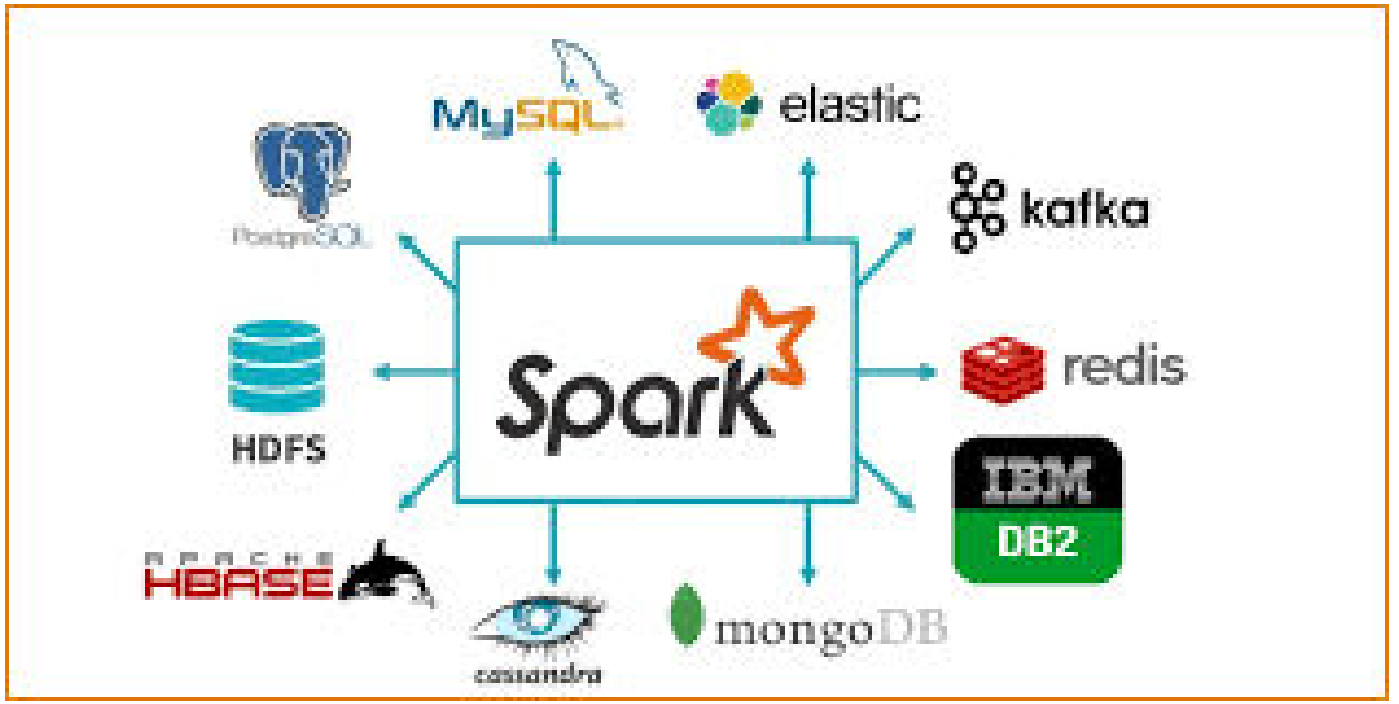


Lecturer: Narges Peyravi

What is Apache Spark?

- Apache Spark is an Open source analytical processing engine for large-scale powerful distributed data processing and machine learning applications.
- Spark was Originally developed at the University of California, Berkeley's, and later donated to the Apache Software Foundation.
- In February 2014, Spark became a Top-Level Apache Project and has been contributed by thousands of engineers making Spark one of the most active open-source projects in Apache.
- Apache Spark 3.5 is a framework that is supported in Scala, Python, R, and Java.

Features of Apache Spark

•**Fast** - It provides high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

•**Easy to Use** - It facilitates to write the application in Java, Scala, Python, R, and SQL. It also provides more than 80 high-level operators.

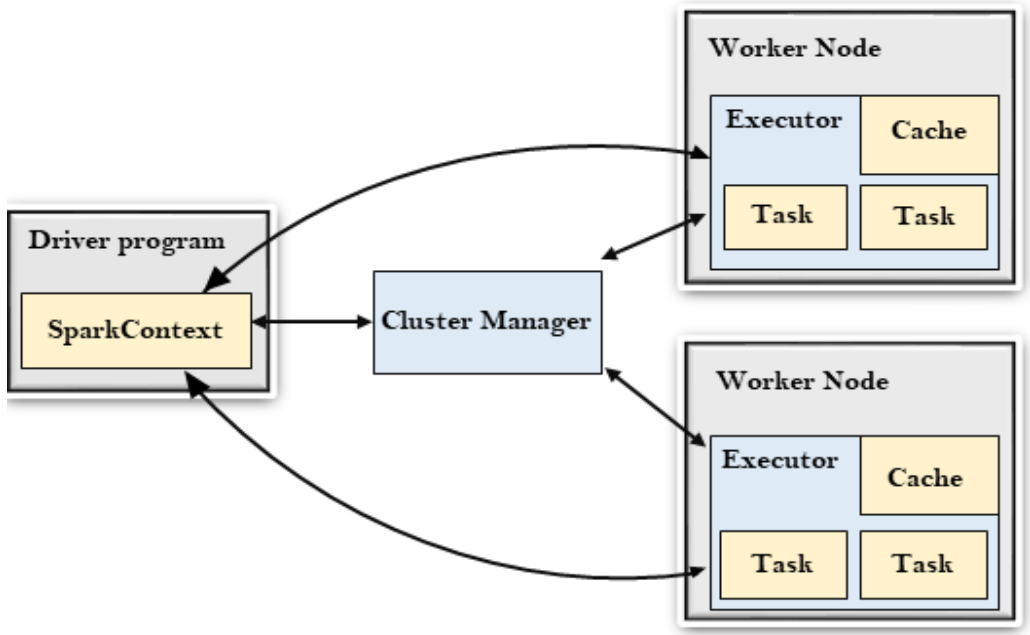
•**Generality** - It provides a collection of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming.

•**Lightweight** - It is a light unified analytics engine which is used for large scale data processing.

•**Runs Everywhere** - It can easily run on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.

Spark Architecture

- Spark works in a master-slave architecture where the master is called the "Driver" and slaves are called "Workers".
- When you run a Spark application, Spark Driver creates a context that is an entry point to your application, and all operations (transformations and actions) are executed on worker nodes, and the resources are managed by Cluster Manager.



Spark Architecture

- The Spark architecture depends upon two abstractions:
  - Resilient Distributed Dataset (RDD)
  - Directed Acyclic Graph (DAG)

Resilient Distributed Datasets (RDD):The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes.

- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

Directed Acyclic Graph (DAG): Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data.

Each node is an RDD partition, and the edge is a transformation on top of data. The graph refers the navigation whereas directed and acyclic refers to how it is done.

Spark Architecture

Driver Program

- The Driver Program is a process that runs the main() function of the application and creates the **SparkContext** object.
- The purpose of **SparkContext** is to coordinate the spark applications, running as independent sets of processes on a cluster.
- To run on a cluster, the **SparkContext** connects to a different type of cluster managers and then perform the following tasks:
  - It acquires executors on nodes in the cluster.
  - Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
  - At last, the SparkContext sends tasks to the executors to run.

Spark Architecture

Cluster Manager

•The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.

•It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.

•Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node

•The worker node is a slave node.

•Its role is to run the application code in the cluster.

Spark Architecture

Executor

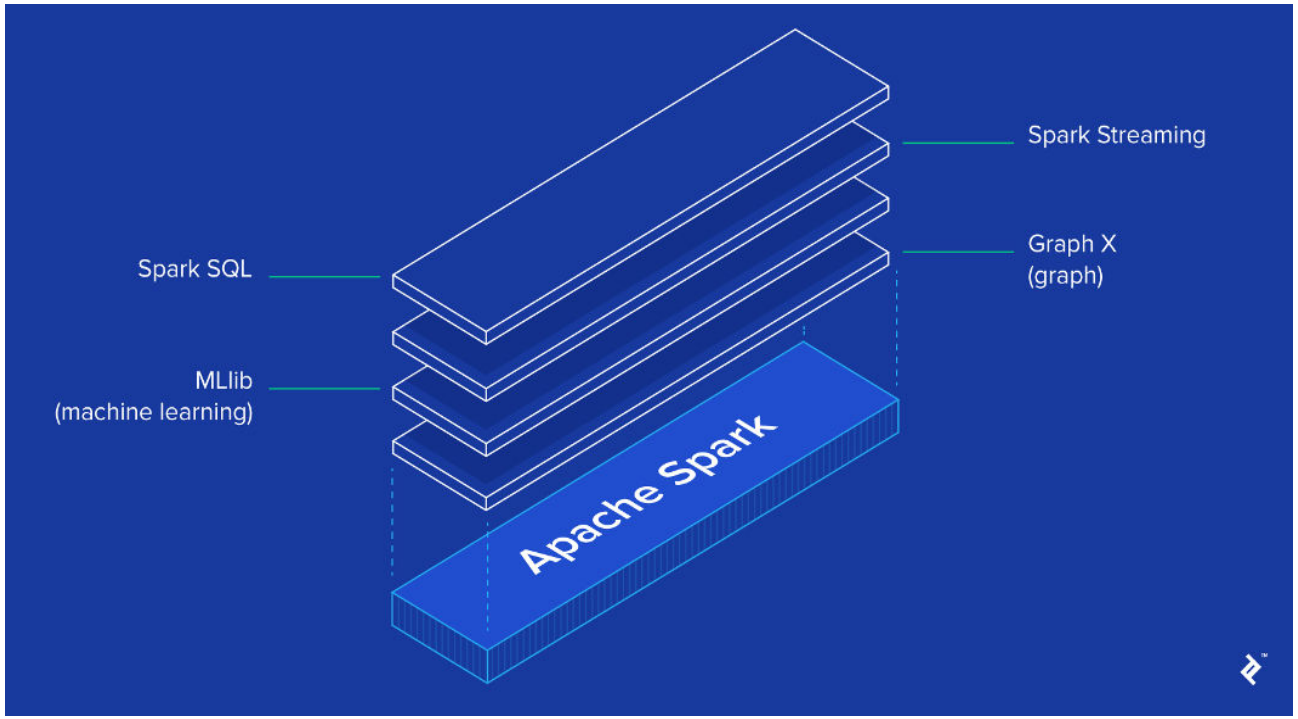
- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and write data to the external sources.
- Every application contains its executor.

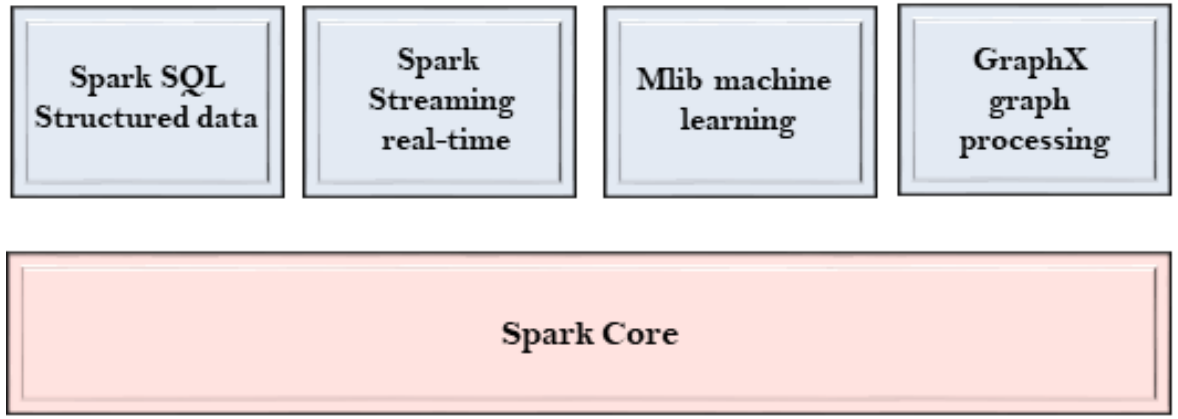
Task

- A unit of work that will be sent to one executor.

Spark Components

The Spark project consists of different types of tightly integrated components. At its core, Spark is a computational engine that can schedule, distribute and monitor multiple applications.





Spark Core

Spark Core is the base engine for large-scale parallel and distributed data processing. It is responsible for:

- The Spark Core is the heart of Spark and performs the core functionality.
- Memory management and fault recovery
- Scheduling, distributing and monitoring jobs on a cluster
- Interacting with storage systems
- It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL

•The Spark SQL is built on the top of Spark Core. It provides support for structured data.

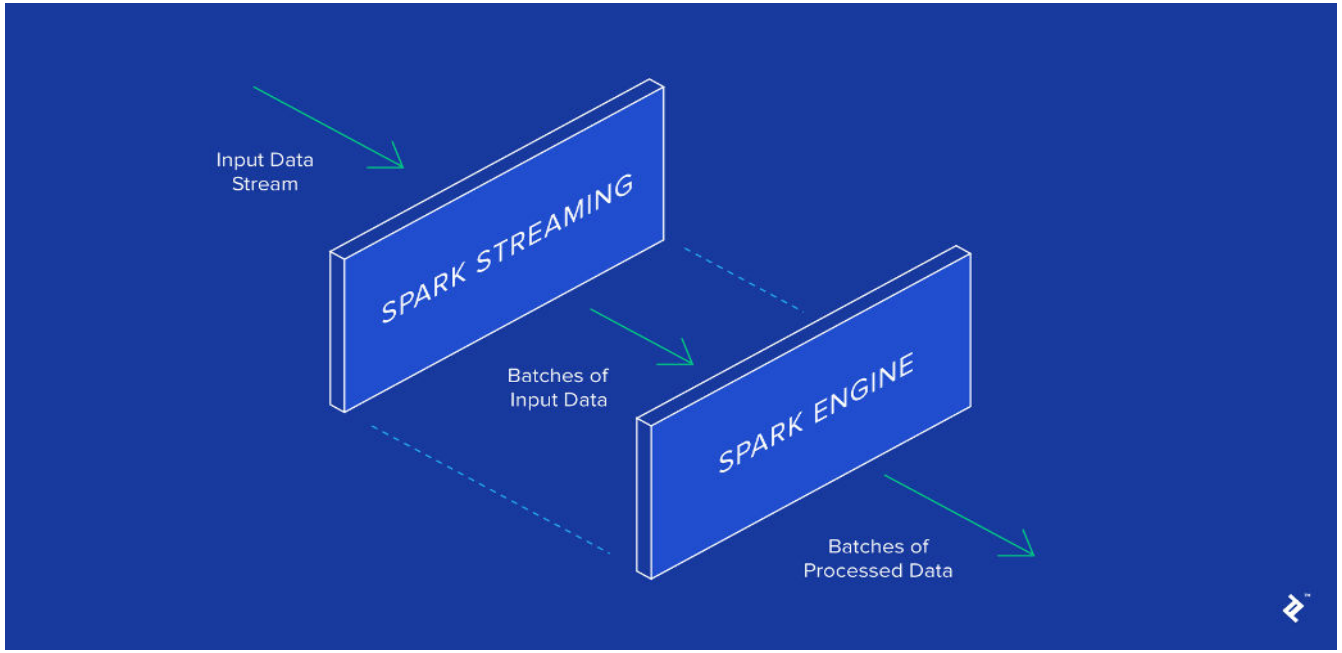
•It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL called the HQL (Hive Query Language).

•It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools.

•It also supports various sources of data like Hive tables, Parquet, and JSON.

Spark Streaming

- Spark Streaming supports real time processing of streaming data, such as production web server log files (e.g. Apache Flume and HDFS/S3), social media like Twitter, and various messaging queues like Kafka.
- Spark Streaming receives the input data streams and divides the data into batches. Next, they get processed by the Spark engine and generate final stream of results in batches, as depicted below.
- It accepts data in mini-batches and performs RDD transformations on that data.



MLlib

- MLlib is a machine learning library that provides various algorithms designed to scale out on a cluster for classification, regression, clustering, collaborative filtering, and so on.
- Some of these algorithms also work with streaming data, such as linear regression using ordinary least squares or k-means clustering .
- Apache Mahout (a machine learning library for Hadoop) has already turned away from MapReduce and joined forces on Spark MLlib.
- It is nine times faster than the disk-based implementation used by Apache Mahout.

GraphX

- The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.
- It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
- To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

What is RDD

- 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.
- 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.

RDD creation

There are two ways to create RDDs:

- Parallelizing an existing data in the driver program.
- Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

sparkContext.parallelize()

sparkContext.parallelize is used to parallelize an existing collection in your driver program. This is a basic method to create RDD.

```
//Create RDD from parallelize
val dataSeq = Seq(("Java", 20000), ("Python", 100000), ("Scala", 3000))
val rdd=spark.sparkContext.parallelize(dataSeq)
```

sparkContext.textFile()

Using textFile() method we can read a text (.txt) file from many sources like HDFS, S#, Azure, local e.t.c into RDD.

```
//Create RDD from external Data source
val rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

RDD Operations

On Spark RDD, you can perform two kinds of operations.

RDD Transformations

Spark RDD Transformations are lazy operations meaning they don't execute until you call an action on RDD.

Since RDDs are immutable, When you run a transformation(for example map()), instead of updating a current RDD, it returns a new RDD.

Some transformations on RDDs are flatMap(), map(), reduceByKey(), filter(), sortByKey() and all these return a new RDD instead of updating the current.

RDD Actions

RDD Action operation returns the values from an RDD to a driver node.

In other words, any RDD function that returns non RDD[T] is considered as an action.

RDD operations trigger the computation and return RDD in a List to the driver program.

Some actions on RDDs are count(), collect(), first(), max(), reduce() and more.

Commonly Used Transformations

Function	Description
map(function)	Returns a new RDD by applying the function on each data element
filter(function)	Returns a new dataset formed by selecting those elements of the source on which the function returns true
filterByRange(lower, upper)	Returns an RDD with elements in the specified range, upper to lower
flatMap(function)	Similar to the map function but returns a sequence, instead of a value
reduceByKey(function,[num Tasks])	Aggregates the values of a key using a function
groupByKey([num Tasks])	Converts (K,V) to (K, <iterable V>)
distinct([num Tasks])	Eliminates duplicates from an RDD
mapPartitions(function)	Similar to map but runs separately on each partition of an RDD
mapPartitionsWithIndex(function)	Similar to the map partition but also provides the function with an integer value representing the index of the partition
sample(withReplacement, fraction, seed)	Samples a fraction of data using the given random number generating seeds
union()	Returns a new RDD containing all elements and arguments of the source RDD
intersection()	Returns a new RDD that contains an intersection of elements in the datasets
Cartesian()	Returns the Cartesian product of all pairs of elements
subtract()	Returns a new RDD created by removing the elements from the source RDD with common arguments
join(RDD,[numTasks])	Joins two elements of the dataset with common arguments; when invoked on (A,B) and (A,C), it creates a new RDD, (A,(B,C))
cogroup(RDD,[numTasks])	Converts (A,B) to (A, <iterable B>)

Commonly Used Actions

Function	Description
count()	Gets the number of data elements in an RDD
collect()	Gets all data elements of an RDD as an array
reduce(function)	Aggregates data elements into an RDD by taking two arguments and returning one
take(n)	Fetches the first *n* elements of an RDD
foreach(function)	Executes the function for each data element of an RDD
first()	Retrieves the first data element of an RDD
saveastextfile(path)	Writes the content of an RDD to a text file, or a set of text files, in the local system
takeordered(n, [ordering])	Returns the first *n* elements of an RDD using either the natural order or a custom comparator