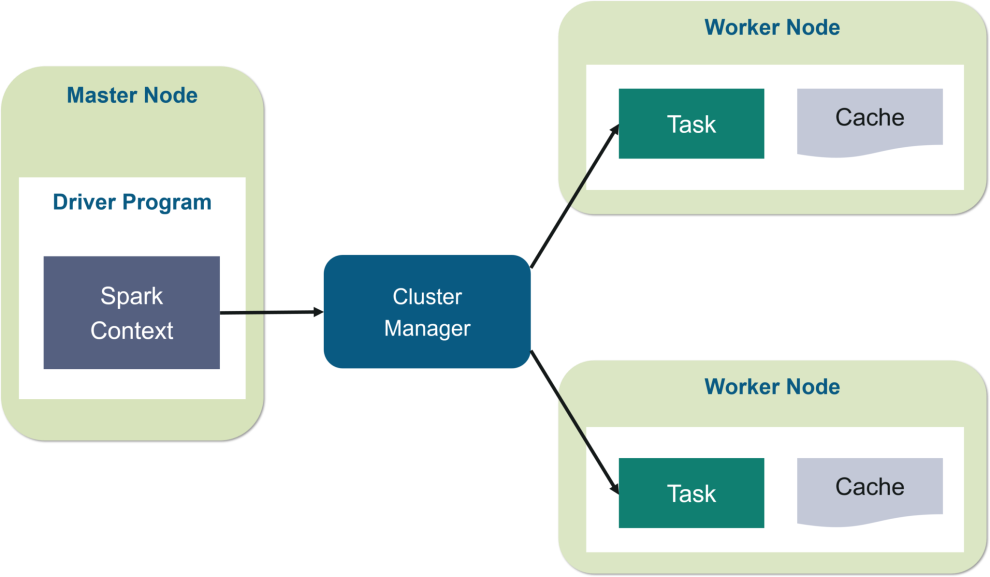


There are two ways to create RDDs − parallelizing an existing collection in your driver program, or by referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, etc.

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.

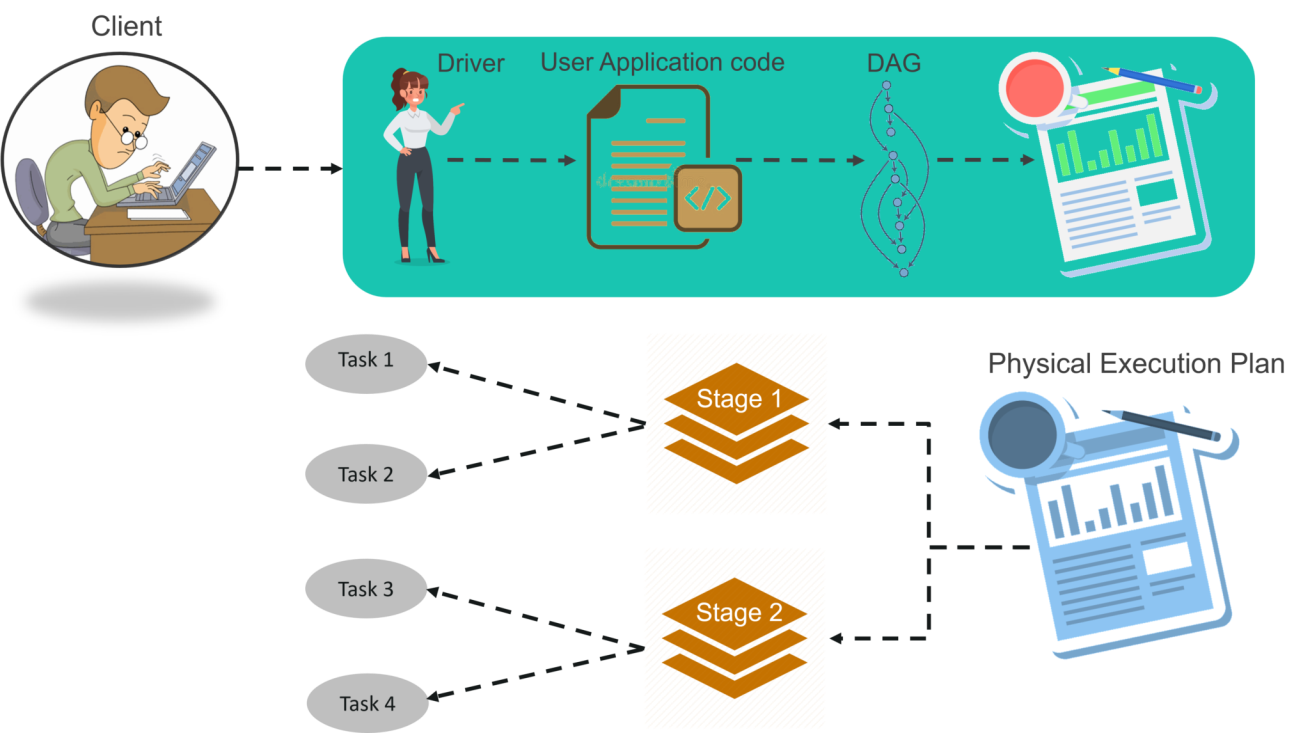
Now, this Spark context works with the ***cluster manager*** to manage various jobs. The driver program & Spark context takes care of the job execution within the cluster. A job is split into multiple tasks which are distributed over the worker node. Anytime an RDD is created in Spark context, it can 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 distribute 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 parallelly 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 **infographic** below:

**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.