# **Apache Spark**

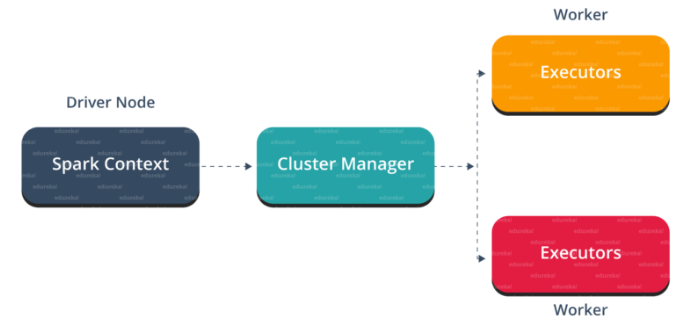
Apache Spark is an open source cluster computing framework for real-time data processing. The main feature of Apache Spark is its ***in-memory cluster computing*** that increases the processing speed of an application.

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.

**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**

1. **SparkCore**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. **SparkStreaming**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. **SparkSQL**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**   
   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.

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)

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.

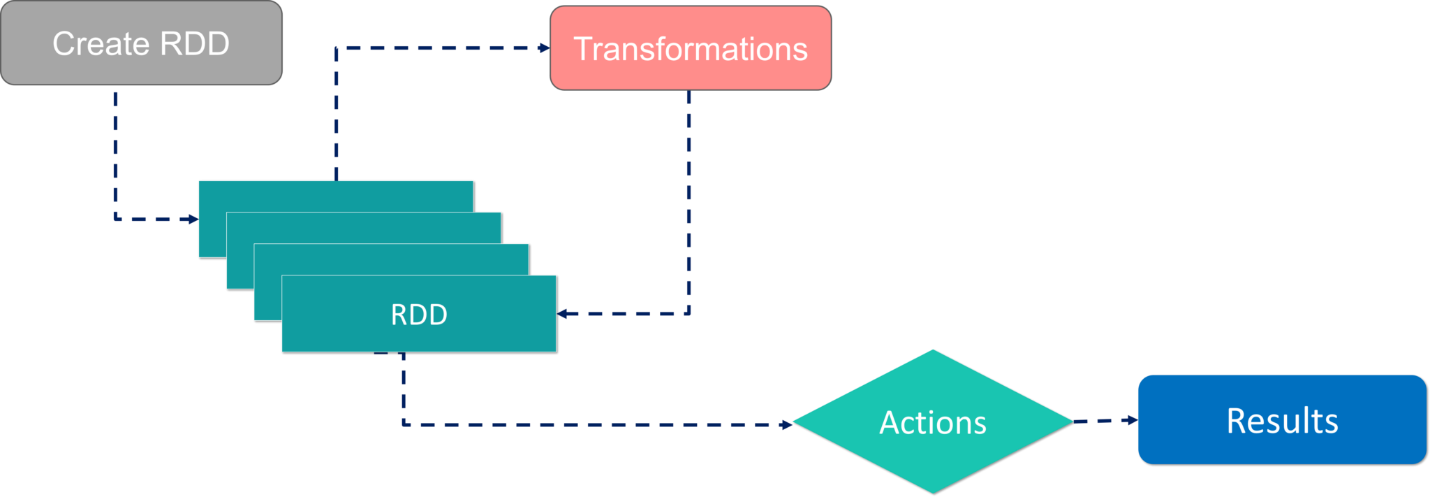
Moreover, once you create an RDD it becomes ***immutable***. By immutable mean, an object whose state cannot be modified after it is created, but they can surely be transformed.

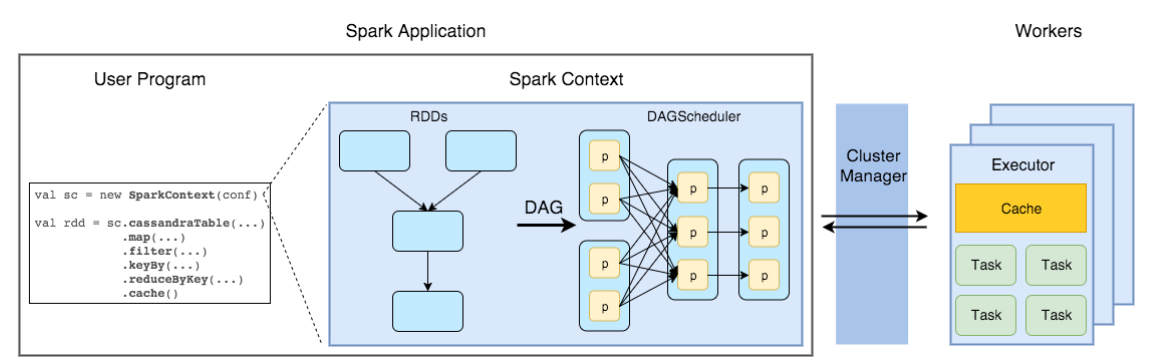
In the distributed environment, each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. Due to this, you can perform transformations or actions on the complete data parallelly. Also, you don’t have to worry about the distribution, because Spark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

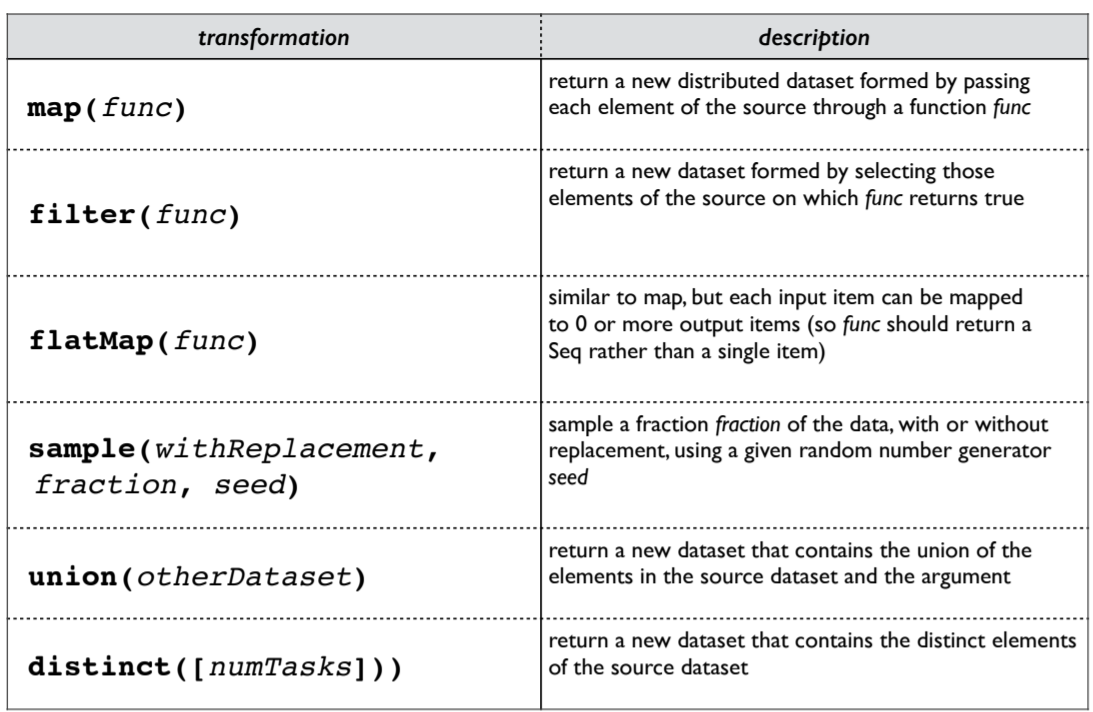
1. **Transformations:** They are the operations that are applied to create a new RDD.
2. **Actions:** They are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.

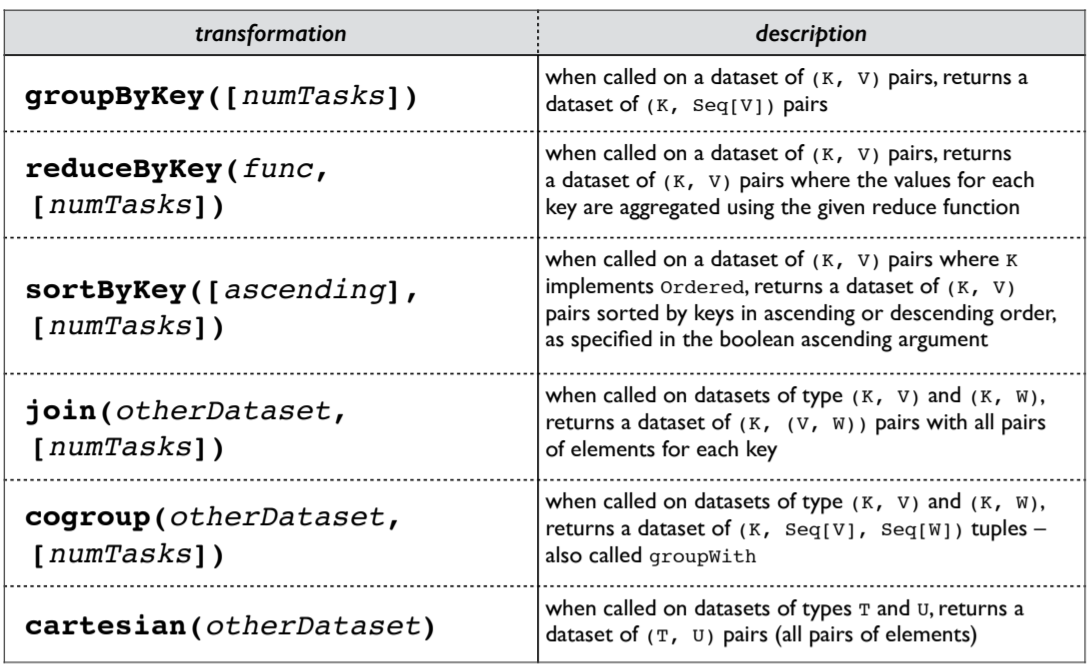




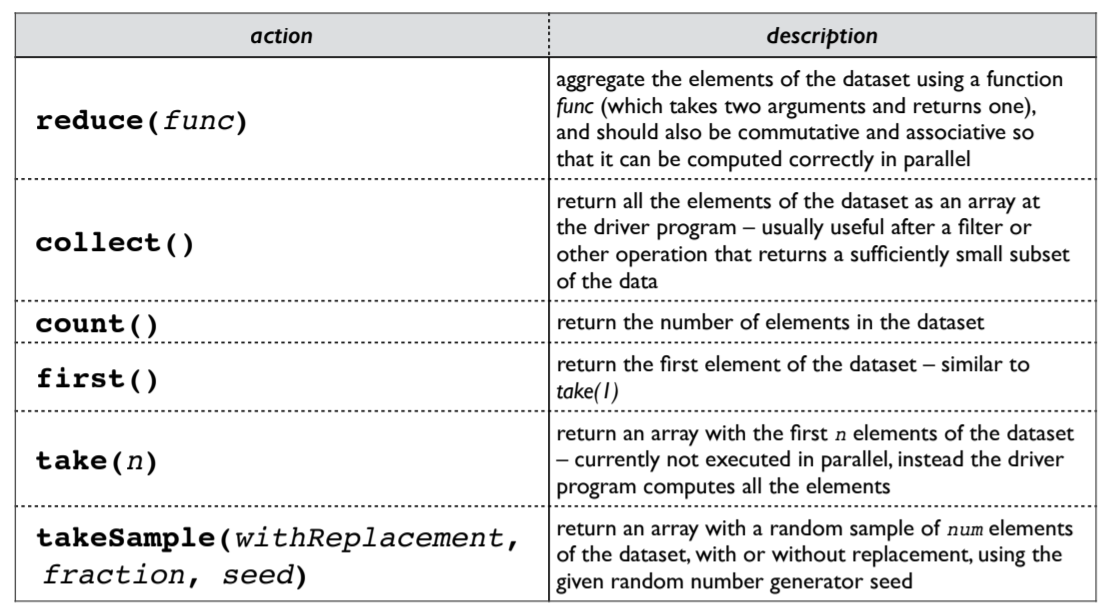
**Spark Essentials:**

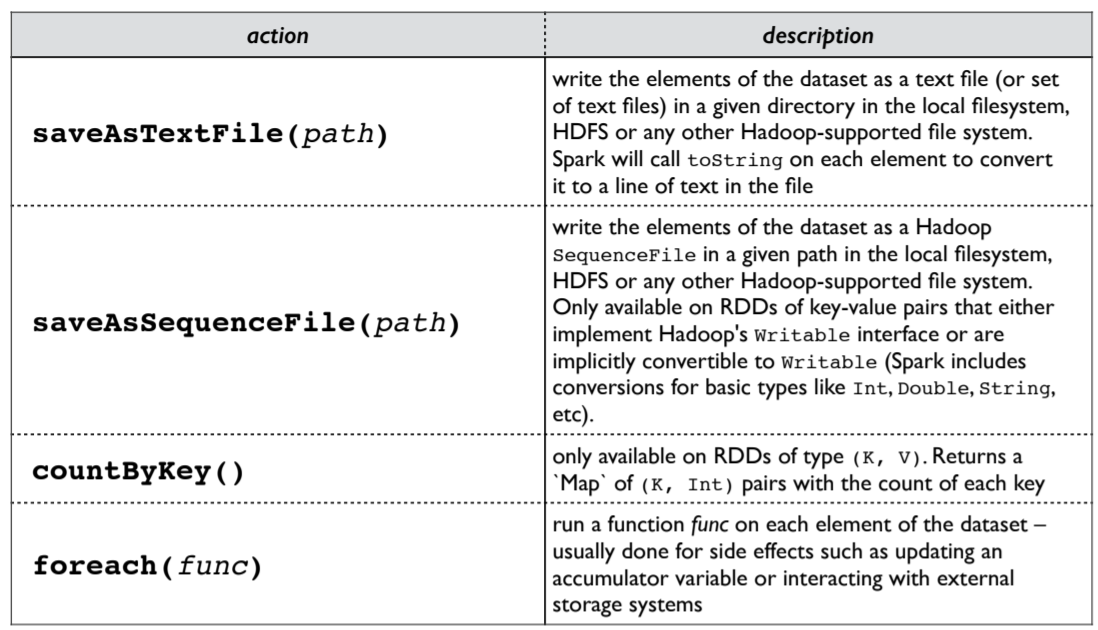
**Transformations**





**Actions**

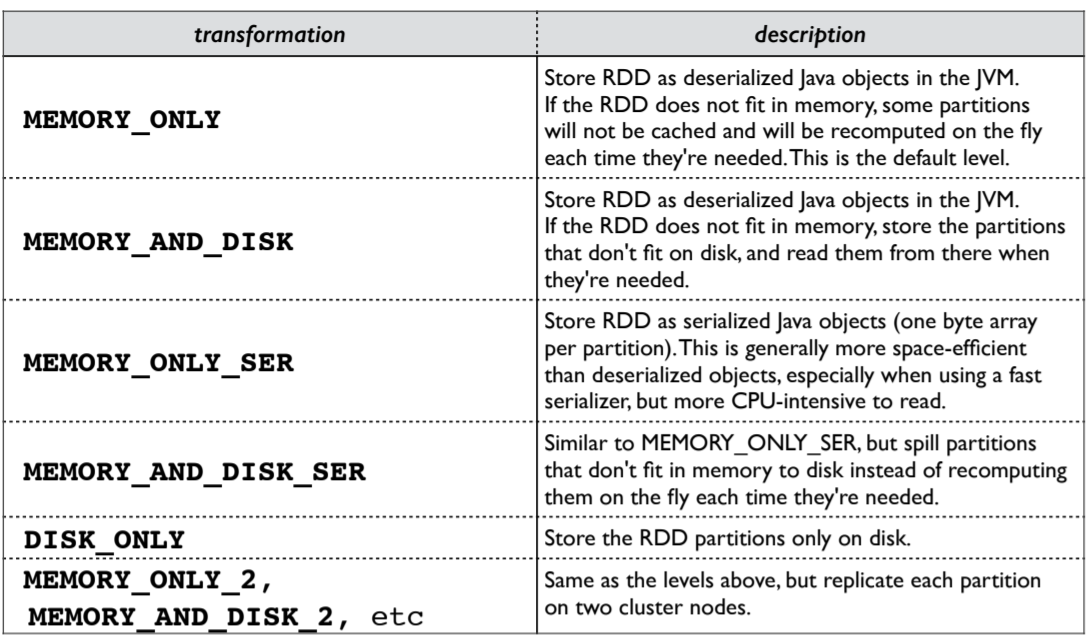




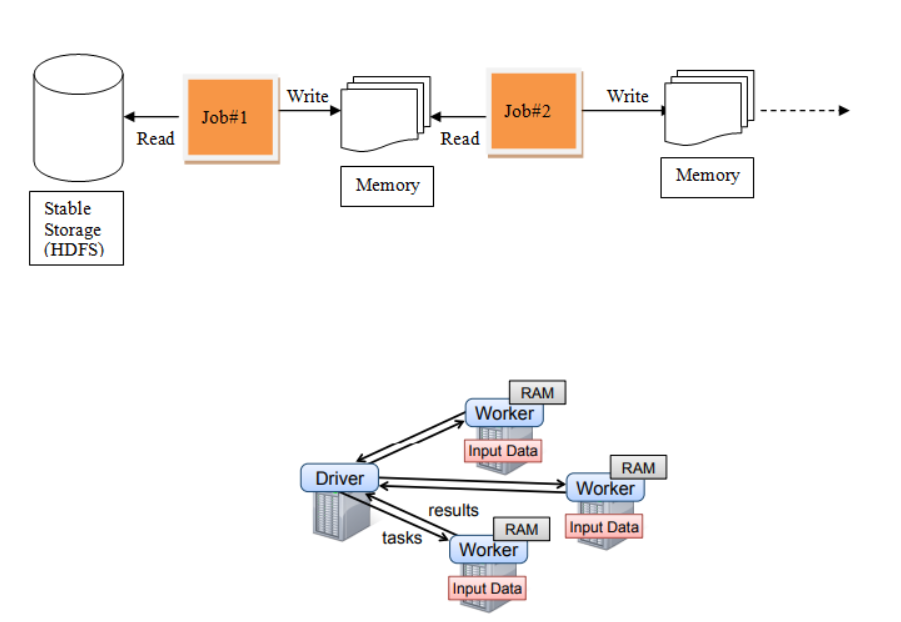
**Persistence:**

Spark can persist (or cache) a dataset in memory across operations.

Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it



**Spark Architecture:**



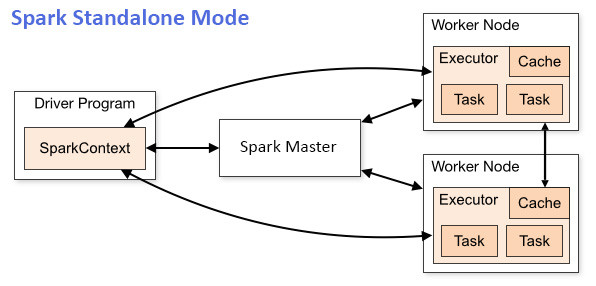
**Master/worker architecture**. Driver, Executors and Cluster manager are key component of spark.

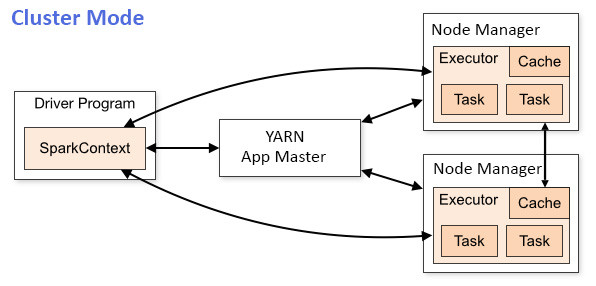
**Master**

It’s running instance of spark that connect with cluster manger to avail the resource to run driver and executors. So don’t confuse with physical master/slave machine. Here it’s a kind of daemon that continuously running with spark.

**Workers**

Worker is running spark instance where executor will be created to run several task throughout lifecycle spark application. Worker node is also a JVM process running with spark master.





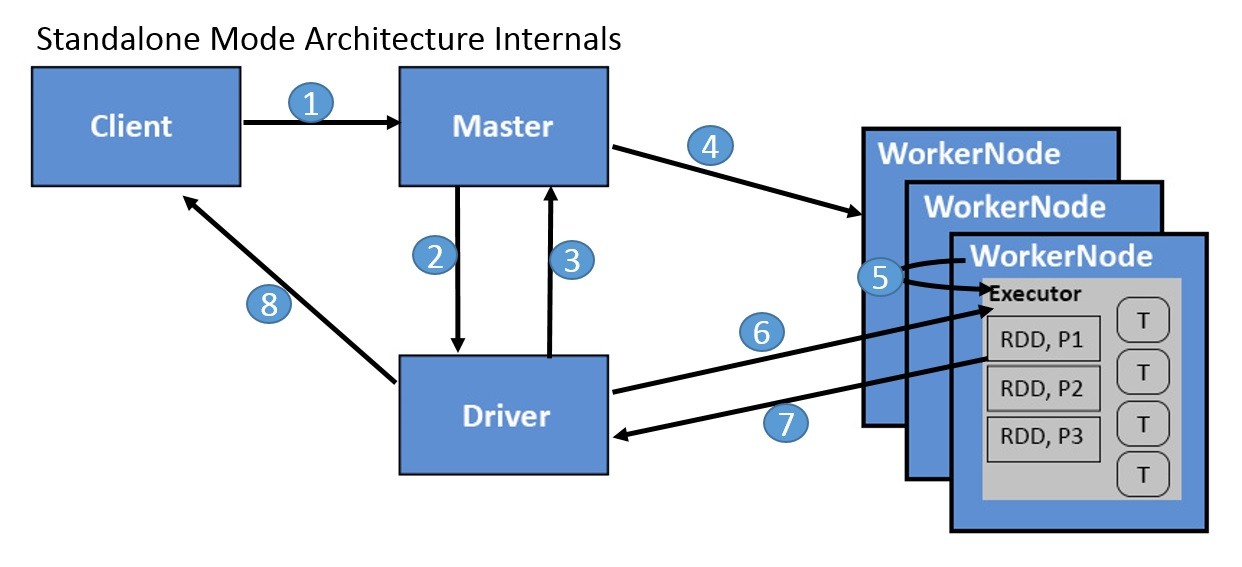
**Driver**

It is a JVM process that hosts SparkContext for a Spark application. It’s responsible to running executor process where task scheduler schedule tasks. It coordinates to workers and overall execution of tasks.

**Executors**

It’s JVM process that execute multiple tasks and it provide in-memory storage for RDD. An executor have multiple slots to run multiple tasks parallel.

**Spark Architecture on Standalone Mode**



**Spark Application Workflow in Standalone Mode**

1.     Client connect to master.

2.     Master start driver on one of node.

3.     Driver connect to master and request for Executors to run the tasks.

4.     Master connect to worker node and request to create executors.

5.     Each Worker node create one executor for each Application.

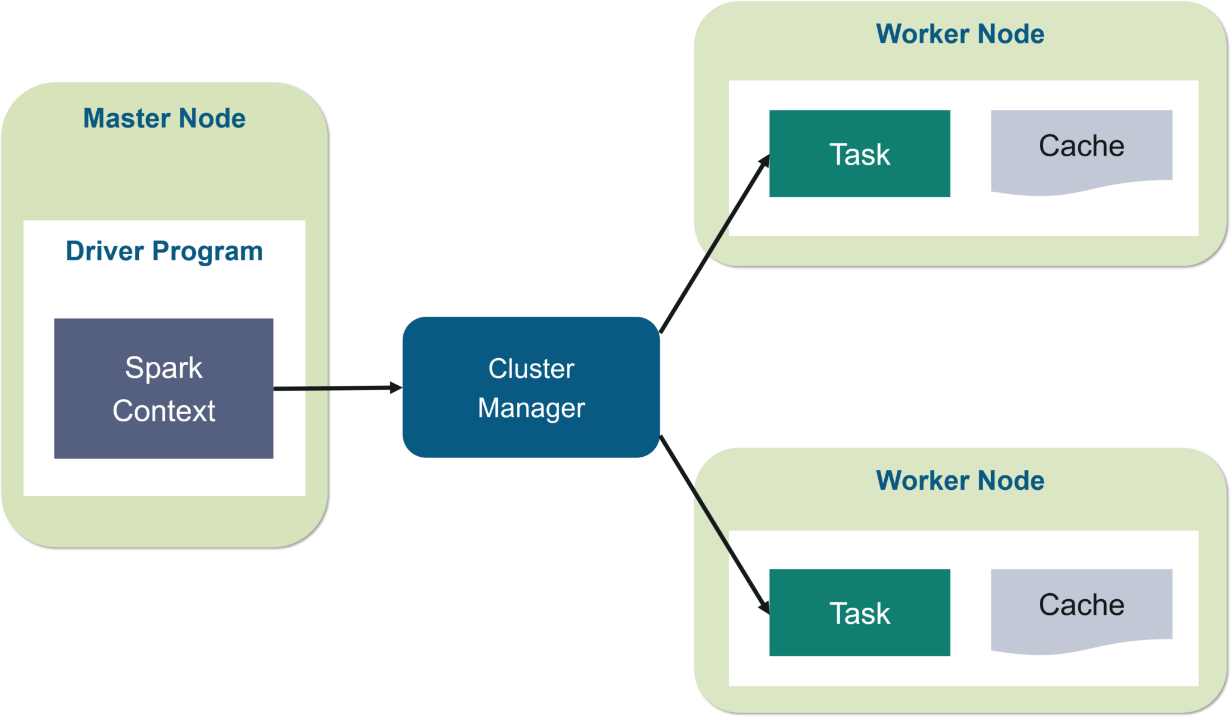
6.     Driver connect to executors and schedule tasks on it.

7.     Update the status of task to driver.

8.     Driver send application output to client.

**Working of Spark Architecture**

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, 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.

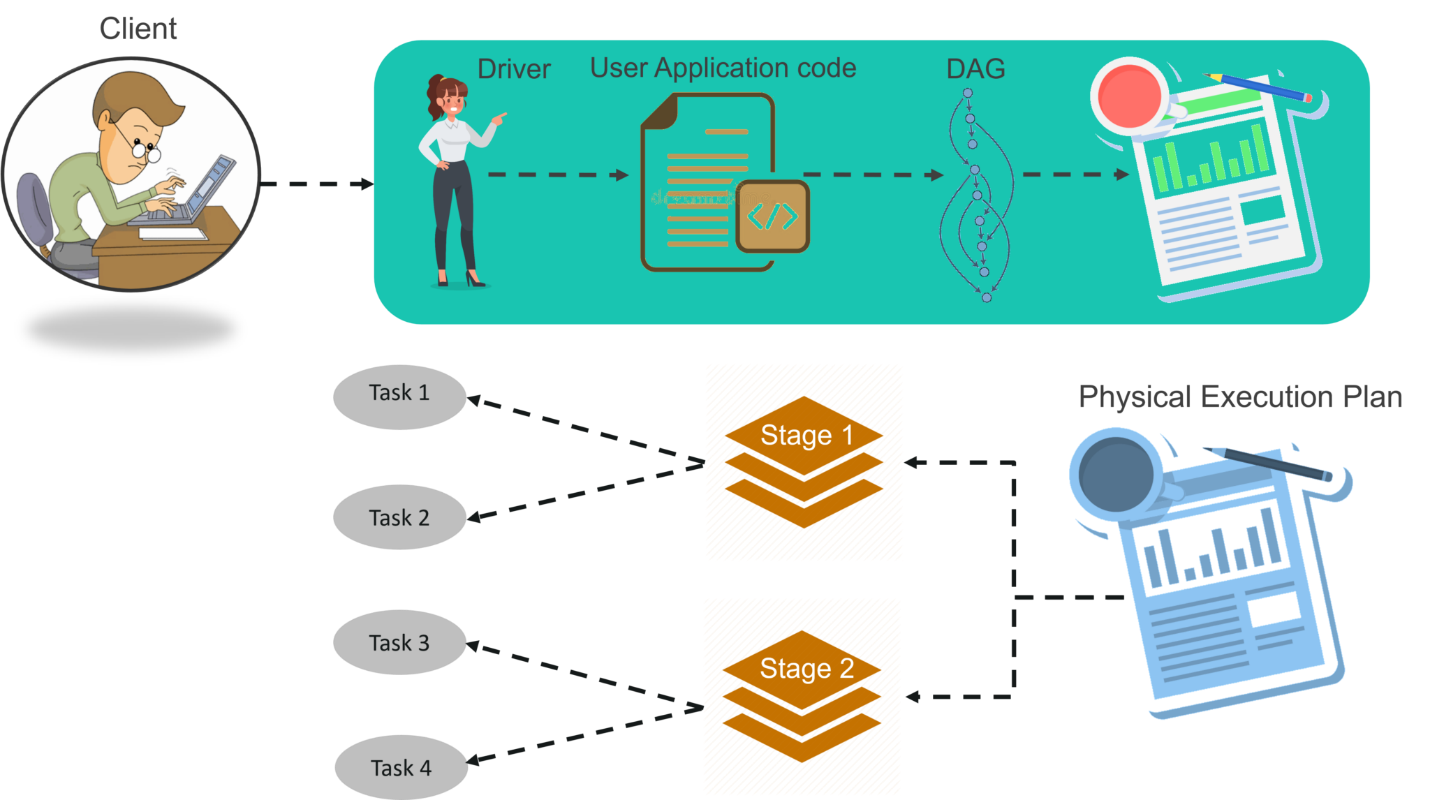


                                                         Fig: Spark Architecture Infographic

**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.

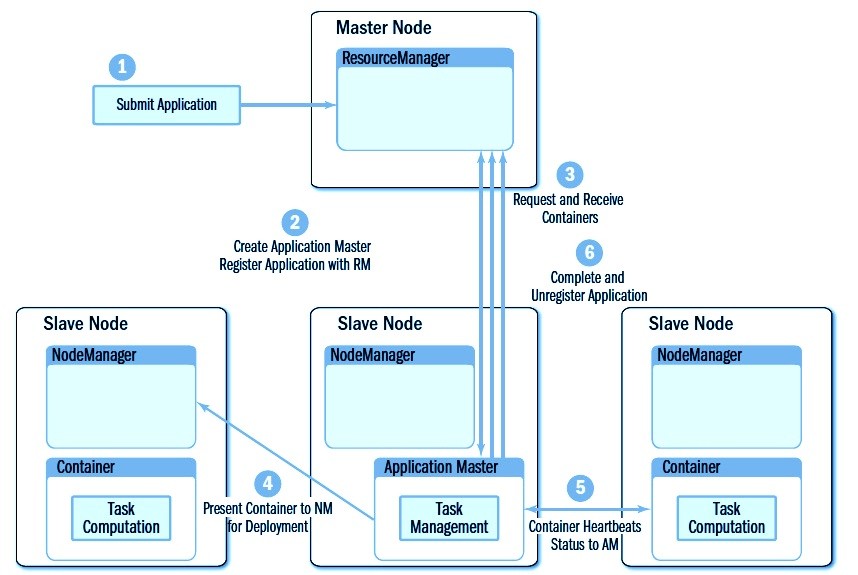
**STEP 2:** After that, it converts the logical graph called DAG into physical execution plan with many stages. After converting into a physical execution plan, it creates physical execution units called tasks under each stage. Then the tasks are bundled and sent to the cluster.

**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.

**YARN Architecture**



**Yarn Application Workflow**

1.     Client submits application lightweight request for an Application ID.

2.     If the request is successful, the Application Manager will respond with an Application ID. This Application ID will be used for the actual application submission to the cluster.

3.     The client submits the application to the Application Manager along with queue, dependencies, container launch commands, etc.

4.     The Application Manager is responsible for finding a Container on a Node Manager to start the Application Master.

5.     Once the Application Master has started, it will establish connection with the Resource Manager, specifically, a component called the Application Master Service. It will then retrieve cluster memory & CPU availability.

6.     The Application Master will send a request for Containers to run the application. In the case of MapReduce, it will ask for specific Node Managers because it wants to co-locate processing with data blocks in HDFS.

7.     The Application Master will receive leases on Containers per the Yarn Scheduler policy. It will also have to take care of graceful shutdown of Containers by releasing its lease(s) to the Resource Manager.

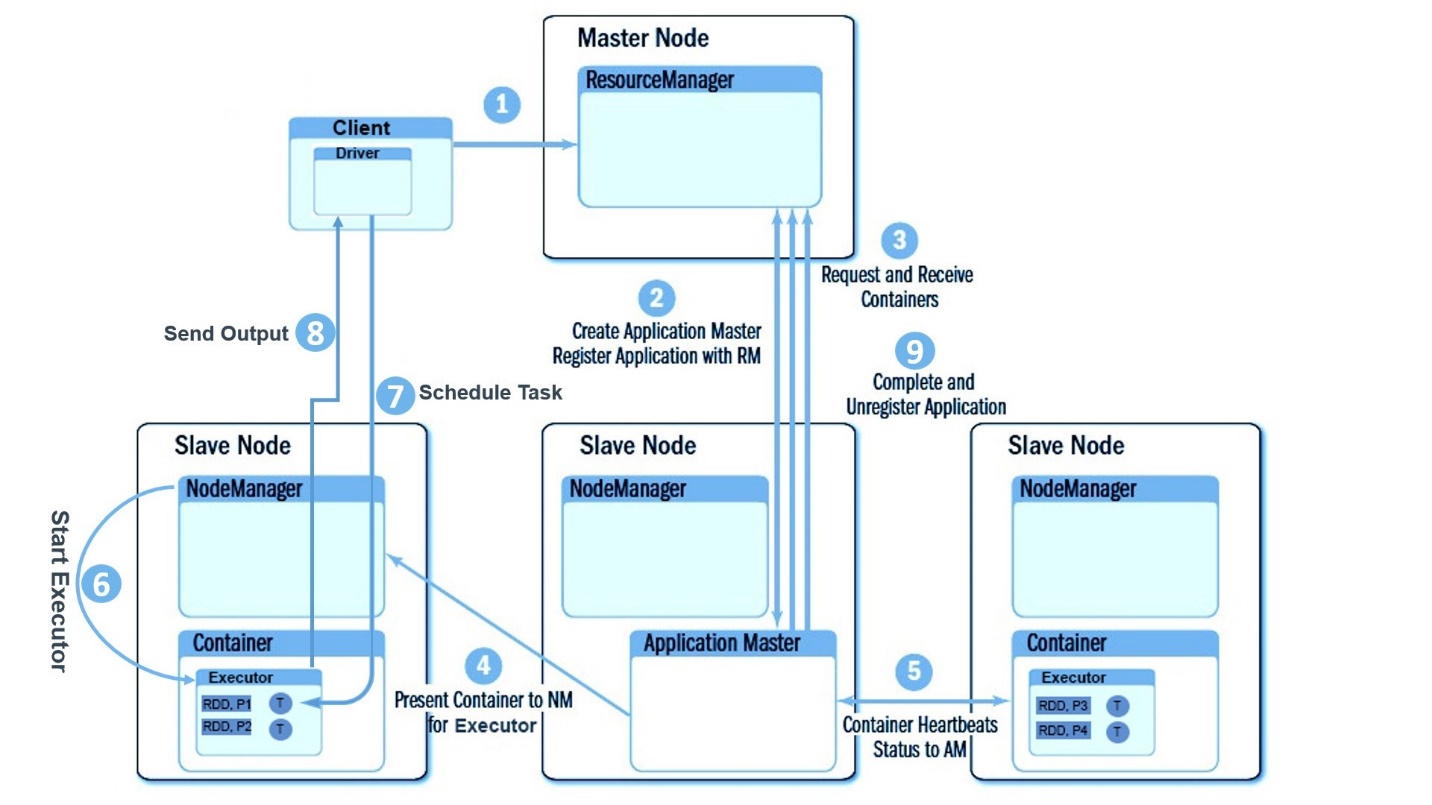
8.     Once the Application Master has determined that the application should end, it can optionally persist application logs to HDFS, and any other post-process activity before it self-terminates.

**Note:**

1.     Resource manager also have one scheduler and it decide where Application master should run and where container should run.

2.     Resource manager has Applications Master. If any app master stop running then Applications master decide whether app master should restart on same machine or different machine.

**Spark Architecture on Yarn Client Mode (YARN Client)**



**Spark Application Workflow in YARN Client mode**

1.     In Yarn Client mode Driver run on client system that may be your laptop or any machine.

Here there is no spark master and no worker node. Client send job to Applications manager running on master node.

2.     Applications manager request to resource manager for resource to run Application master.

3.     Once Application manager get resource on one of slave node then create container and run Application Master or App master.

4.     Once Application master start running driver connect to Application master and request for N no of executors. Application master again connect to resource manager and request for container to run executors. Here resource manager is doing job of spark master in Standalone mode.

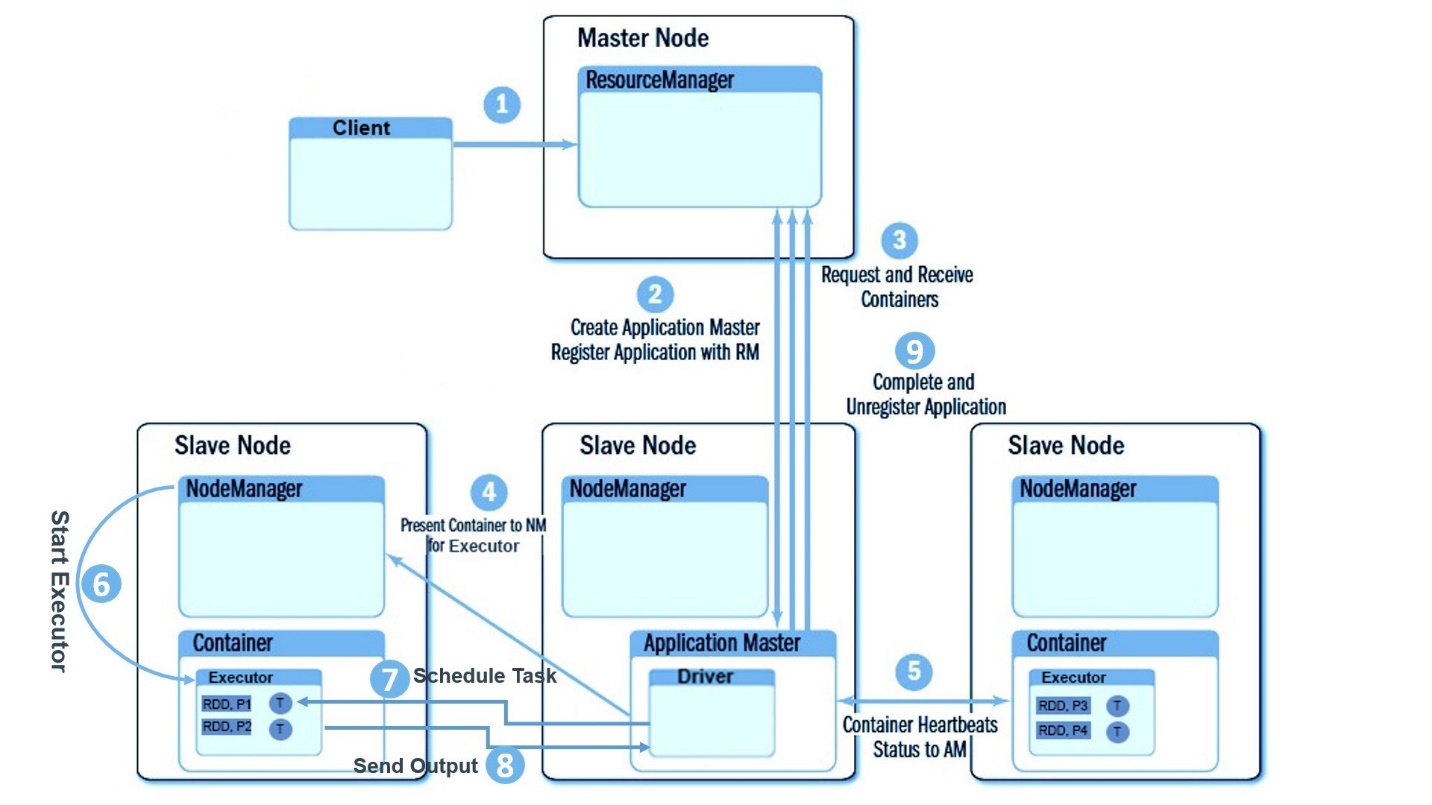
5.     Once it get information about nodes that have resource to create container. It connect to those node managers and request to create container.

6.     Node Manager create container to run executors. Once container is ready application master run executor in it.

7.     Once executors start running driver schedule tasks in multiple slot for each executors.

8.     Once all tasks completed it send output to driver.

**Spark Architecture on Yarn Cluster Mode (YARN Cluster)**



**Spark Application Workflow in YARN Cluster mode**

Cluster mode work flow is almost similar to Client mode but there is one difference that here driver run inside Application master and because of that client can disconnect with cluster after submitting the job. Job will still running and you will get output when it is finished which is not possible in client mode. In client mode if you are disconnecting client to cluster job will stop running and you not get any output

**YARN Cluster vs. YARN Client vs. Spark Standalone**

