SPARK:

# Components

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program).

Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers (either Spark’s own standalone cluster manager, Mesos, YARN or Kubernetes), which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run.



There are several useful things to note about this architecture:

1. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.
2. Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN/Kubernetes).
3. The driver program must listen for and accept incoming connections from its executors throughout its lifetime (e.g., see [spark.driver.port in the network config section](https://spark.apache.org/docs/latest/configuration.html" \l "networking)). As such, the driver program must be network addressable from the worker nodes.
4. Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network. If you’d like to send requests to the cluster remotely, it’s better to open an RPC to the driver and have it submit operations from nearby than to run a driver far away from the worker nodes.

# Cluster Manager Types

The system currently supports several cluster managers:

* [Standalone](https://spark.apache.org/docs/latest/spark-standalone.html) – a simple cluster manager included with Spark that makes it easy to set up a cluster.
* [Apache Mesos](https://spark.apache.org/docs/latest/running-on-mesos.html) – a general cluster manager that can also run Hadoop MapReduce and service applications. (Deprecated)
* [Hadoop YARN](https://spark.apache.org/docs/latest/running-on-yarn.html) – the resource manager in Hadoop 2.
* [Kubernetes](https://spark.apache.org/docs/latest/running-on-kubernetes.html) – an open-source system for automating deployment, scaling, and management of containerized applications.

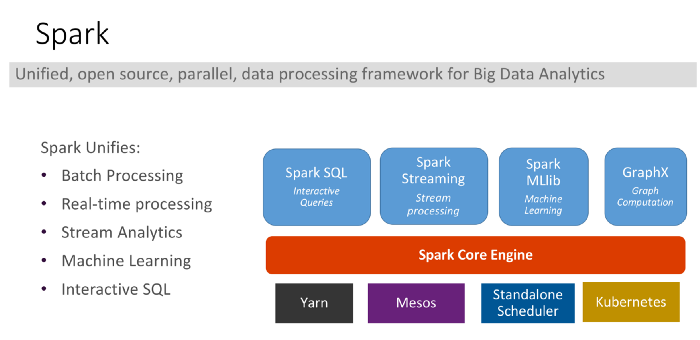
## Overview

* Learn about the Spark Architecture
* Learn about different execution modes

## Introduction

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters. It is the most actively developed open-source engine for this task, making it a standard tool for any developer or data scientist interested in big data.

Spark supports multiple widely-used programming languages (Python, Java, Scala, and R), includes libraries for diverse tasks ranging from SQL to streaming and machine learning, and Spark runs anywhere from a laptop to a cluster of thousands of servers. This makes it an easy system to start with and scale-up to big data processing or an incredibly large scale.



With more than 500 contributors from across 200 organizations responsible for code and a user base of 225,000+ members, Apache Spark has become mainstream and most in-demand big data framework across all major industries. E-commerce companies like Alibaba, social networking companies like Tencent, and Chinese search engine Baidu, all run apache spark operations at scale.

This article is a single-stop resource that gives the Spark architecture overview with the help of a spark architecture diagram.

## Table of contents

* The Architecture of a Spark Application
  + The Spark driver
  + The Spark Executors
  + The Cluster manager
* Cluster Manager types
* Execution Modes
  + Cluster Mode
  + Client Mode
  + Local Mode

## The Architecture of a Spark Application

Below are the high-level components of the architecture of the Apache Spark application:

#### **The Spark driver**

The driver is the process “in the driver seat” of your Spark Application. It is the controller of the execution of a Spark Application and maintains all of the states of the Spark cluster (the state and tasks of the executors). It must interface with the cluster manager in order to actually get physical resources and launch executors.

At the end of the day, this is just a process on a physical machine that is responsible for maintaining the state of the application running on the cluster.

#### **The Spark executors**

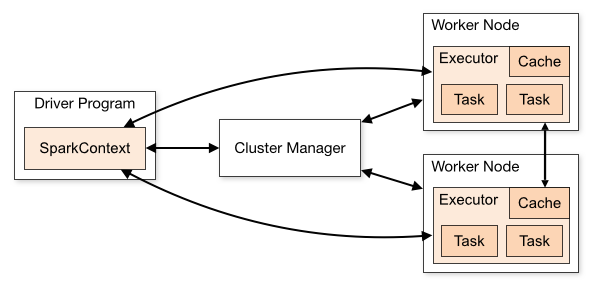
Spark executors are the processes that perform the tasks assigned by the Spark driver. Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state (success or failure) and results. Each Spark Application has its own separate executor processes.

#### **The cluster manager**

The Spark Driver and Executors do not exist in a void, and this is where the cluster manager comes in. The cluster manager is responsible for maintaining a cluster of machines that will run your Spark Application(s). Somewhat confusingly, a cluster manager will have its own “driver” (sometimes called master) and “worker” abstractions.

The core difference is that these are tied to physical machines rather than processes (as they are in Spark). The machine on the left of the illustration is the Cluster Manager Driver Node. The circles represent daemon processes running on and managing each of the individual worker nodes. There is no Spark Application running as of yet—these are just the processes from the cluster manager.

When the time comes to actually run a Spark Application, we request resources from the cluster manager to run it. Depending on how our application is configured, this can include a place to run the Spark driver or might be just resources for the executors for our Spark Application. Over the course of Spark Application execution, the cluster manager will be responsible for managing the underlying machines that our application is running on.



#### **There are several useful things to note about this architecture:**

1. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs).  
   However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.
2. Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN).
3. The driver program must listen for and accept incoming connections from its executors throughout its lifetime (e.g., see [spark.driver.port in the network config section](https://spark.apache.org/docs/latest/configuration.html" \l "networking" \t "_blank)). As such, the driver program must be network addressable from the worker nodes.
4. Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network. If you’d like to send requests to the cluster remotely, it’s better to open an RPC to the driver and have it submit operations from nearby than to run a driver far away from the worker nodes.

## Cluster Manager Types

The system currently supports several cluster managers:

* [**Standalone**](https://spark.apache.org/docs/latest/spark-standalone.html)– a simple cluster manager included with Spark that makes it easy to set up a cluster.
* [**Apache Mesos**](https://spark.apache.org/docs/latest/running-on-mesos.html)– a general cluster manager that can also run Hadoop MapReduce and service applications.
* [**Hadoop YARN**](https://spark.apache.org/docs/latest/running-on-yarn.html)– the resource manager in Hadoop 2.
* [**Kubernetes**](https://spark.apache.org/docs/latest/running-on-kubernetes.html)– an open-source system for automating deployment, scaling, and management of containerized applications.

A third-party project (not supported by the Spark project) exists to add support for [Nomad](https://github.com/hashicorp/nomad-spark) as a cluster manager.

## Execution Modes

An execution mode gives you the power to determine where the aforementioned resources are physically located when you go running your application. You have three modes to choose from:

1. Cluster mode
2. Client mode
3. Local mode

#### **Cluster mode**

Cluster mode is probably the most common way of running Spark Applications. In cluster mode, a user submits a pre-compiled JAR, Python script, or R script to a cluster manager. The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes. This means that the cluster manager is responsible for maintaining all Spark Application– related processes.

#### **Client mode**

Client mode is nearly the same as cluster mode except that the Spark driver remains on the client machine that submitted the application. This means that the client machine is responsible for maintaining the Spark driver process, and the cluster manager maintains the executor processes. These machines are commonly referred to as gateway machines or edge nodes.

#### **Local mode**

Local mode is a significant departure from the previous two modes: it runs the entire Spark Application on a single machine. It achieves parallelism through threads on that single machine. This is a common way to learn Spark, to test your applications, or experiment iteratively with local development.

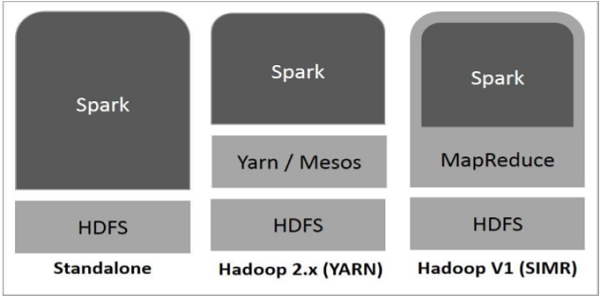
However, we do not recommend using local mode for running production applications.

## Conclusion

To sum up, Spark helps us break down the intensive and high-computational jobs into smaller, more concise tasks which are then executed by the worker nodes. It also achieves the processing of real-time or archived data using its basic architecture.

Spark Built on Hadoop

The following diagram shows three ways of how Spark can be built with Hadoop components.

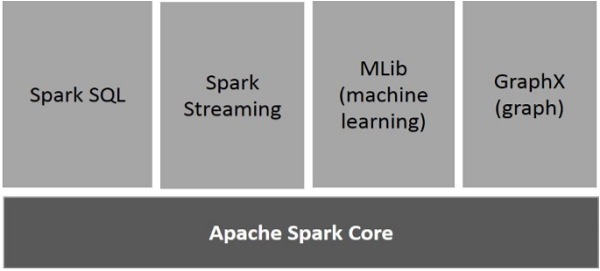


There are three ways of Spark deployment as explained below.

* **Standalone** − Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.
* **Hadoop Yarn** − Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.
* **Spark in MapReduce (SIMR)** − Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

Components of Spark

The following illustration depicts the different components of Spark.



Apache Spark Core

Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL

Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

Spark Streaming

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

MLlib (Machine Learning Library)

MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of **Apache Mahout** (before Mahout gained a Spark interface).

GraphX

GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

## Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs − **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

## Data Sharing is Slow in MapReduce

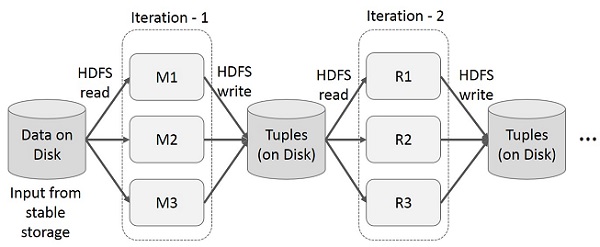
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex − between two MapReduce jobs) is to write it to an external stable storage system (Ex − HDFS). Although this framework provides numerous abstractions for accessing a cluster’s computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

## Iterative Operations on MapReduce

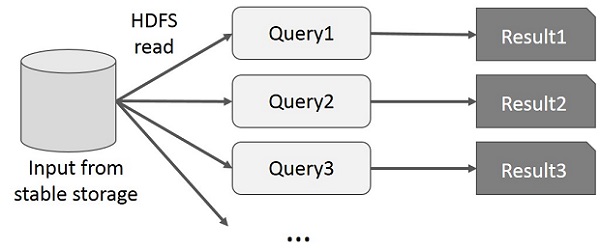
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



## Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



## Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

## Iterative Operations on Spark RDD

The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

**Note** − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



## Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

SPARK and ELT:

In this post, I am going to discuss Apache Spark and how you can create simple but robust ETL pipelines in it. You will learn how Spark provides APIs to transform different data format into Data frames and SQL for analysis purpose and how one data source could be transformed into another without any hassle.

# **What is Apache Spark?**

According to [Wikipedia](https://en.wikipedia.org/wiki/Apache_Spark):

*Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.*

From [Official Website](https://spark.apache.org/):

*Apache Spark™ is a unified analytics engine for large-scale data processing.*

In short, Apache Spark is a framework which is used for processing, querying and analyzing Big data. Since the computation is done in memory hence it’s multiple fold fasters than the competitors like MapReduce and others. The rate at which terabytes of data is being produced every day, there was a need for a solution that could provide real-time analysis at high speed. Some of the Spark features are:

* It is 100 times faster than traditional large-scale data processing frameworks.
* Easy to use as you can write Spark applications in Python, R, and Scala.
* It provides libraries for SQL, Steaming and Graph computations.

# **Apache Spark Components**

Diagram

Description automatically generated

# **Spark Core**

It contains the basic functionality of Spark like task scheduling, memory management, interaction with storage, etc.

# **Spark SQL**

It is a set of libraries used to interact with structured data. It used an SQL like interface to interact with data of various formats like CSV, JSON, Parquet, etc.

# **Spark Streaming**

Spark Streaming is a Spark component that enables the processing of live streams of data. Live streams like Stock data, Weather data, Logs, and various others.

# **MLib**

MLib is a set of Machine Learning Algorithms offered by Spark for both supervised and unsupervised learning

# **GraphX**

It is Apache Spark’s API for graphs and graph-parallel computation. It extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. It provides a uniform tool for ETL, exploratory analysis and iterative graph computations.

# **Spark Cluster Managers**

Spark supports the following resource/cluster managers:

* **Spark Standalone** — a simple cluster manager included with Spark
* **Apache Mesos** — a general cluster manager that can also run Hadoop applications.
* **Apache Hadoop YARN** — the resource manager in Hadoop 2
* **Kubernetes** — an open source system for automating deployment, scaling, and management of containerized applications.

# **Setup and Installation**

Download the binary of Apache Spark from [here](https://spark.apache.org/downloads.html). You must have Scala installed on the system and its path should also be set.

For this tutorial, we are using version 2.4.3 which was released in May 2019. Move the folder in /usr/local

mv spark-2.4.3-bin-hadoop2.7 /usr/local/spark

And then export the path of both Scala and Spark.

#Scala Path  
export PATH="/usr/local/scala/bin:$PATH"#Apache Spark path  
export PATH="/usr/local/spark/bin:$PATH"

Invoke the Spark Shell by running the spark-shell command on your terminal. If all goes well, you will see something like below:

It loads the Scala based shell. Since we are going to use Python language then we have to install **PySpark**.

pip install pyspark

Once it is installed you can invoke it by running the command pyspark in your terminal:

Graphical user interface, text, application

Description automatically generated

You find a typical Python shell but this is loaded with Spark libraries.

# **Development in Python**

Let’s start writing our first program.

from pyspark.sql import SparkSession  
from pyspark.sql import SQLContextif \_\_name\_\_ == '\_\_main\_\_':  
 scSpark = SparkSession \  
 .builder \  
 .appName("reading csv") \  
 .getOrCreate()

We have imported two libraries: SparkSession and SQLContext.

SparkSession is the entry point for programming Spark applications. It let you interact with DataSet and DataFrame APIs provided by Spark. We set the application name by calling appName. The getOrCreate() method either returns a new SparkSession of the app or returns the existing one.

Our next objective is to read CSV files. I have created a sample CSV file, called data.csv which looks like below:

name,age,country  
adnan,40,Pakistan  
maaz,9,Pakistan  
musab,4,Pakistan  
ayesha,32,Pakistan

And the code:

if \_\_name\_\_ == '\_\_main\_\_':  
 scSpark = SparkSession \  
 .builder \  
 .appName("reading csv") \  
 .getOrCreate()data\_file = '/Development/PetProjects/LearningSpark/data.csv'  
 sdfData = scSpark.read.csv(data\_file, header=True, sep=",").cache()  
 print('Total Records = {}'.format(sdfData.count()))  
 sdfData.show()

I set the file path and then called .read.csv to read the CSV file. The parameters are self-explanatory. The .cache() caches the returned resultset hence increase the performance. When I run the program it returns something like below:

Text

Description automatically generated

Looks interesting, No? Now, what if I want to read multiple files in a dataframe. Let’s create another file, I call it data1.csv and it looks like below:

name,age,country

noreen,23,England

Aamir,9,Pakistan

Noman,4,Pakistan

Rasheed,12,Pakistan

All I have to do this:

data\_file = '/Development/PetProjects/LearningSpark/data\*.csv' and it will read all files starts with dataand of type CSV.

What it will do that it’d read all CSV files that match a pattern and dump result

Graphical user interface, text

Description automatically generated

As you can see, it dumps all the data from the CSVs into a single dataframe. Pretty cool huh.

But one thing, this dumping will only work if all the CSVs follow a certain schema. If you have a CSV with different column names then it’s gonna return the following message.

19/06/04 18:59:05 WARN CSVDataSource: Number of column in CSV header is not equal to number of fields in the schema:  
 Header length: 3, schema size: 17  
CSV file: file:///Development/PetProjects/LearningSpark/data.csv

As you can see, Spark complains about CSV files that are not the same are unable to be processed.

You can perform many operations with DataFrame but Spark provides you much easier and familiar interface to manipulate the data by using SQLContext. It is the gateway to SparkSQL which lets you use SQL like queries to get the desired results.

Before we move further, let’s play with some real data. For that purpose, we are using Supermarket’s sales data which I got from [Kaggle](https://www.kaggle.com/aungpyaeap/supermarket-sales). Before we try SQL queries, let’s try to group records by Gender. We are dealing with the **EXTRACT** part of the ETL here.

data\_file = '/Development/PetProjects/LearningSpark/supermarket\_sales.csv'  
sdfData = scSpark.read.csv(data\_file, header=True, sep=",").cache()gender = sdfData.groupBy('Gender').count()  
print(gender.show())

When you run, it returns something like below:

Graphical user interface, text, application

Description automatically generated

groupBy() groups the data by the given column. In our case, it is the **Gender**column.

SparkSQL allows you to use SQL like queries to access the data.

sdfData.registerTempTable("sales")  
output = scSpark.sql('SELECT \* from sales')  
output.show()

First, we create a temporary table out of the dataframe. For that purpose registerTampTable is used. In our case the table name is **sales**. Once it’s done you can use typical SQL queries on it. In our case it is **Select \* from sales**.

Table

Description automatically generated

Or something like below:

output = scSpark.sql('SELECT \* from sales WHERE `Unit Price` < 15 AND Quantity < 10')  
output.show()

Or even aggregated values.

output = scSpark.sql('SELECT COUNT(\*) as total, City from sales GROUP BY City')  
output.show()

Text

Description automatically generated

Pretty flexible, right?

We are just done with the **TRANSFORM** part of the ETL here.

Finally the **LOAD** part of the ETL. What if you want to save this transformed data? Well, you have many options available, RDBMS, XML or JSON.

output.write.format('json').save('filtered.json')

When you run it Sparks create the following folder/file structure.

Text

Description automatically generated

It created a folder with the name of the file, in our case it is **filtered.json**. Then, a file with the name **\_SUCCESS**tells whether the operation was a success or not. In case it fails a file with the name **\_FAILURE** is generated. Then, you find multiple files here. The reason for multiple files is that each work is involved in the operation of writing in the file. If you want to create a single file(which is not recommended) then coalesce can be used that collects and reduces the data from all partitions to a single dataframe.

output.coalesce(1).write.format('json').save('filtered.json')

And it will output the following data:

{"total":328,"City":"Naypyitaw"}  
{"total":332,"City":"Mandalay"}  
{"total":340,"City":"Yangon"}

# **MySQL and Apache Spark Integration**

The above dataframe contains the transformed data. We would like to load this data into MYSQL for further usage like Visualization or showing on an app.

First, we need the MySQL connector library to interact with Spark. We will download the connector from [MySQL website](https://cdn.mysql.com/Downloads/Connector-J/mysql-connector-java-8.0.16.tar.gz) and put it in a folder. We will amend SparkSession to include the JAR file.

scSpark = SparkSession \  
 .builder \  
 .appName("reading csv") \  
 .config("spark.driver.extraClassPath", "/usr/local/spark/jars/mysql-connector-java-8.0.16.jar") \  
 .getOrCreate()

The output now looks like below:

output = scSpark.sql('SELECT COUNT(\*) as total, City from sales GROUP BY City')  
 output.show()  
 output.write.format('jdbc').options(  
 url='jdbc:mysql://localhost/spark',  
 driver='com.mysql.cj.jdbc.Driver',  
 dbtable='city\_info',  
 user='root',  
 password='root').mode('append').save()

I created the required Db and table in my DB before running the script. If all goes well you should see the result like below:

Table

Description automatically generated

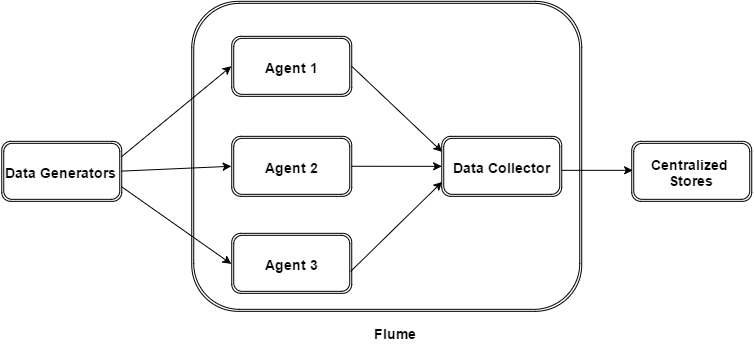
As you can see, Spark makes it easier to transfer data from One data source to another.

# **Conclusion**

Apache Spark is a very demanding and useful Big Data tool that helps to write ETL very easily. You can load the Petabytes of data and can process it without any hassle by setting up a cluster of multiple nodes. This tutorial just gives you the basic idea of Apache Spark’s way of writing ETL. You should check the docs and other resources to dig deeper.

FLUME architecture:

The following illustration depicts the basic architecture of Flume. As shown in the illustration, **data generators** (such as Facebook, Twitter) generate data which gets collected by individual Flume **agents** running on them. Thereafter, a **data collector** (which is also an agent) collects the data from the agents which is aggregated and pushed into a centralized store such as HDFS or HBase.



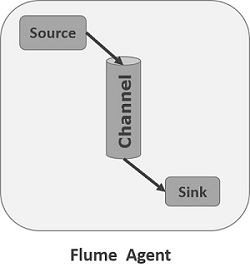
## Flume Event

An **event** is the basic unit of the data transported inside **Flume**. It contains a payload of byte array that is to be transported from the source to the destination accompanied by optional headers. A typical Flume event would have the following structure −



## Flume Agent

An **agent** is an independent daemon process (JVM) in Flume. It receives the data (events) from clients or other agents and forwards it to its next destination (sink or agent). Flume may have more than one agent. Following diagram represents a **Flume Agent**



As shown in the diagram a Flume Agent contains three main components namely, **source**, **channel**, and **sink**.

### Source

A **source** is the component of an Agent which receives data from the data generators and transfers it to one or more channels in the form of Flume events.

Apache Flume supports several types of sources and each source receives events from a specified data generator.

**Example** − Avro source, Thrift source, twitter 1% source etc.

### Channel

A **channel** is a transient store which receives the events from the source and buffers them till they are consumed by sinks. It acts as a bridge between the sources and the sinks.

These channels are fully transactional and they can work with any number of sources and sinks.

**Example** − JDBC channel, File system channel, Memory channel, etc.

### Sink

A **sink** stores the data into centralized stores like HBase and HDFS. It consumes the data (events) from the channels and delivers it to the destination. The destination of the sink might be another agent or the central stores.

**Example** − HDFS sink

**Note** − A flume agent can have multiple sources, sinks and channels. We have listed all the supported sources, sinks, channels in the Flume configuration chapter of this tutorial.

## Additional Components of Flume Agent

What we have discussed above are the primitive components of the agent. In addition to this, we have a few more components that play a vital role in transferring the events from the data generator to the centralized stores.

### Interceptors

Interceptors are used to alter/inspect flume events which are transferred between source and channel.

### Channel Selectors

These are used to determine which channel is to be opted to transfer the data in case of multiple channels. There are two types of channel selectors −

* **Default channel selectors** − These are also known as replicating channel selectors they replicates all the events in each channel.
* **Multiplexing channel selectors** − These decides the channel to send an event based on the address in the header of that event.

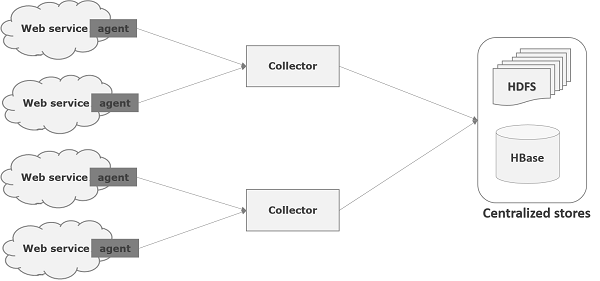
### Sink Processors

These are used to invoke a particular sink from the selected group of sinks. These are used to create failover paths for your sinks or load balance events across multiple sinks from a channel.

Flume is a framework which is used to move log data into HDFS. Generally events and log data are generated by the log servers and these servers have Flume agents running on them. These agents receive the data from the data generators.

The data in these agents will be collected by an intermediate node known as **Collector**. Just like agents, there can be multiple collectors in Flume.

Finally, the data from all these collectors will be aggregated and pushed to a centralized store such as HBase or HDFS. The following diagram explains the data flow in Flume.



## Multi-hop Flow

Within Flume, there can be multiple agents and before reaching the final destination, an event may travel through more than one agent. This is known as **multi-hop flow**.

## Fan-out Flow

The dataflow from one source to multiple channels is known as **fan-out flow**. It is of two types −

* **Replicating** − The data flow where the data will be replicated in all the configured channels.
* **Multiplexing** − The data flow where the data will be sent to a selected channel which is mentioned in the header of the event.

## Fan-in Flow

The data flow in which the data will be transferred from many sources to one channel is known as **fan-in flow**.

## Failure Handling

In Flume, for each event, two transactions take place: one at the sender and one at the receiver. The sender sends events to the receiver. Soon after receiving the data, the receiver commits its own transaction and sends a “received” signal to the sender. After receiving the signal, the sender commits its transaction. (Sender will not commit its transaction till it receives a signal from the receiver.)