

04 Introduction to Spark

Apache Spark is an open source analytics engine used for big data workloads. It can handle both batches as well as real-time analytics and data processing workloads.

Apache Spark started in 2009 as a research project at the University of California, Berkeley. Researchers were looking for a way to speed up processing jobs in Hadoop systems. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing.

Spark provides native bindings for the Java, Scala, Python, and R programming languages. In addition, it includes several libraries to support build applications for machine learning [MLlib], stream processing [Spark Streaming], and graph processing [GraphX].

Apache Spark consists of Spark Core and a set of libraries. Spark Core is the heart of Apache Spark and it is responsible for providing distributed task transmission, scheduling, and I/O functionality. The Spark Core engine uses the concept of a Resilient Distributed Dataset (RDD) as its basic data type. The RDD is designed so it will hide most of the computational complexity from its users.

Spark is intelligent on the way it operates on data; data and partitions are aggregated across a server cluster, where it can then be computed and either moved to a different data store or run through an analytic model.

In summary Spark is a multi language , in memory , distributed compute engine. its a plug and play system.

Spark + data lake storage (hdfs or cloud or local storage) + resource manager(Yarn , Kubernetes, mesos, Standalone) ⇒ complete solution

Spark is just a replacement for MapReduce

Spark abstracts away the notion that you are writing code to run across cluster. You just run a SQL query or PySpark code and it just runs on cluster.

Key Features of Spark

Feature	Description
In-memory computation	Keeps intermediate data in memory - 100x faster than MapReduce
Unified engine	One platform for batch, streaming, ML, and graph
Ease of use	APIs in Python, Scala, Java, R
Resilient Distributed Dataset (RDD)	Core abstraction for fault-tolerant parallel computation
Lazy evaluation	Optimizes the execution DAG before running

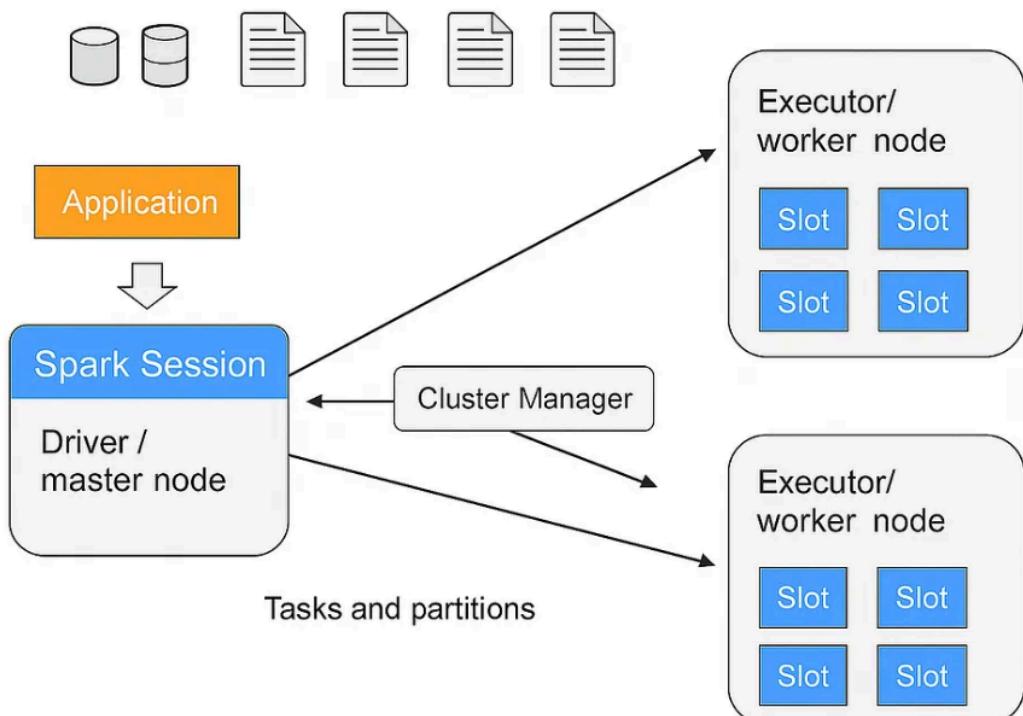
✓ Spark Advantages over MapReduce

- **In-Memory Processing**
 - Spark keeps intermediate data in memory using RDD/DataFrame caching.
 - Avoids repeated disk writes → dramatically faster. (only in case of wide transformations the data is written to disk)
- **Much Faster Execution**
 - Often **10x to 100x faster** than MapReduce for many workloads.
- **Easier to Develop**
 - Supports high-level APIs: **Python (PySpark), SQL, Scala, Java**
 - Shorter, cleaner code compared to MapReduce.

- **Supports Multiple Workloads**
 - Batch processing
 - Streaming (real time)
 - Machine learning (MLlib)
 - SQL analytics (Spark SQL)
 - Graph processing (GraphX)
- **Reuse of Data**
 - Spark can cache datasets in memory and reuse them for iterative workloads.

At very high level .:

Apache Spark Architecture



Let's understand what happens when we run our code in Apache Spark.
Assume we have a cluster where we have 2 executors and each executor has

4 CPU cores.

Step 1: Starting with SparkSession

Every Spark application starts with a `SparkSession`. You can think of this as the main gate or entry point to your cluster.

Whenever we read a file, run a transformation like `filter` or `groupBy`, or write the result to a file it's all done through the `SparkSession`.

```
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.appName("MyApp").getOrCreate()
```

Step 2: The Driver Program Controls Everything

When we run our Spark application, it gets picked up by something called the **Driver**.

The driver is the boss. It reads our code, understands the operations, and breaks the work into small parts like planning a task list.

Step 3: Work is Split into Tasks

Based on the code, Spark creates a **DAG (Directed Acyclic Graph)** behind the scenes. It's just a fancy way of saying: Spark figures out what needs to run first, second, and so on.

Then this DAG is split into **stages**, and each stage contains multiple **tasks**.

Think of tasks as the smallest piece of work like reading one part of a file or doing a simple transformation.

Step 4: Executors Do the Actual Work

Now the real work begins.

We have 2 **executors** (think of them as worker machines), and each one has **4 CPU cores**. So:

Each executor can run **4 tasks at the same time**. So in total, we can run **8 tasks in parallel** (2 executors × 4 cores).

| If Spark has more tasks than cores, it waits and runs them in the next round.

Step 5: Results are Collected or Written

Once all the tasks are done, Spark either:

Sends the final result back to the **driver** (if we use `.show()` or `.collect()`), or Writes the output to a file or database (if we use `.write()`).

What is a Cluster Resource Manager in Spark

A **cluster resource manager** is responsible for:

- Allocating CPU and memory
- Launching executors
- Monitoring running applications
- Sharing cluster resources across users

Spark **does not manage hardware resources by itself**.

It always runs **on top of a resource manager**.

Apache Spark Standalone (we will use this)

Spark's built-in resource manager with master and worker nodes.

Best for learning, demos, and small clusters.

Apache Hadoop YARN

Hadoop's resource manager that runs Spark in containers.

Used widely in on-prem and Hadoop based enterprises.

Kubernetes

Runs Spark driver and executors as containers in pods.

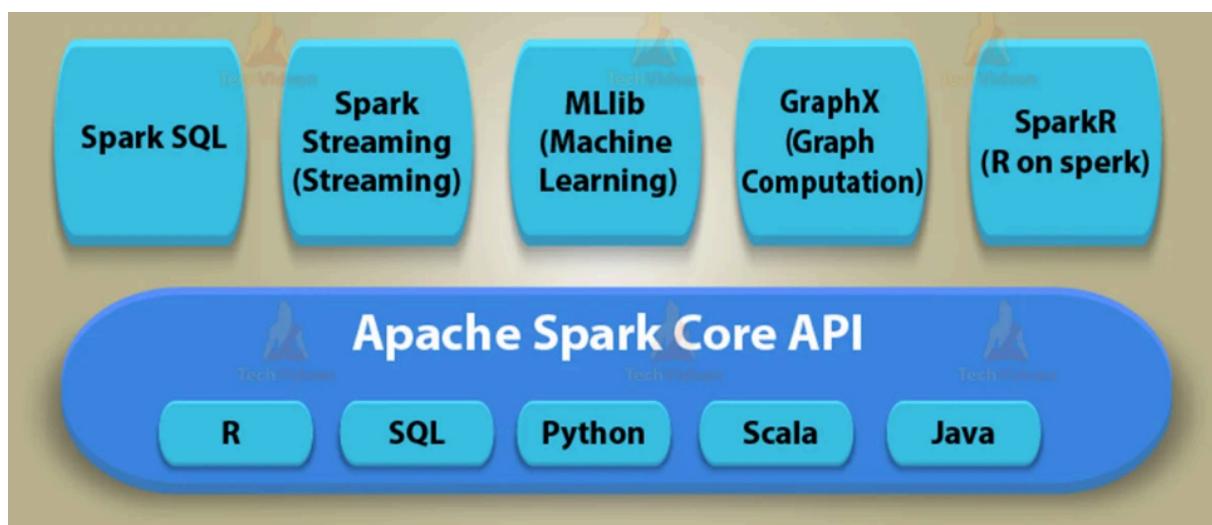
Preferred for cloud-native, elastic, Docker based workloads.

Apache Mesos

General purpose cluster manager that can run Spark and other frameworks.

Largely replaced by Kubernetes and rarely used today.

Spark Ecosystem Components



Component	Purpose
Spark Core	Basic functionality (RDD, scheduling, memory mgmt)
Spark SQL	Work with structured data using SQL syntax
Spark Streaming	Real-time data processing
MLlib	Machine learning algorithms
GraphX	Graph analytics (social networks etc.)
SparkR / PySpark	R and Python APIs for Spark

Spark core APIs :

RDD: Resilient Distributed Dataset

The fundamental, low-level API, an immutable, distributed collection of objects.

Its a bit harder to work with RDDs

Higher level APIs:

Spark SQL/ Data Frames → we will focus on these

and others as in image above

underneath the code is executed in RDD always.

High level project flow using spark :

1- load the data from source : hdfs / s3 / ADLS / GCS

2- do transformation → filter , aggregation , join

3- Write final results to target location (data lake / deltalake)

spark → in memory compute engine → cluster → manage -

databricks → managed spark / spark on cloud → optimizations → features

AWS → cloud , ec2 , storage → 10 tb, ec2 → aws glue → Athena

AZURE → cloud

snowflake → datawarehouse→ analytics platform

datalake → snowflake tables (orders,) → analytics

RDD → low level api → transform data

list , dictionaries, tuple , loop , functions

dataframe API

sql , python , spark ,data modelling ,warehosuing → databricks
snowflake

was data engineer

Kafka

disc - > spilled disc→ harddisc

compute ram + cpu