

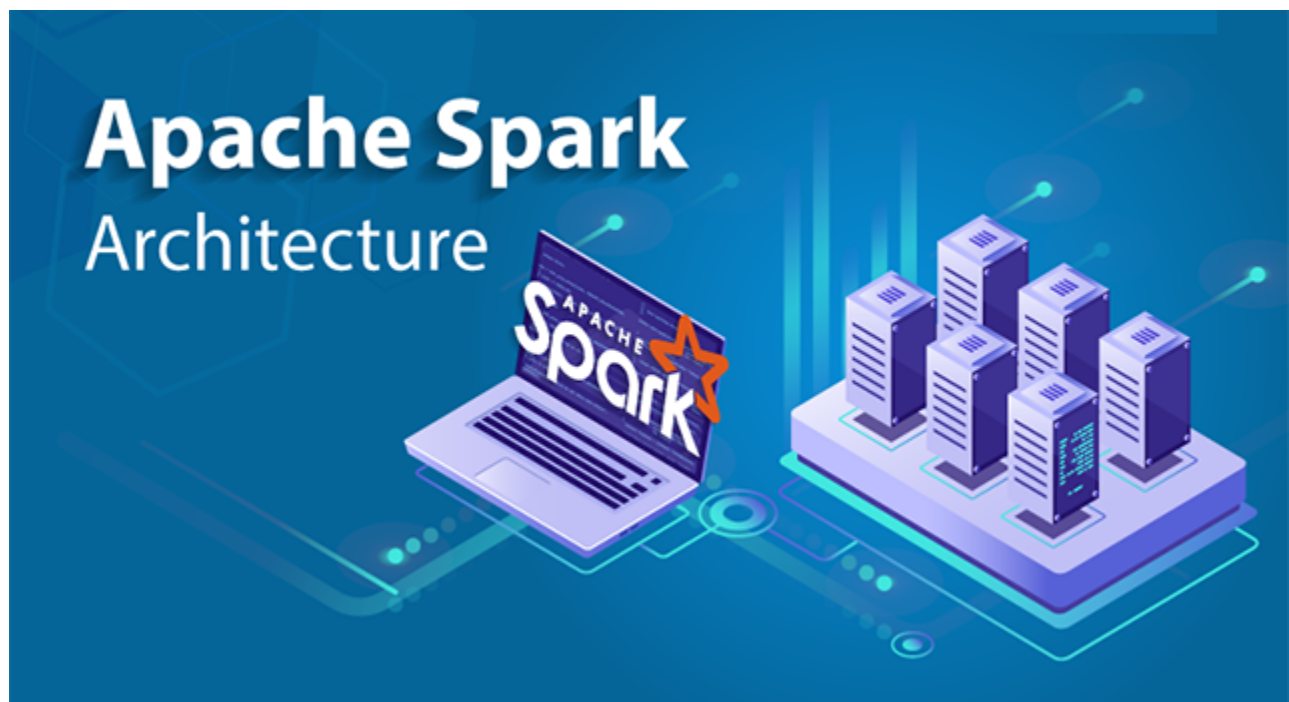
Apache Spark Architecture - Distributed System Architecture Explained



Shubham Sinha

[Follow](#)

Sep 28, 2018 · 10 min read



Apache Spark Architecture — Edureka

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. In this article, I will give you a brief insight on Spark Architecture and the fundamentals that underlie Spark Architecture.

In this Spark Architecture article, I will be covering the following topics:

- Spark & its Features
- Spark Architecture Overview

- Spark Eco-System
- Resilient Distributed Datasets (RDDs)
- Working of Spark Architecture
- Example using Scala in Spark Shell

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.**

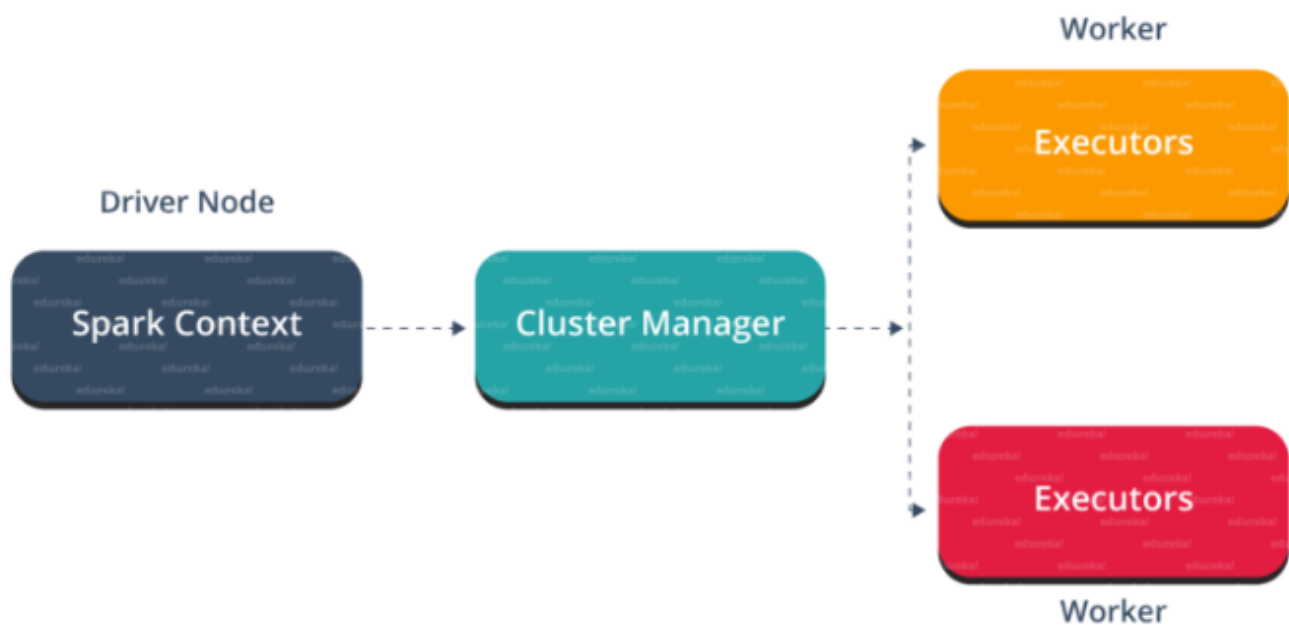
Polyglot

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)*



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.

Let me first explain what is Spark Eco-System.

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.



Spark Core

Spark Core is the base engine for large-scale parallel and distributed data processing. Further, additional libraries that 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.

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.

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.

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).

MLlib (Machine Learning)

MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

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.

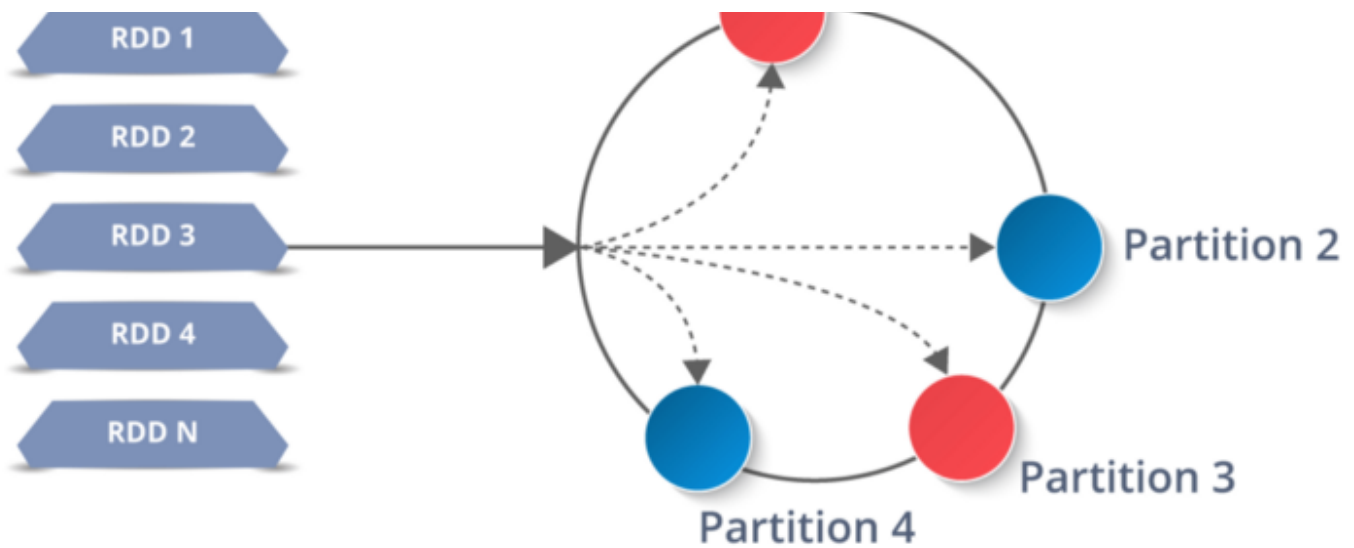
As you can see, Spark comes packed with high-level libraries, including support for R, SQL, Python, Scala, Java etc. These standard libraries increase the seamless integrations in a complex workflow. Over this, it also allows various sets of services to integrate with it like MLlib, GraphX, SQL + Data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

Now, let's discuss the fundamental Data Structure of Spark, i.e. RDD.

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

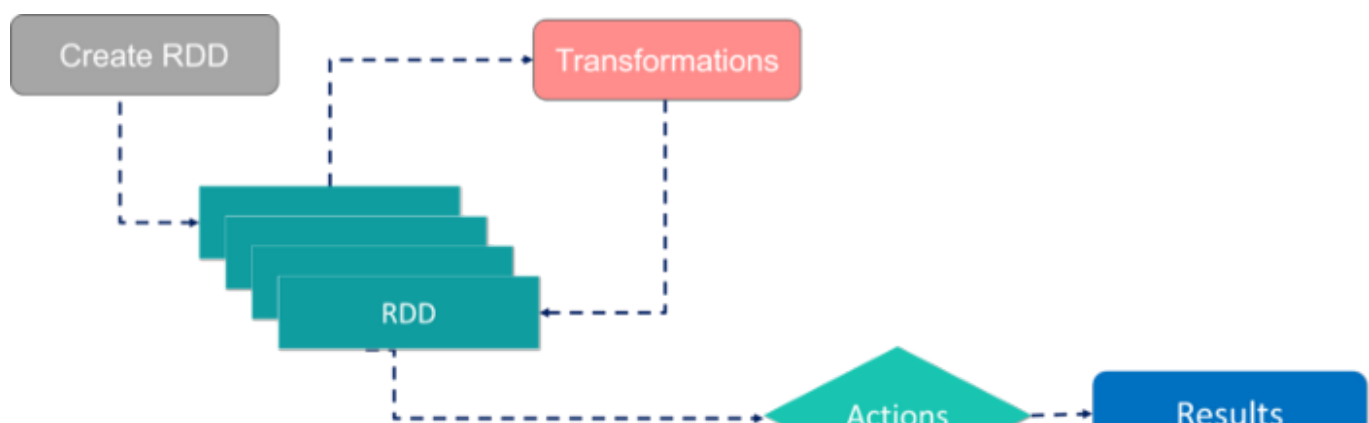


It is a layer of abstracted data over the distributed collection. It is immutable in nature and follows *lazy transformations*.

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:

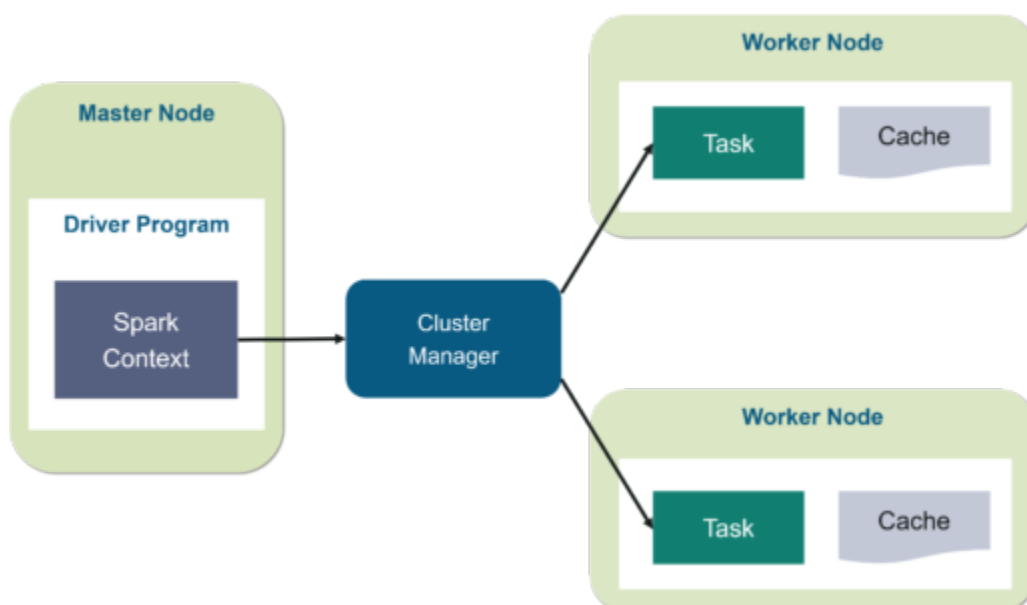
1. **Transformations:** They are the operations that are applied to create a new RDD.
2. **Actions:** They are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.

I hope you got a thorough understanding of RDD concepts. Now let's move further and see the working of Spark Architecture.

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.

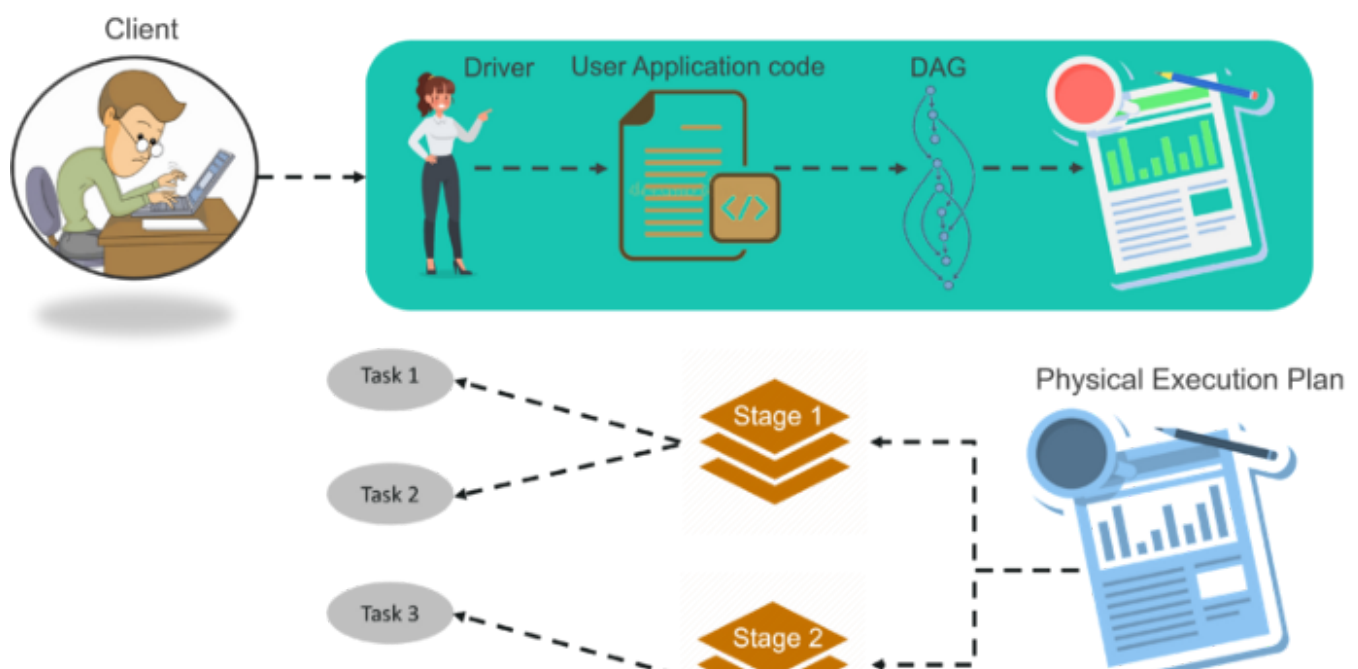
Worker nodes are the slave nodes whose job is to basically execute the tasks. These tasks are then executed on the partitioned RDDs in the worker node and hence returns back the result to the Spark Context.

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:

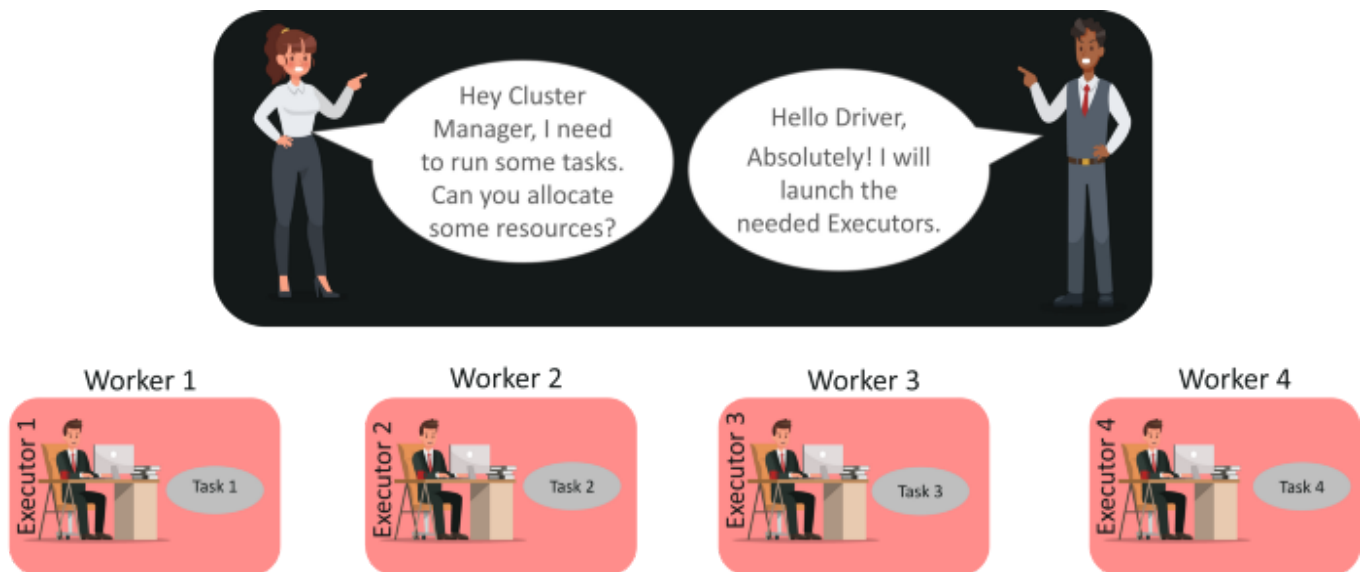
The client submits spark user application code. When application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically *directed acyclic graph* called **DAG**. At this stage, it also performs optimizations such as pipelining transformations.

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.



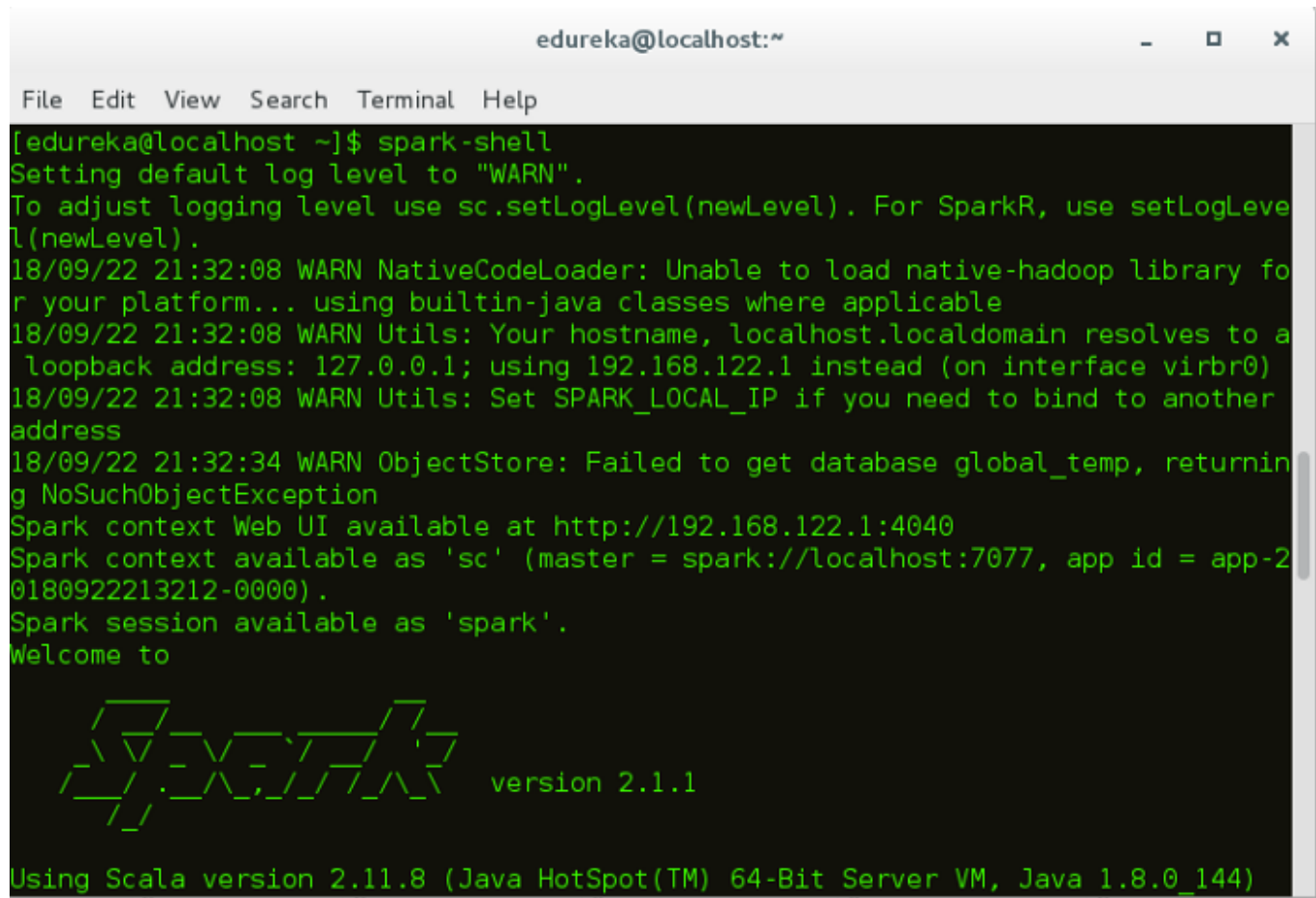
STEP 4:

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.

This was all about Spark Architecture. Now, let's get a hand's on the working of a Spark shell.

Example using Scala in Spark shell

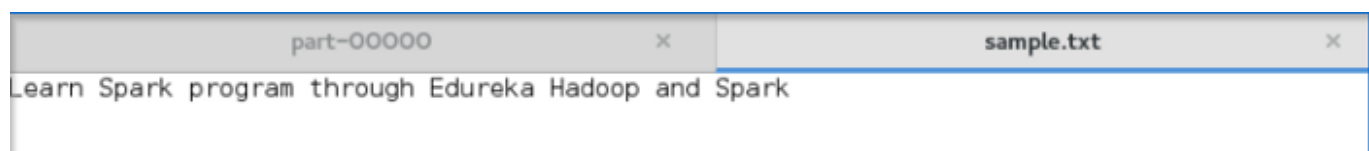
At first, let's start the Spark shell by assuming that Hadoop and Spark daemons are up and running. **Web UI** port for Spark is **localhost:4040**.



```
edureka@localhost:~  
File Edit View Search Terminal Help  
[edureka@localhost ~]$ spark-shell  
Setting default log level to "WARN".  
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).  
18/09/22 21:32:08 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  
18/09/22 21:32:08 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.122.1 instead (on interface virbr0)  
18/09/22 21:32:08 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address  
18/09/22 21:32:34 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException  
Spark context Web UI available at http://192.168.122.1:4040  
Spark context available as 'sc' (master = spark://localhost:7077, app id = app-20180922213212-0000).  
Spark session available as 'spark'.  
Welcome to  
  
          version 2.1.1  
  
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_144)
```

Once you have started the Spark shell, now let's see how to execute a word count example:

1. In this case, I have created a simple text file and stored it in the hdfs directory. You can also use other large data files as well.



2. Once the spark shell has started, let's create an RDD. For this, you have to specify the input file path and apply the transformation **flatMap()**. Below code illustrates the same:

```
scala> var map =
sc.textFile("hdfs://localhost:9000/Example/sample.txt").flatMap(line
=> line.split(" ")).map(word => (word,1));
```

3. On executing this code, an RDD will be created as shown in the figure.

```
scala> var map = sc.textFile("hdfs://localhost:9000/word/sample.txt").flatMap(line => line.split(" ")).map(word => (word,1));
map: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:24
```

4. After that, you need to apply the action **reduceByKey()** to the created RDD.

```
scala> var counts = map.reduceByKey(_+_);
```

After applying action, execution starts as shown below.

```
scala> var counts = map.reduceByKey(_+_);
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:26
```

5. Next step is to save the output in a text file and specify the path to store the output.

```
scala> counts.saveAsTextFile("hdfs://localhost:9000/word/output1.txt");
```

6. After specifying the output path, go to the *hdfs web browser localhost:50040*. Here you can see the output text in the ‘part’ file as shown below.

Browse Directory							
<input type="text" value="/word/output1.txt"/>							<input data-bbox="1404 1606 1469 1648" type="button" value="Go!"/>
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	edureka	supergroup	0 B	9/18/2018, 10:54:36 AM	1	128 MB	_SUCCESS
-rw-r--r--	edureka	supergroup	75 B	9/18/2018, 10:54:35 AM	1	128 MB	part-00000

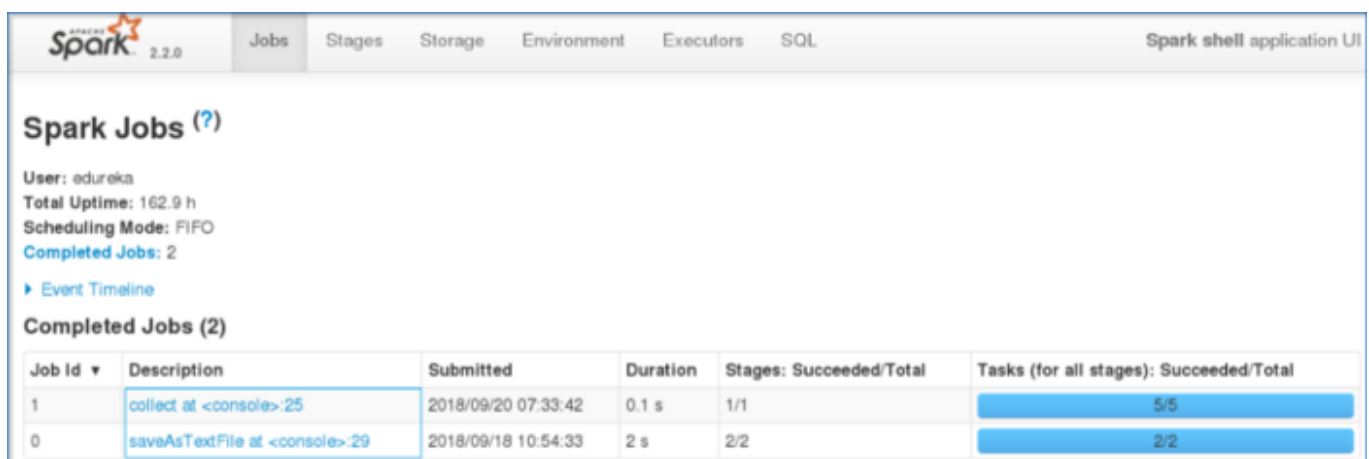
7. Below figure shows the output text present in the ‘part’ file.



```
Spark,2)
Learn,1)
through,1)
and,1)
Edureka,1)
program,1)
Hadoop,1)
```

I hope that you have understood how to create a Spark Application and arrive at the output.

Now, let me take you through the web UI of Spark to understand the DAG visualizations and partitions of the executed task.



Spark Jobs (?)

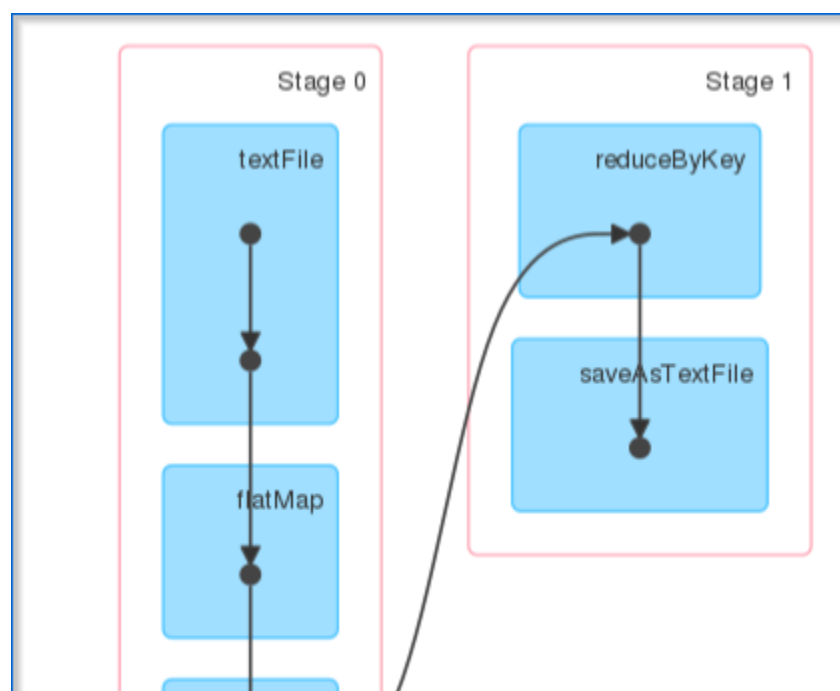
User: edureka
Total Uptime: 162.9 h
Scheduling Mode: FIFO
Completed Jobs: 2

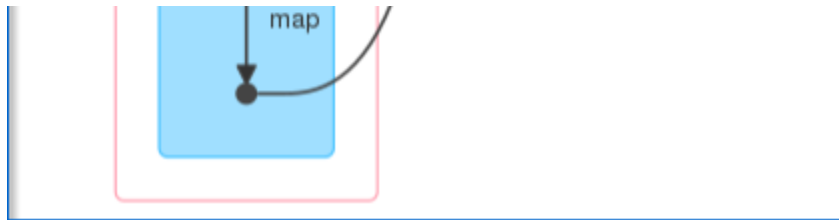
▶ Event Timeline

Completed Jobs (2)

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <console>:25	2018/09/20 07:33:42	0.1 s	1/1	5/5
0	saveAsTextFile at <console>:29	2018/09/18 10:54:33	2 s	2/2	2/2

- On clicking the task that you have submitted, you can view the Directed Acyclic Graph (DAG) of the completed job.





- Also, you can view the summary metrics of the executed task like — time taken to execute the task, job ID, completed stages, host IP Address etc.

Now, let's understand about partitions and parallelism in RDDs.

- A **partition** is a *logical chunk* of a *large distributed data set*.
- By default, Spark tries to *read data into an RDD* from the *nodes* that are *close to it*.

Now, let's see how to execute a parallel task in the shell.

```
Welcome to
Spark version 2.1.1

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

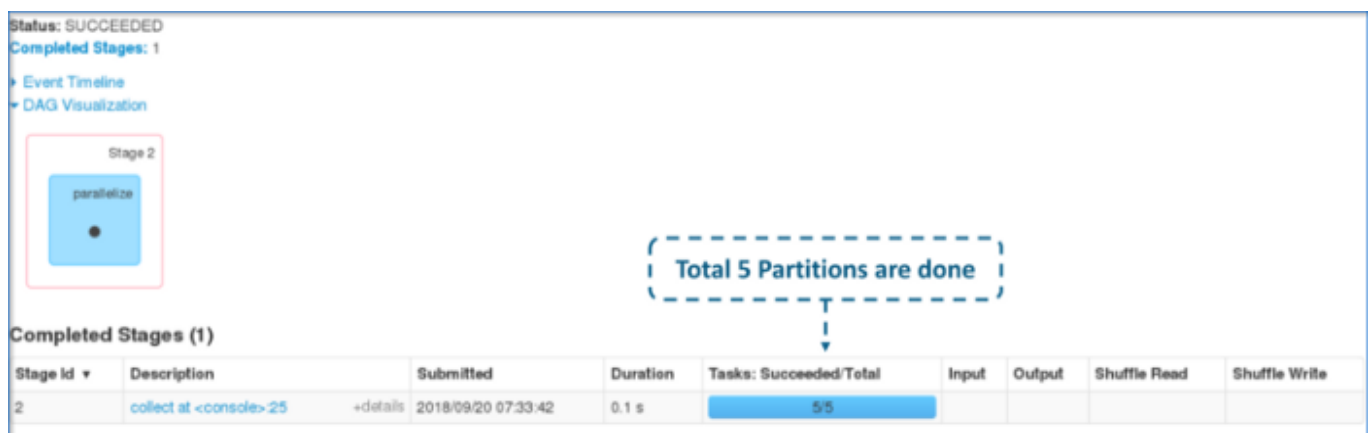
scala> sc.parallelize(1 to 100, 5).collect()

res4: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

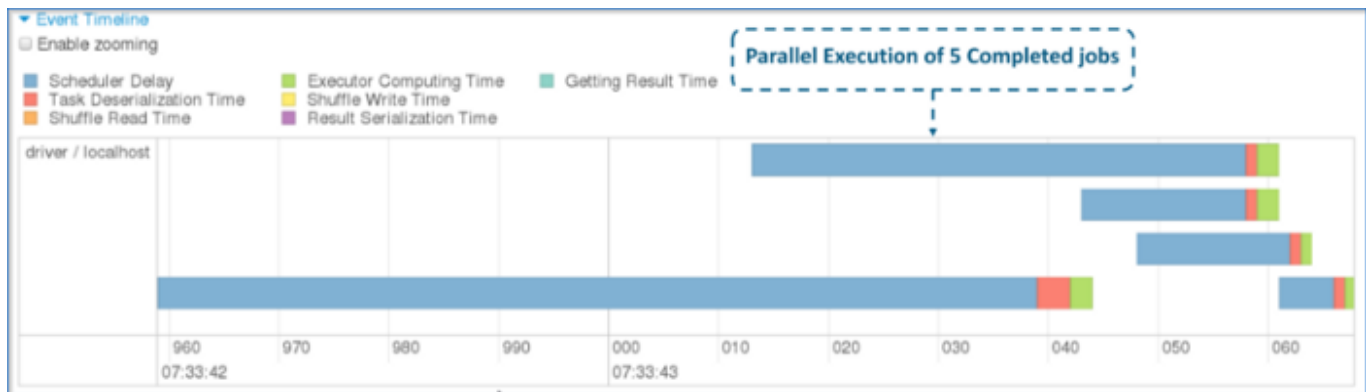
`sc.parallelize(1 to 100, 5).collect()`

5 here, represents the number of partitions the RDD is split into which in turn runs them parallelly.

- Below figure shows the total number of partitions on the created RDD.



- Now, let me show you how parallel execution of 5 different tasks appears.



This brings us to the end of the blog on Apache Spark Architecture. I hope this blog was informative and added value to your knowledge.

So this is it! I hope this blog was informative and added value to your knowledge. If you wish to check out more articles on the market's most trending technologies like Artificial Intelligence, DevOps, Ethical Hacking, then you can refer to Edureka's official site.

Do look out for other articles in this series which will explain the various other aspects of Spark.

1.[Apache Spark Tutorial](#)

2.[Spark Streaming Tutorial](#)

3.[Spark MLlib](#)

4.[Spark SQL Tutorial](#)

5.[Spark GraphX Tutorial](#)

6.[Spark Java Tutorial](#)

Originally published at www.edureka.co on September 28, 2018.

[Hadoop](#)[Apache Spark](#)[Scala](#)[Spark](#)[Rdd](#)[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

