INTRODUCTION TO APACHE SPARK

Syllabus:

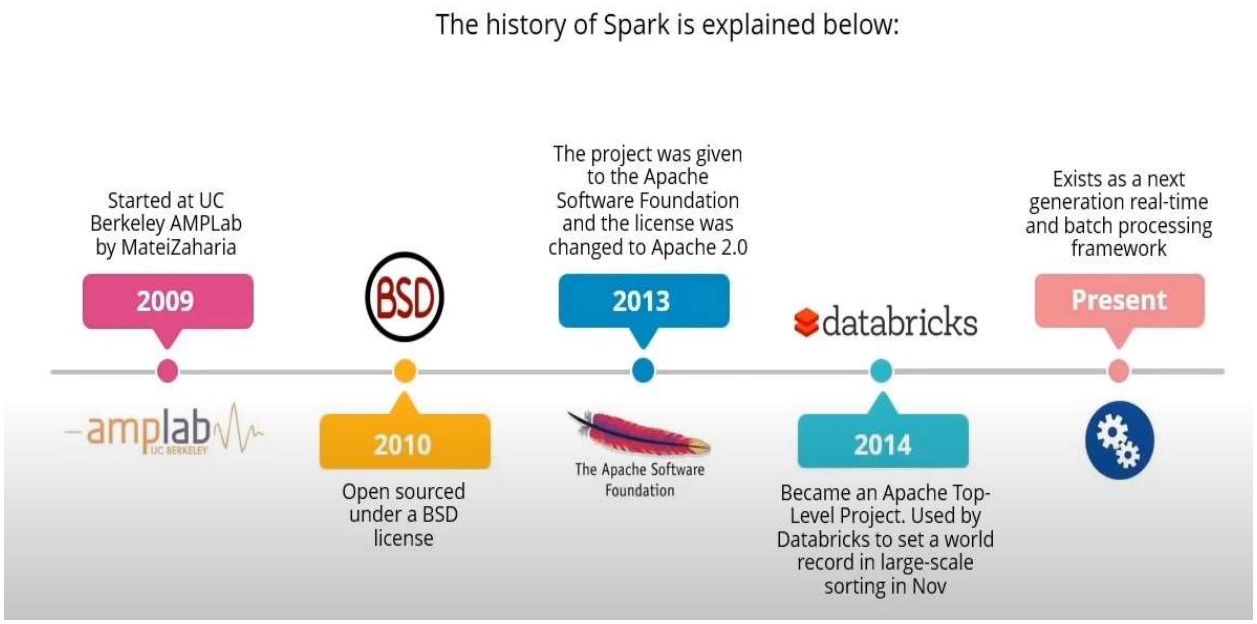
Introduction to Apache Spark: *The genesis of Spark, Hadoop at Yahoo and Spark early years, What is Apache Spark, Unified Analytics, Apache Spark's Distributed Execution, Spark Application and Spark session, Spark Jobs, Spark stages , Spark tasks, Transformation, Actions and Lazy Evaluation, Narrow and wide transformation, The Spark UI, Your first Standalone application.*

Reference: Text book-2, edureka, simplilearn

Introduction to Apache Spark: A Unified Analytics Engine

Apache Spark is an *open-source cluster computing framework* for *real-time processing*. It is of the most successful projects in the Apache Software Foundation.

History of Apache Spark



- Researchers at UC Berkeley who had previously worked on Hadoop MapReduce took on the challenge (make Hadoop and MR simpler and faster) with a project they called ***Spark***. They acknowledged that MR was inefficient (or intractable) for interactive or iterative computing jobs and a complex framework to learn. So from the onset they embraced the idea of making Spark simpler, faster, and easier.
- Apache Spark is a subproject of Hadoop developed in the year 2009 by **Matei Zaharia** in UC Berkeley's AMPLab. The first users of Spark were the group inside UC Berkeley including machine learning researchers, which used Spark to monitor and predict traffic congestion in the San Francisco Bay Area.
- Spark has open sourced in the year 2010 under BSD license.
- Spark became a project of Apache Software Foundation in the year 2013 and is now the biggest project of Apache foundation.

Batch vs. Real-Time Processing

The features below show a comparison of batch and real-time analytics in the enterprise use cases:

Batch Processing

- Large group of data/transactions is processed in a single run.
- Jobs run without any manual intervention.
- The entire data is pre-selected and fed using command-line parameters and scripts.
- It is used to execute multiple operations, handle heavy data load, reporting, and offline data workflow.

Example:
Regular reports requiring decision making

Analytics based on the data collected over a period of time is Batch Analytics

Real-Time Processing

- Data processing takes place upon data entry or command receipt instantaneously.
- It must execute on response time within stringent constraints.

Example: Fraud detection

Analytics based on immediate data for instant result is Real-Time (Stream) Analytics

Hadoop vs Spark

hadoop	APACHE Spark
Processing data using MapReduce in Hadoop is slow	Spark processes data 100 times faster than MapReduce as it is done in-memory
Performs batch processing of data	Performs both batch processing and real-time processing of data
Hadoop has more lines of code. Since it is written in Java, it takes more time to execute	Spark has fewer lines of code as it is implemented in Scala
Hadoop supports Kerberos authentication, which is difficult to manage	Spark supports authentication via a shared secret. It can also run on YARN leveraging the capability of Kerberos

What Is Apache Spark?

- **Apache Spark** is a unified engine designed for large-scale distributed data processing, on premises in data centers or in the cloud.
- Spark provides in-memory storage for intermediate computations, making it much faster than Hadoop MapReduce. It incorporates libraries with composable APIs for machine learning (MLlib), SQL for interactive queries (Spark SQL), stream processing (Structured Streaming) for interacting with real-time data, and graph processing (GraphX).

Why Spark?

Apache Spark was developed to overcome the **limitations of Hadoop** MapReduce cluster computing paradigm. Some of the drawbacks of Hadoop MapReduce are:

- Use only **Java** for application building.
- Since the maximum framework is written in Java there is some security concern. Java being heavily exploited by cybercriminals this may result in numerous security breaches.
- Opt only for batch processing. Does not support stream processing(real-time processing).
- Hadoop MapReduce uses disk-based processing.

Features of Apache Spark

Apache Spark has many features which make it a great choice as a big data processing engine. Many of these features establish the advantages of Apache Spark over other Big Data processing engines. Let us look into details of some of the main features:

1. Speed

Spark has pursued the goal of speed in several ways.

- **First,** its internal implementation benefits immensely from the hardware industry's recent huge strides in improving the price and performance of CPUs and memory. Today's commodity servers come cheap, with hundreds of gigabytes of memory, multiple cores, and the underlying Unix-based operating system taking advantage of efficient multithreading and parallel processing. The framework is optimized to take advantage of all of these factors.
- **Second,** Spark builds its query computations as a directed acyclic graph (DAG); its DAG scheduler and query optimizer construct an efficient computational graph that can usually be decomposed into tasks that are executed in parallel across workers on the cluster.
- **Third,** its physical execution engine, Tungsten, uses whole-stage code generation to generate compact code for execution.

2. Ease of use

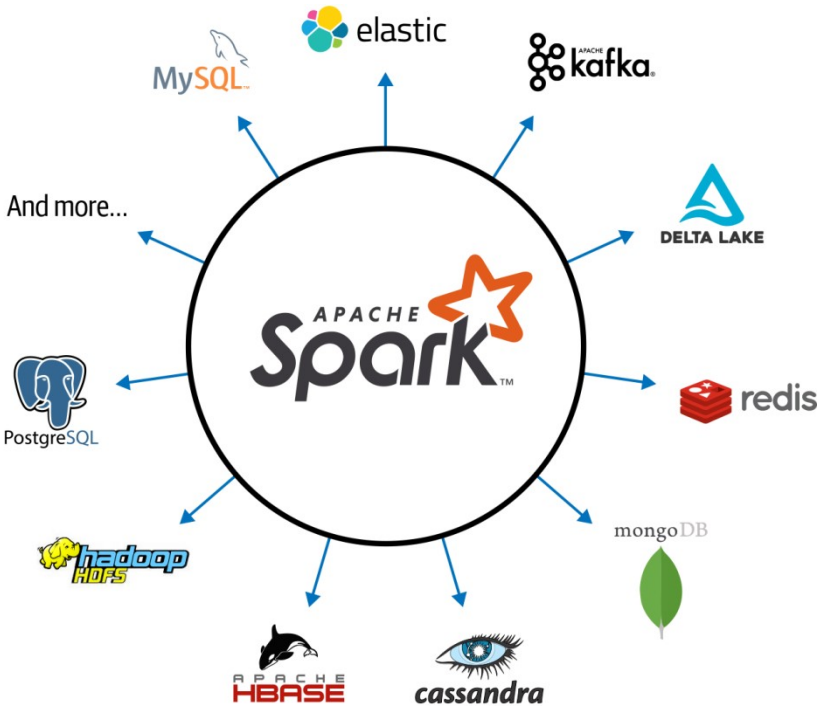
- Spark achieves simplicity by providing a fundamental abstraction of a simple logical data structure called a Resilient Distributed Dataset (RDD) upon which all other higher-level structured data abstractions, such as DataFrames and Datasets, are constructed.
- By providing a set of transformations and actions as operations, Spark offers a simple programming model that you can use to build big data applications in familiar languages.

3. Modularity

- Spark operations can be applied across many types of workloads and expressed in any of the supported programming languages: Scala, Java, Python, SQL, and R.
- Spark offers unified libraries with well-documented APIs that include the following modules as core components: Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX, combining all the workloads running under one engine.
- You can write a single Spark application that can do it all—no need for distinct engines for disparate workloads, no need to learn separate APIs.
- With Spark, you get a unified processing engine for your workloads.

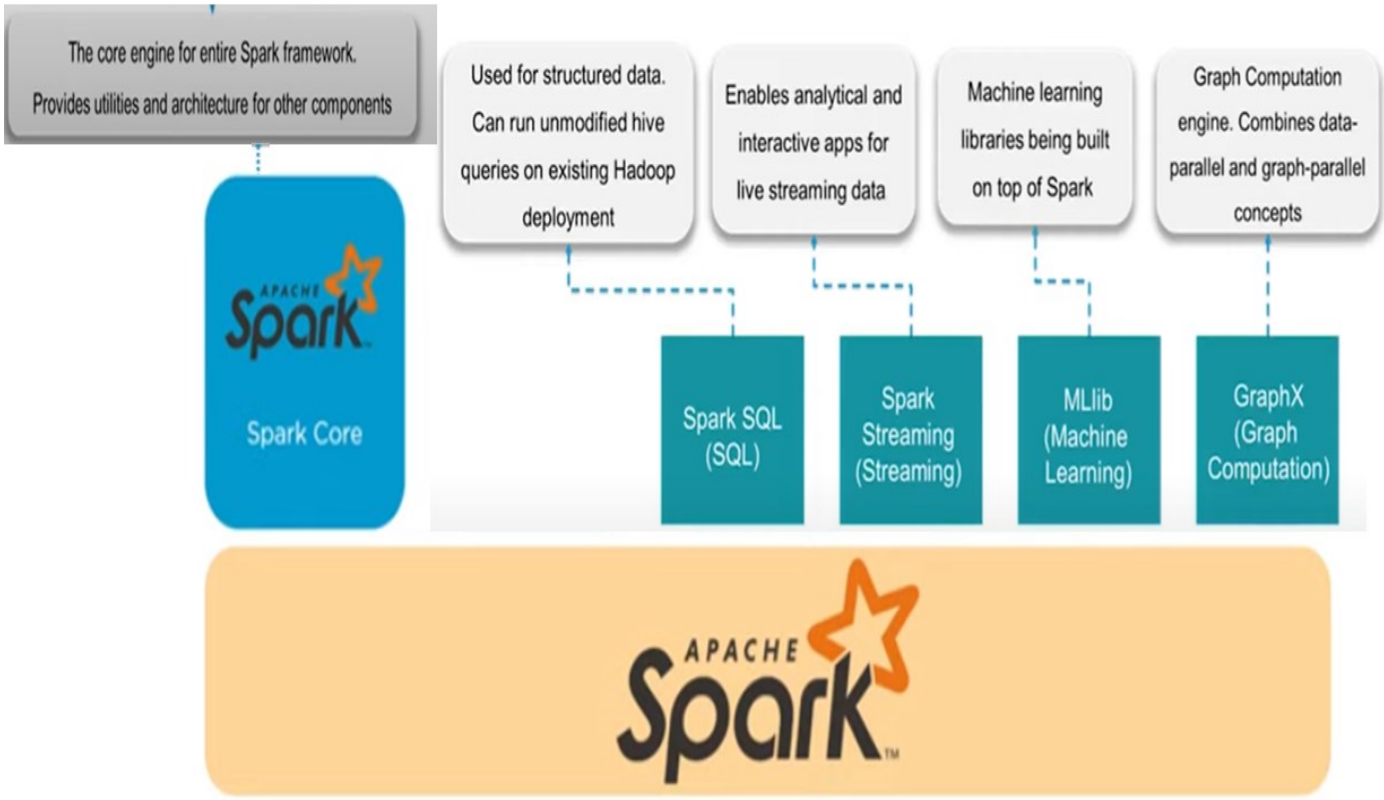
4. Extensibility

- Spark focuses on its fast, parallel computation engine rather than on storage.
- Unlike Apache Hadoop, which included both storage and compute, Spark decouples the two. That means you can use Spark to read data stored in myriad sources—Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMSs, and more—and process it all in memory.
- Spark's DataFrameReaders and DataFrameWriters can also be extended to read data from other sources, such as Apache Kafka, Kinesis, Azure Storage, and Amazon S3, into its logical data abstraction, on which it can operate.



- The community of Spark developers maintains a list of third-party Spark packages as part of the growing ecosystem (see Figure above).
- This rich ecosystem of packages includes Spark connectors for a variety of external data sources, performance monitors, and more.

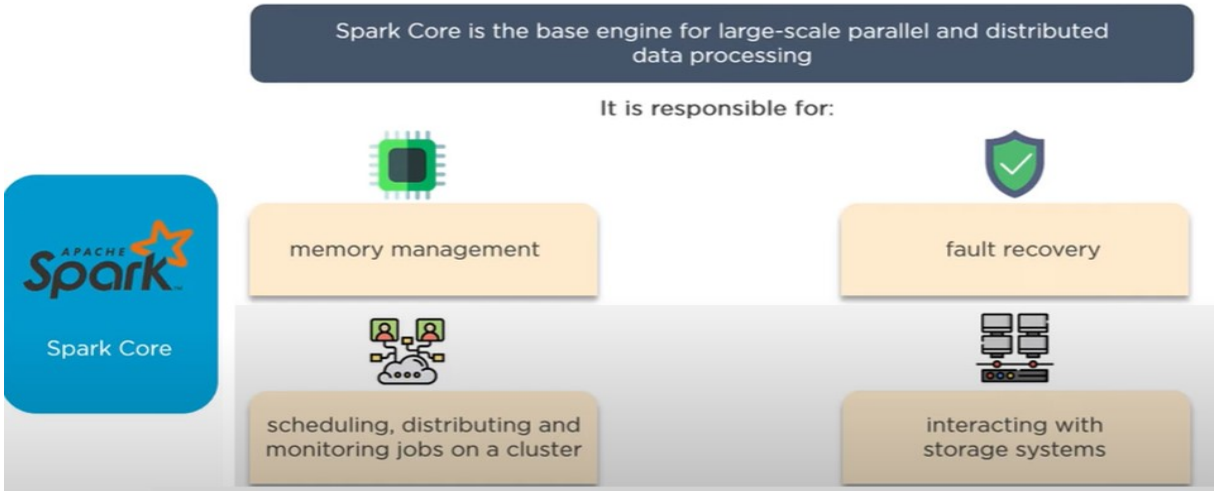
Apache Spark Components as a Unified Stack

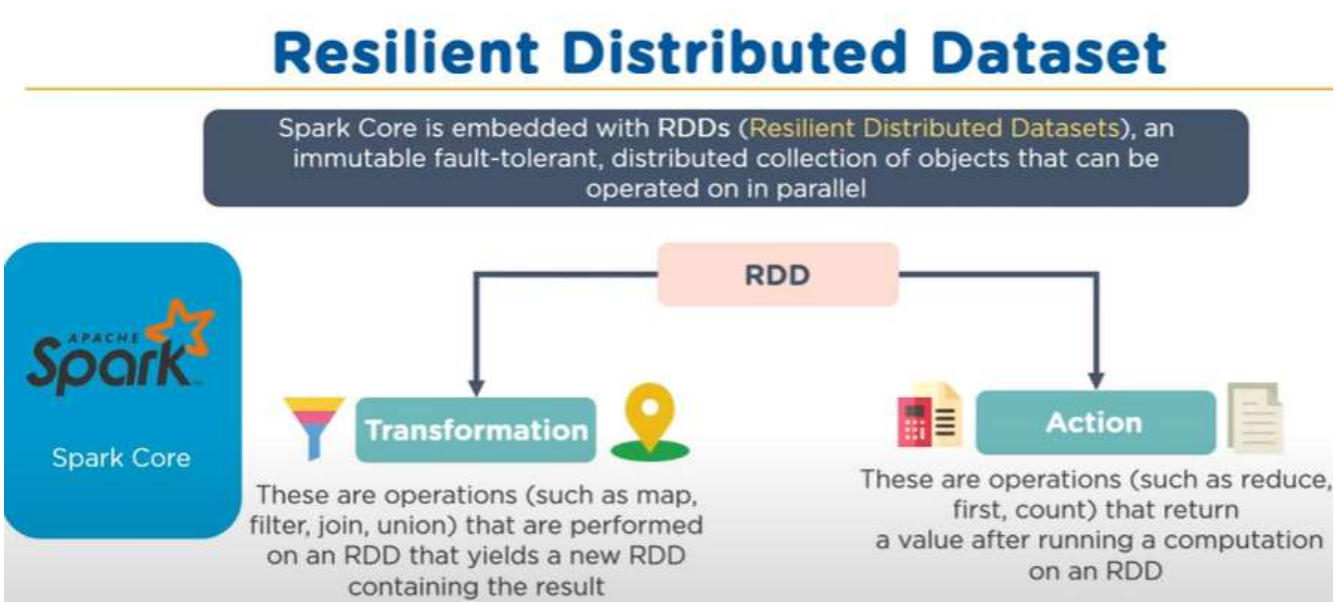


- Spark offers four distinct components as libraries for diverse workloads: **Spark SQL, Spark MLlib, Spark Structured Streaming,** and **GraphX.**
- Each of these components is separate from **Spark's core fault-tolerant engine**, in that you use APIs to write your Spark application and Spark converts this into a DAG that is executed by the core engine.

Spark Core

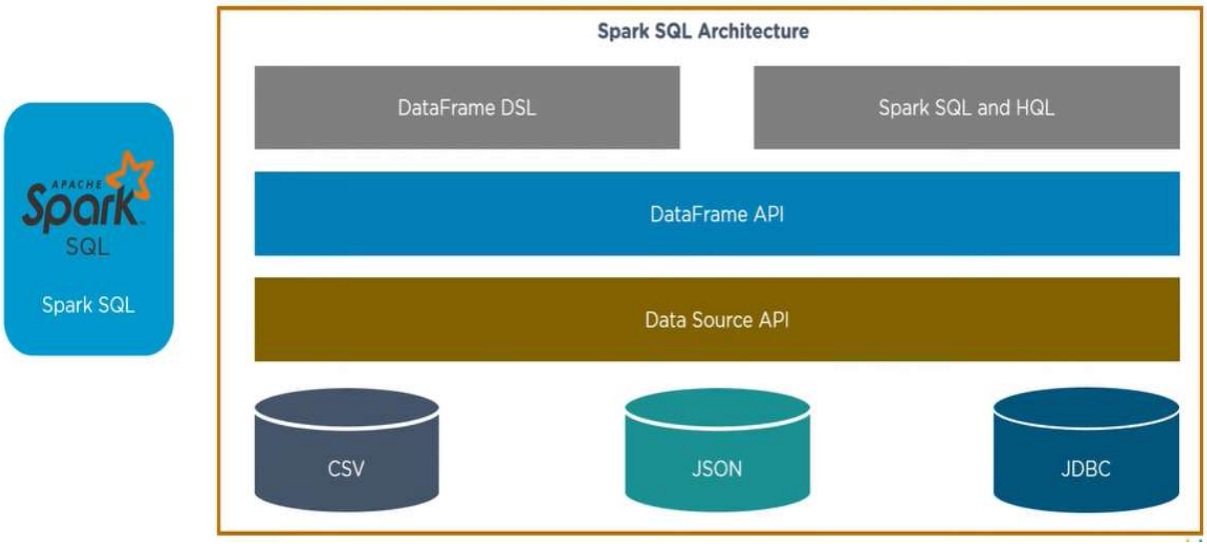
- Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems etc.
- Spark Core is also home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction.
- RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. RDD offers two types of operations: Transformation, Action.





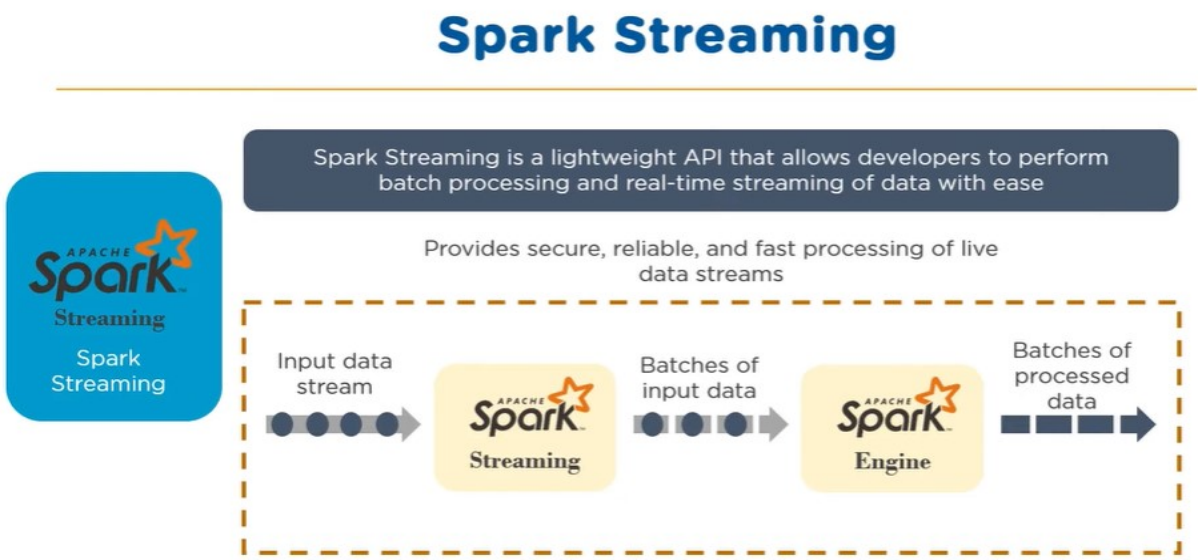
Spark SQL

- Spark SQL is Spark's package for working with structured data.
- We can read data stored in an RDBMS table or from file formats with structured data (CSV, text, JSON, Avro, ORC, Parquet, etc.) and then construct permanent or temporary tables in Spark.
- Also, when using Spark's Structured APIs in Java, Python, Scala, or R, you can combine SQL-like queries to query the data just read into a Spark DataFrame.
-



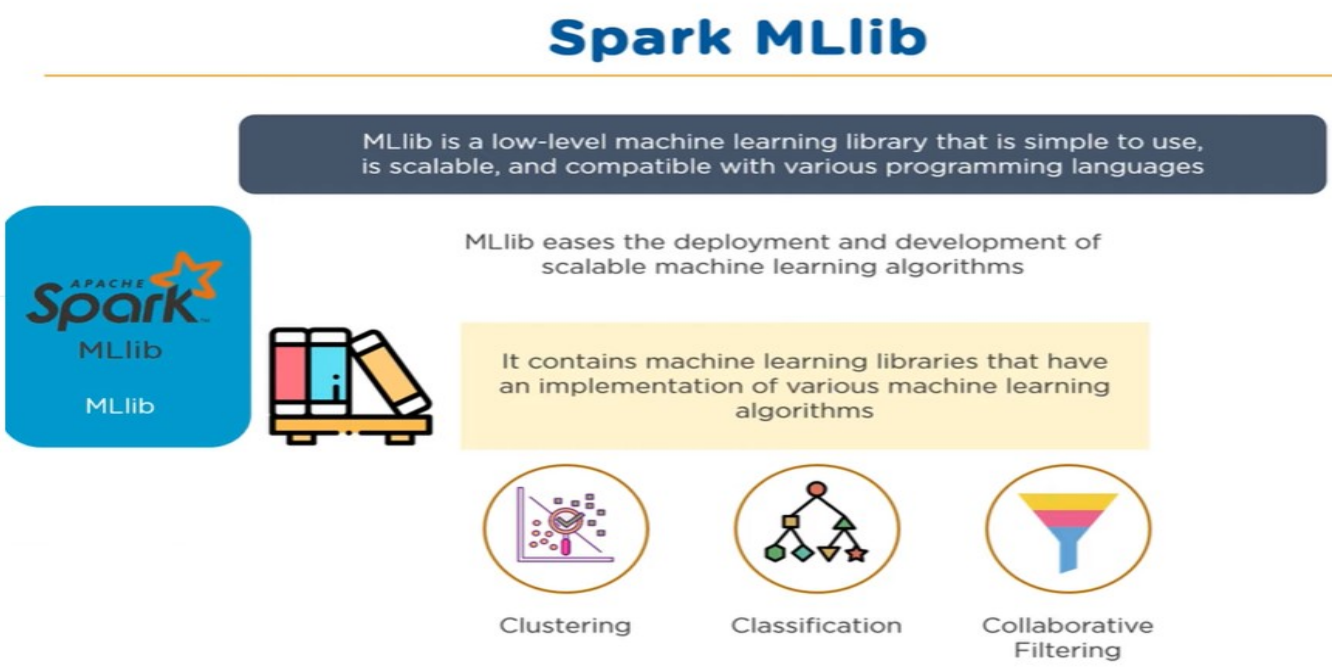
Spark Streaming

- Spark Streaming is a Spark component that enables processing of live streams of data.
- Examples of data streams include log files generated by production web servers, or queues of messages containing status updates posted by users of a web service.



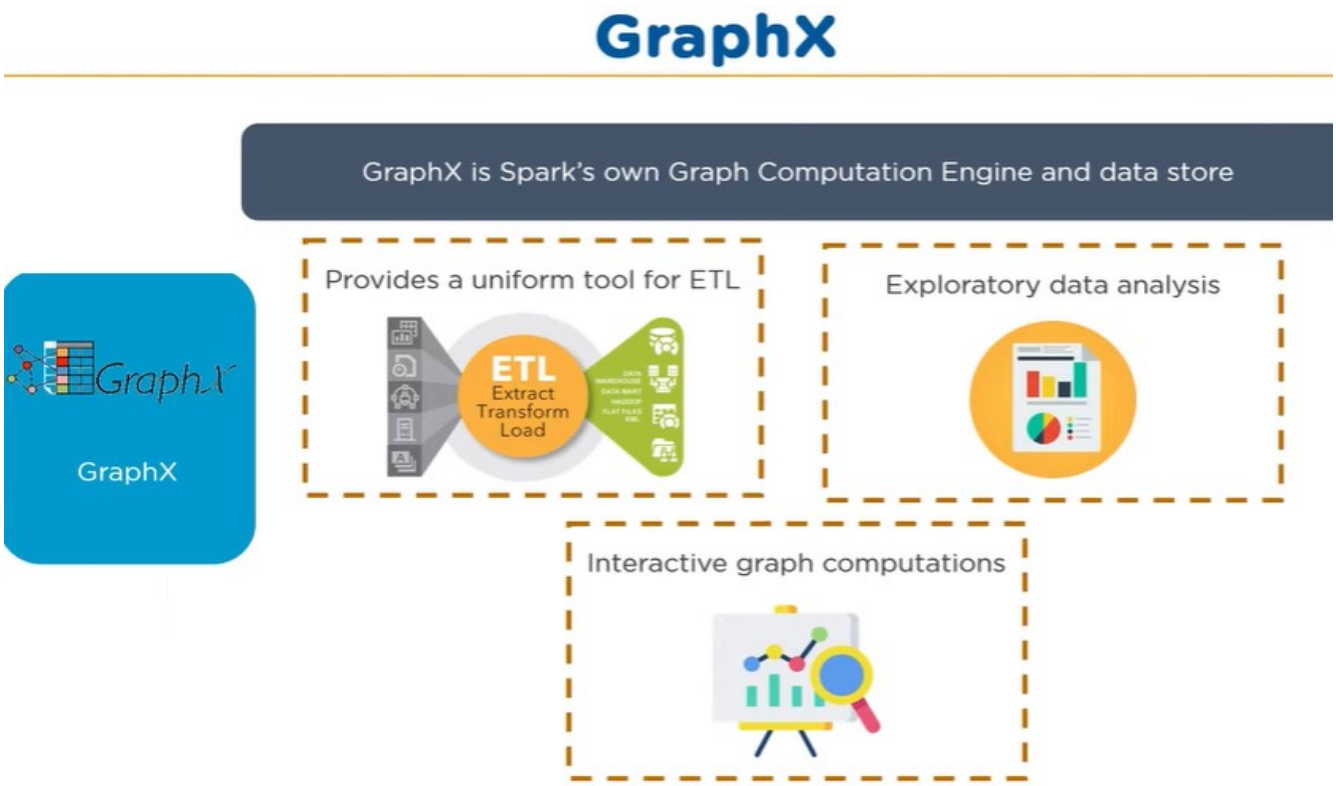
Spark MLlib

- Spark comes with a library containing common machine learning (ML) functionality, called MLlib.
- MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import etc.
- *All of these methods are designed to scale out across a cluster.*



GraphX

- GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations.
- It offers the standard graph algorithms for
- analysis, connections, and traversals, contributed by users in the community: the
- available algorithms include PageRank, Connected Components, and Triangle
- Counting.



Apache Spark's Distributed Execution/ Spark Architecture Overview

- Spark is a distributed data processing engine with its components working collaboratively on a cluster of machines.
- We shall understand how all the components of Spark's distributed architecture work together and communicate, and what deployment modes are available.
- The components of the architecture is shown in Figure 1-4

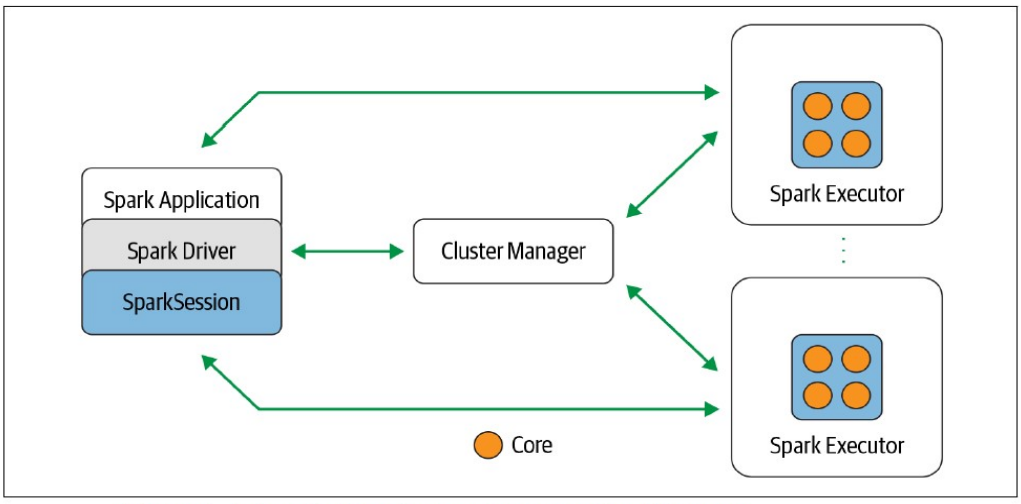


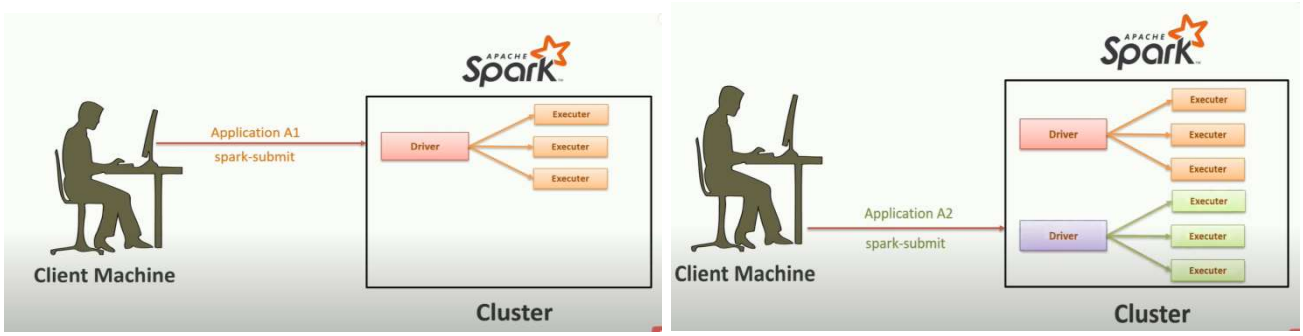
Figure 1-4. Apache Spark components and architecture

- At a high level in the Spark architecture, a Spark application consists of a driver program that is responsible for orchestrating parallel operations on the Spark cluster.
- The driver accesses the distributed components in the cluster—the Spark executors and cluster manager—through a SparkSession.

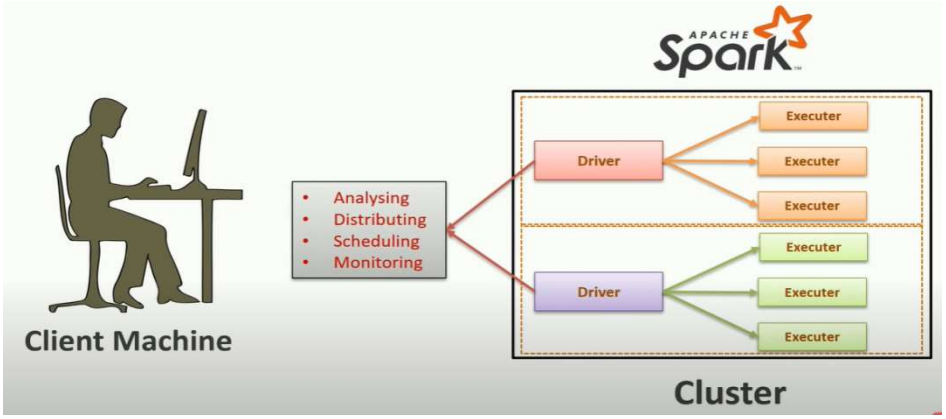
Spark driver

- As the part of the Spark application responsible for instantiating a SparkSession, the Spark driver has multiple roles:
  - it communicates with the cluster manager;
  - it requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors(JVMs); and
  - it transforms all the Spark operations into DAG computations, schedules them, and distributes their execution as tasks across the Spark executors.
  - Once the resources are allocated, it communicates directly with the executors.

Step-1: Communicating with cluster manager to allocate resources

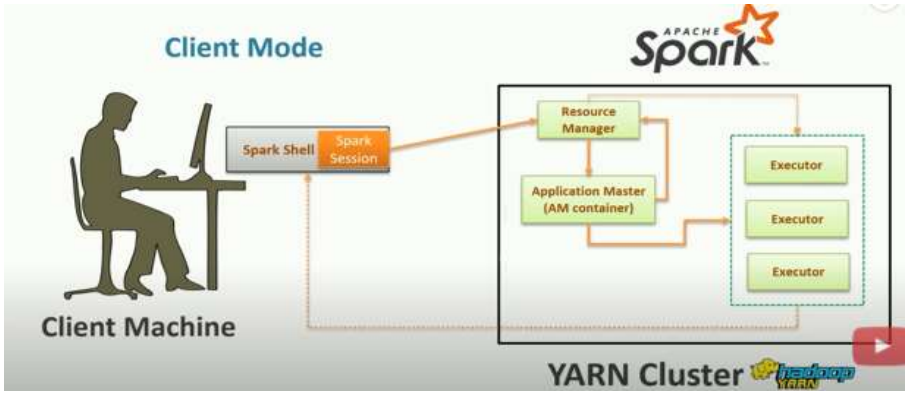


Step-2: Once the resources are allocated, it communicates directly with the executors



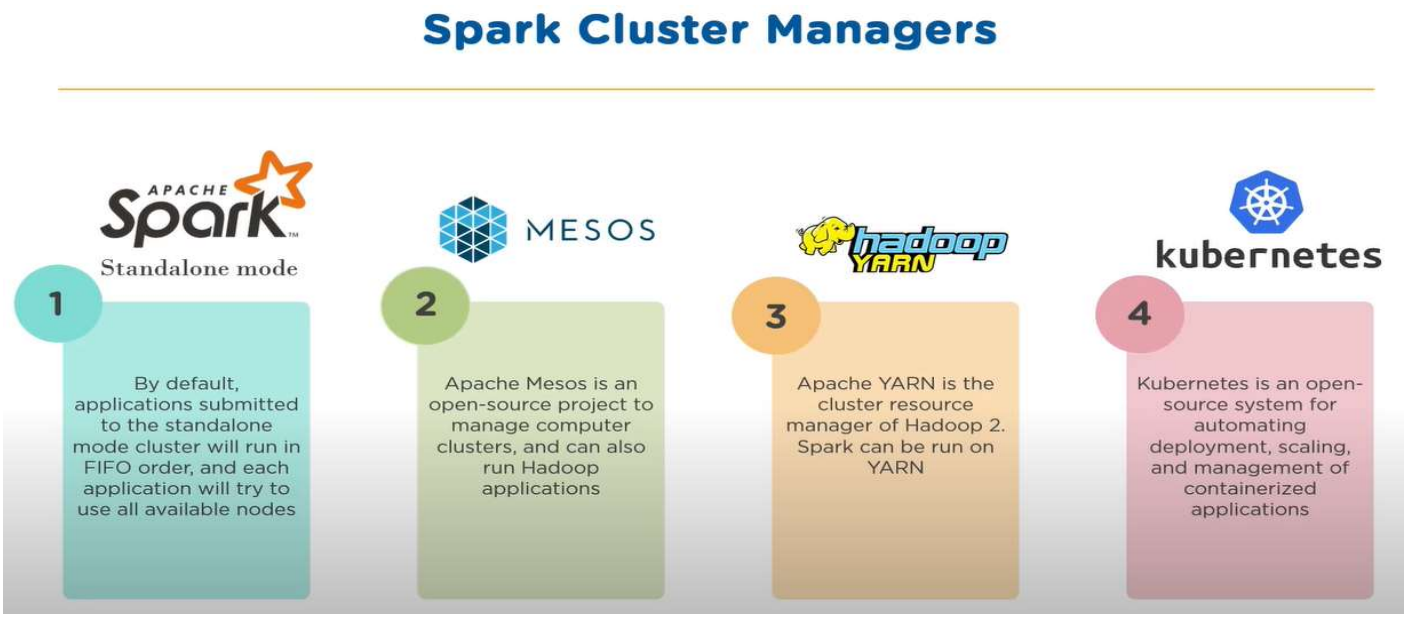
Spark Session

- Spark Session is a simplified entry point into Spark application.
- Spark Session is introduced in Spark 2.x.
- We can think **spark session** as a data structure where the driver maintains all the information including the executor location and their status.



Cluster manager

- The cluster manager is responsible for managing and allocating resources for the cluster of nodes on which your Spark application runs.
- Currently, Spark supports four cluster managers: *the* ***built-in standalone cluster manager****,* ***Apache Hadoop YARN, Apache Mesos,*** *and* ***Kubernetes.***
-



- **Standalone** – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- **Apache Mesos** – a general cluster manager that can also run Hadoop MapReduce and service applications.
- **Apache Hadoop YARN** – the resource manager in Hadoop 2.
- **Kubernetes** – an open-source system for automating deployment, scaling, and management of containerized applications.

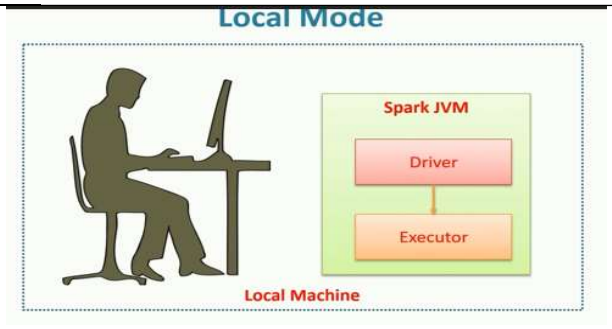
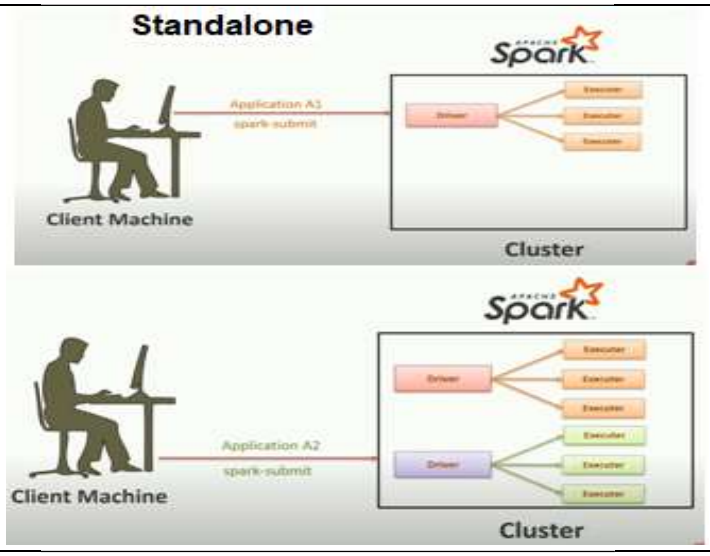
Spark executor

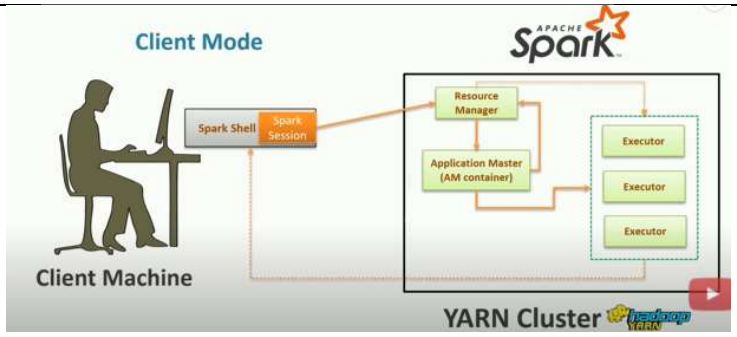
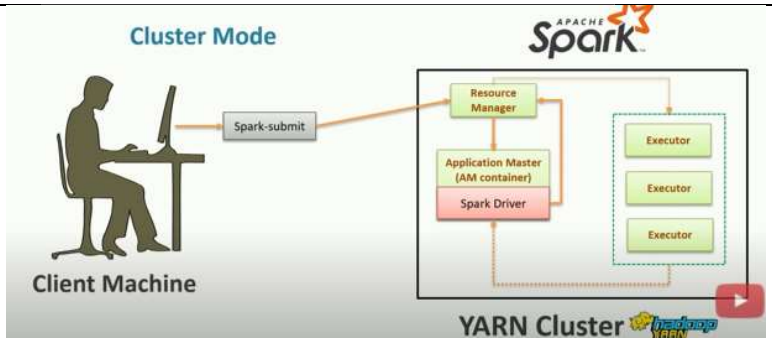
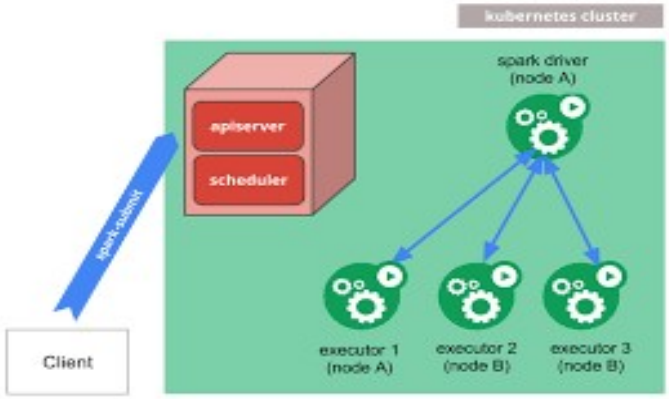
- Spark executors are the processes that perform the tasks assigned by the Spark driver.
- Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state (success or failure) and results.
- Each Spark Application has its own separate executor processes.

Deployment modes

- An attractive feature of Spark is its support for myriad deployment modes, enabling Spark to run in different configurations and environments.
- Table below summarizes the available deployment modes.

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Local	
Standalone	

YARN (client)	
YARN (cluster)	
Kubernetes	

Distributed data and partitions

- Actual physical data is distributed across storage as partitions residing in either HDFS or cloud storage (see Figure 1-5).
- While the data is distributed as partitions across the physical cluster, Spark treats each partition as a high-level logical data abstraction—asa DataFrame in memory.
- Though this is not always possible, each Spark executor is preferably allocated a task that requires it to read the partition closest to it in the network, observing data locality.

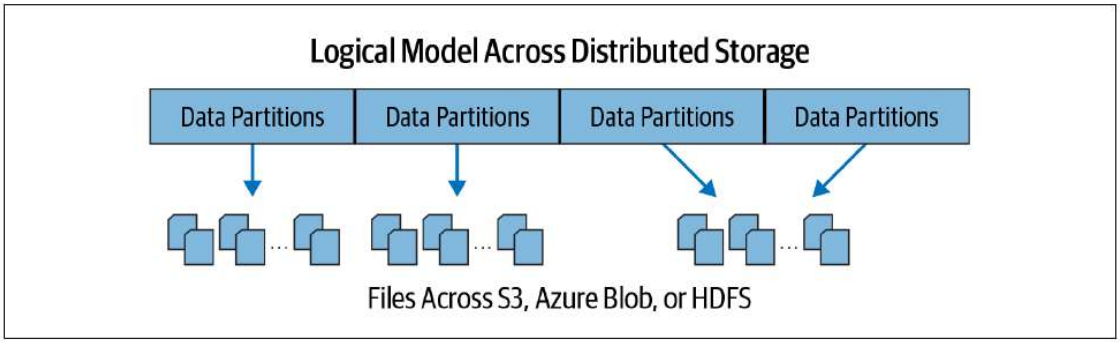


Figure 1-5. Data is distributed across physical machines

- Partitioning allows for efficient parallelism.
- A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth.
- That is, each executor's core is assigned its own data partition to work on (see Figure 1-6).

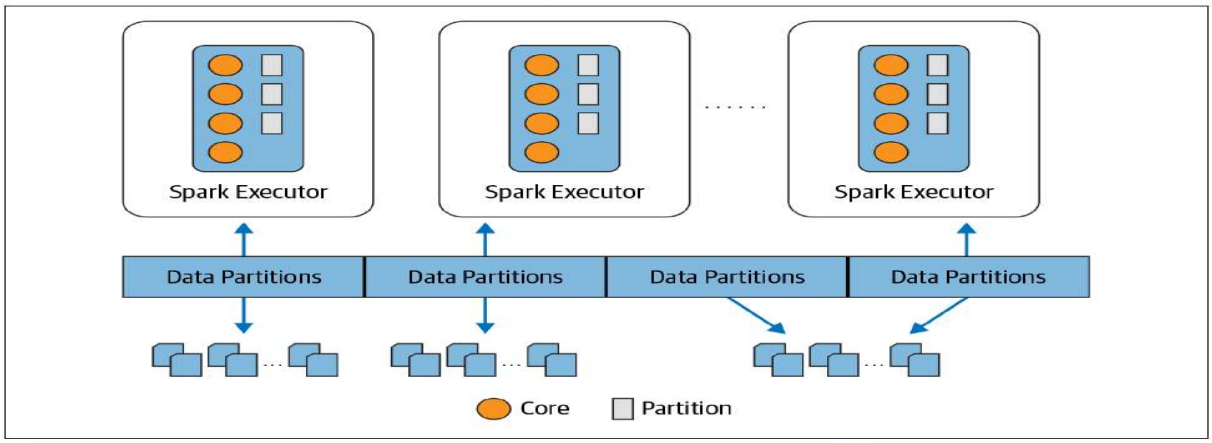


Figure 1-6. Each executor's core gets a partition of data to work on

Spark Use Case

Whether you are a data engineer, data scientist, or machine learning engineer, you'll find Spark useful for the following use cases:

- Processing in parallel large data sets distributed across a cluster
- Performing ad hoc or interactive queries to explore and visualize data sets
- Building, training, and evaluating machine learning models using MLlib
- Implementing end-to-end data pipelines from myriad streams of data
- Analyzing graph data sets and social networks

Spark Application Concepts

The key concepts are to be familiar of a Spark application to understand how the code is transformed and executed as tasks across the Spark executors. Some important terms:

Application- It is a user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.

SparkSession- It is an object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs. In an interactive Spark shell, the Spark driver instantiates a SparkSession for you, while in a Spark application, you create a SparkSession object yourself.

Job-A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()).

Stage-Each job gets divided into smaller sets of tasks called stages that depend on each other.

Task- A single unit of work or execution that will be sent to a Spark executor.

Spark Application and SparkSession:

- At the core of every Spark application is the Spark driver program, which creates a SparkSession object.
- When you're working with a Spark shell, the driver is part of the shell and the SparkSession object (accessible via the variable spark) is created for you.
- Figure 2-2 shows how Spark executes on a cluster once you've done this.
- Once you have a SparkSession, you can program Spark using the APIs to perform Spark operations.

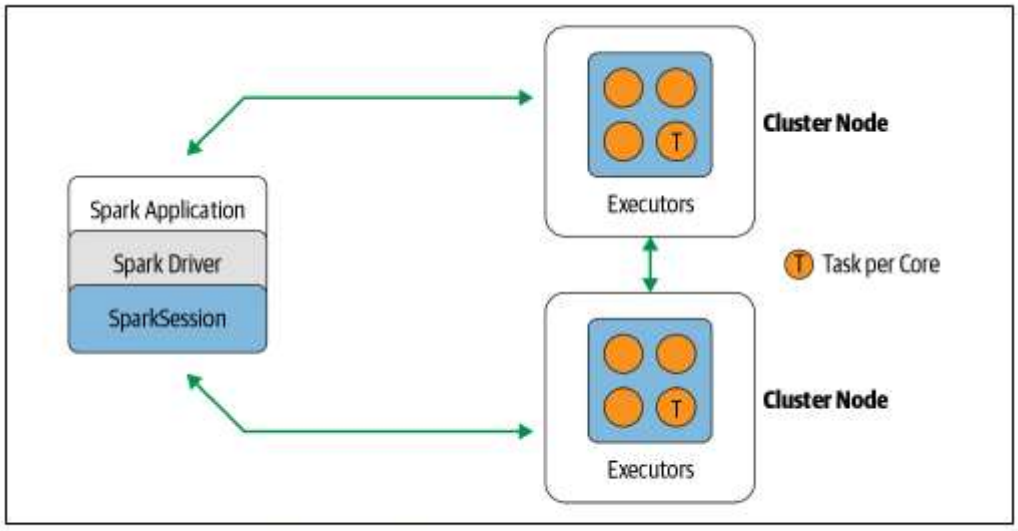


Figure 2-2. Spark components communicate through the Spark driver in Spark's distributed architecture

Spark Jobs:

- During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs (Figure 2-3).
- It then transforms each job into a DAG.
- This, in essence, is Spark's execution plan, where each node within a DAG could be a single or multiple Spark stages.

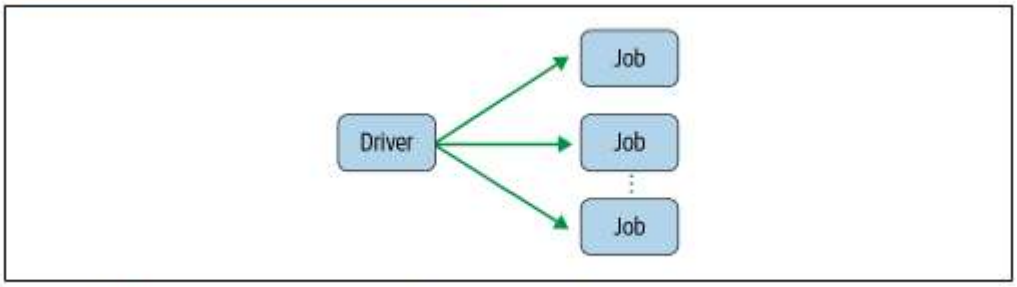


Figure 2-3. Spark driver creating one or more Spark jobs

- **(Directed Acyclic Graph) DAG** in **Apache Spark** is a set of **Vertices** and **Edges**, where *vertices* represent the **RDDs** and the *edges* represent the **Operation to be applied on RDD**.
- In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of *Action*, the created DAG submits to **DAG Scheduler** which further splits the graph into the **stages** of the **task**.

Spark Stages:

- As part of the DAG nodes, stages are created based on what operations can be performed serially or in parallel (Figure 2-4).
- Not all Spark operations can happen in a single stage, so they may be divided into multiple stages.
- Often stages are delineated on the operator's computation boundaries, where they dictate data transfer among Spark executors.

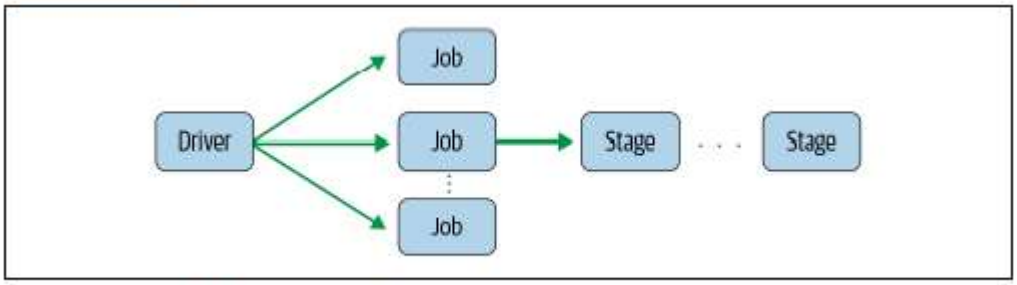


Figure 2-4. Spark job creating one or more stages

Spark Tasks:

- Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data (Figure 2-5).
- As such, an executor with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel, making the execution of Spark's tasks exceedingly parallel.

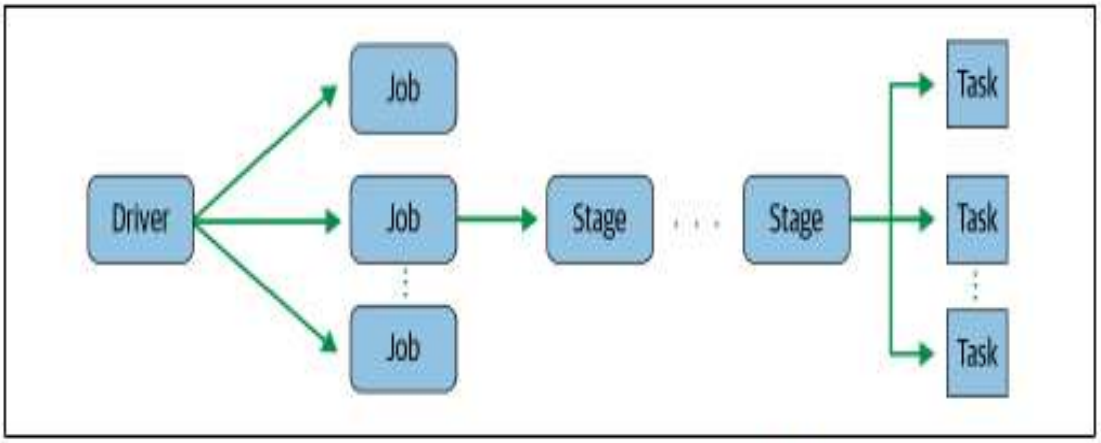
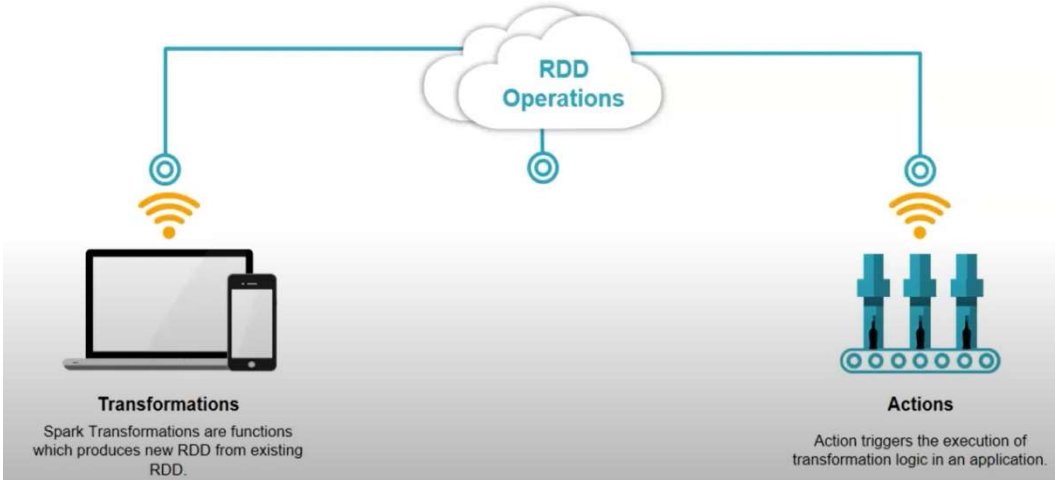


Figure 2-5. Spark stage creating one or more tasks to be distributed to executors

Types of operations

- Spark operations on distributed data can be classified into two types: ***transformations*** **and** ***actions*****.**
- **Transformations,** as the name suggests, **transform a Spark DataFrame into a new DataFrame** without altering the original data, giving it the property of immutability. Put another way, an operation such as ***select( ) or filter( )*** will not change the original DataFrame; instead, it will return the transformed results of the operation as a new DataFrame.

- Actions refer to an operation which also applies on Spark DataFrame , that instructs Spark to perform computation and send the result back to driver. This is an example of action.



- Table 2-1 lists some examples of transformations and actions.

Table 2-1. Transformations and actions as Spark operations

Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

Transformations-Lazy Evaluation:

- Transformations are **lazy** in nature i.e., they get execute when we call an action.
- All transformations are evaluated ***lazily***. That is, their results are not computed immediately, but they are recorded or remembered as a lineage. A recorded lineage allows Spark, at a later time in its execution plan, to rearrange certain transformations, coalesce them, or optimize transformations into stages for more efficient execution.
- **Lazy evaluation** is Spark's strategy for delaying execution until an action is invoked or data is "touched" (read from or written to disk).An action triggers the lazy evaluation of all the recorded transformations.

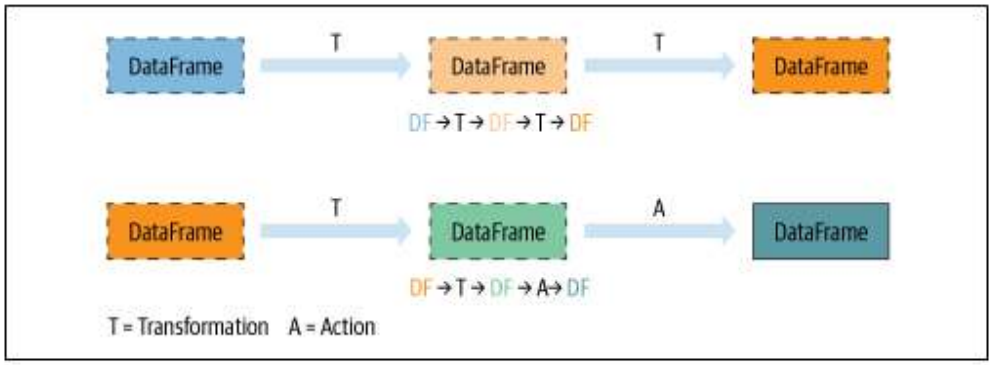


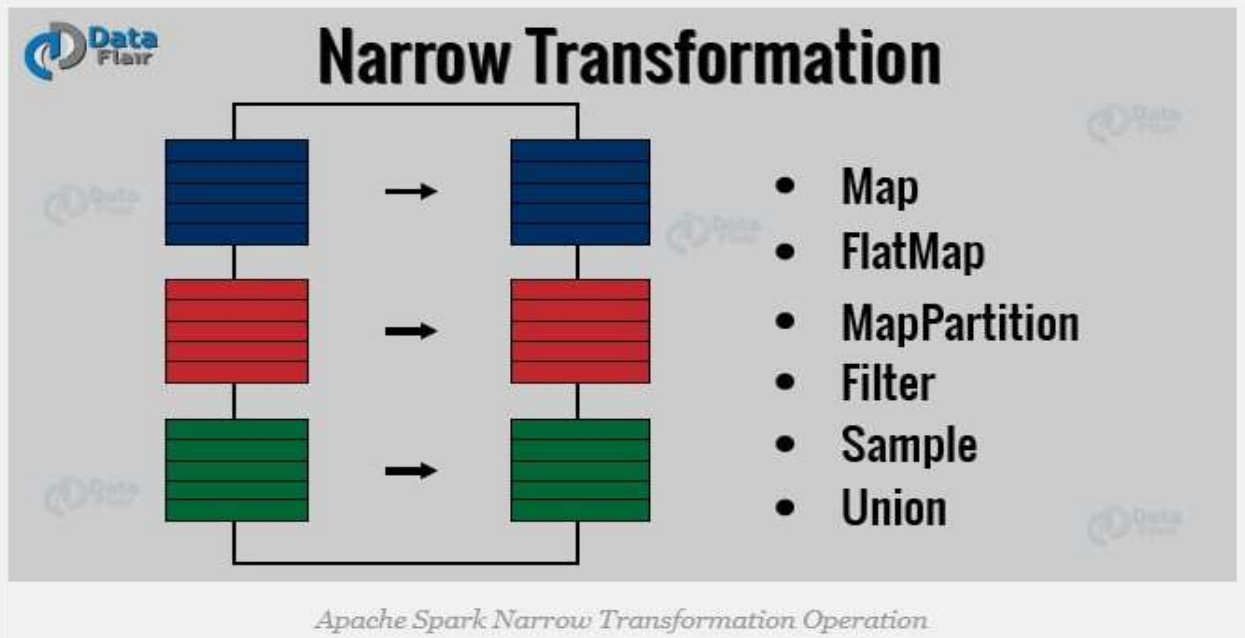
Figure 2-6. Lazy transformations and eager actions

- In Figure 2-6, all transformations T are recorded until the action A is invoked. Each transformation T produces a new DataFrame.
- While lazy evaluation allows Spark to optimize your queries by peeking into your chained transformations, lineage and data immutability provide fault tolerance.

Narrow and Wide Transformations

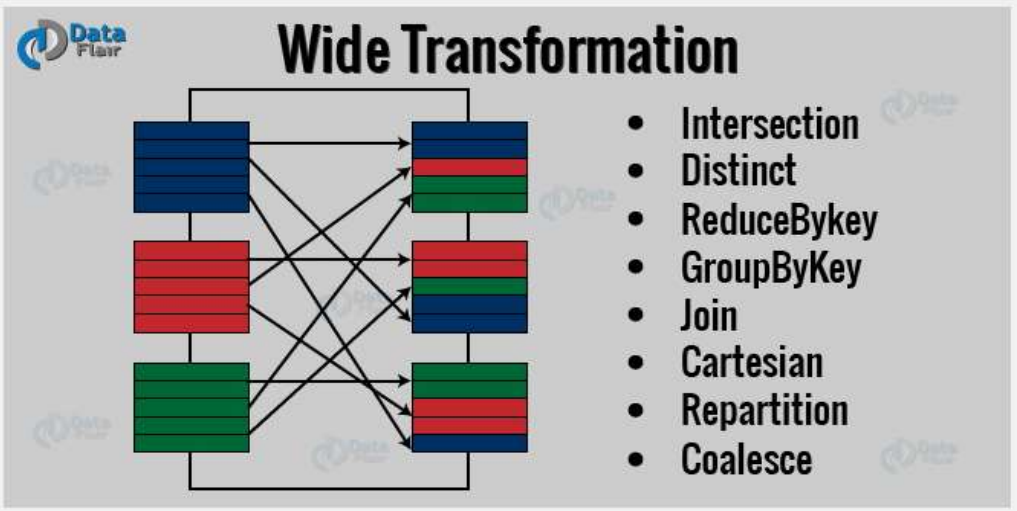
There are two types of transformations:

1. **Narrow transformation** – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of *map(), filter().*



Apache Spark Narrow Transformation Operation

2. **Wide transformation –** In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*.



Spark Wide Transformation Operations

Your First Standalone Application

Let's write a Spark program that reads a file with over 100,000 entries (where each row or line has a <*state*, *mnm_color*, *count*>) and computes and aggregates the counts for each color and state. These aggregated counts tell us the colors of M&Ms favored by students in each state. The complete Python listing is provided below:

Example 2-1. Counting and aggregating M&Ms (Python version)

```
# Import the necessary libraries.
# Since we are using Python, import the SparkSession and related functions
# from the PySpark module.
import sys

from pyspark.sql import SparkSession
from pyspark.sql.functions import count

# Build a SparkSession using the SparkSession APIs.
# If one does not exist, then create an instance. There
# can only be one SparkSession per JVM.
spark = (SparkSession
  .builder
  .appName("PythonMnMCount")
  .getOrCreate())
# Get the M&M data set filename from the command-line arguments
mnm_file = sys.argv[1]
```

```
# Read the file into a Spark DataFrame using the CSV
# format by inferring the schema and specifying that the
# file contains a header, which provides column names for comma-
# separated fields.
mnm_df = (spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load(mnm_file))

# We use the DataFrame high-level APIs. Note
# that we don't use RDDs at all. Because some of Spark's
# functions return the same object, we can chain function calls.
# 1. Select from the DataFrame the fields "State", "Color", and "Count"
# 2. Since we want to group each state and its M&M color count,
#    we use groupBy()
# 3. Aggregate counts of all colors and groupBy() State and Color
# 4  orderBy() in descending order
count_mnm_df = (mnm_df
  .select("State", "Color", "Count")
  .groupBy("State", "Color")
  .agg(count("Count").alias("Total"))
  .orderBy("Total", ascending=False))
# Show the resulting aggregations for all the states and colors;
# a total count of each color per state.
# Note show() is an action, which will trigger the above
# query to be executed.
count_mnm_df.show(n=60, truncate=False)
print("Total Rows = %d" % (count_mnm_df.count()))
```

Let's submit our first Spark job using the Python APIs (for an explanation of what the code does, please read the inline comments in Example 2-1):

```
$SPARK_HOME/bin/spark-submit mnmcount.py data/mnm_dataset.csv
```

```
+-----+------+-----+
|State|Color |Total|
+-----+------+-----+
|CA   |Yellow|1807 |
|WA   |Green |1779 |
|OR   |Orange|1743 |
|TX   |Green |1737 |
|TX   |Red   |1725 |
|CA   |Green |1723 |
|CO   |Yellow|1721 |
|CA   |Brown |1718 |
|CO   |Green |1713 |
|NV   |Orange|1712 |
|TX   |Yellow|1703 |
|NV   |Green |1698 |
|AZ   |Brown |1698 |
|CO   |Blue  |1695 |
|WY   |Green |1695 |
|NM   |Red   |1690 |
|AZ   |Orange|1689 |
|NM   |Yellow|1688 |
|NM   |Brown |1687 |
|UT   |Orange|1684 |
|NM   |Green |1682 |
|UT   |Red   |1680 |
|AZ   |Green |1676 |
|NV   |Yellow|1675 |
|NV   |Blue  |1673 |
+-----+------+-----+
:
:
Total Rows = 60
```

QUESTIONS ON UNIT-5

1. Compare batch processing and real time processing
2. Compare Hadoop and Spark
3. Explain the key features of Apache spark
4. Explain about the different Cluster Managers in Apache Spark?
5. Explain the key features of Apache spark.
6. Explain the various components in the apache spark ecosystem.
7. Explain the apache spark architecture.
8. Explain the Spark deployment modes.
9. Explain the following terms with respect to spark application
    a. SparkApplication
    b. SparkSession
    c. Job
    d. Stage
    e. Task
10. Explain various operations on distributed data/RDD in spark.
11. Explain lazy evaluation, narrow transformation, wide transformation,
12. Write a Spark program(in python) that reads a file with over 100,000 entries (where each row or line has a <*state*, *mnm_color*, *count*>) and computes and aggregates the counts for each color and state.