**Day 8 – Introduction to Apache Spark**

📆 – 16/06/2025

**What is Hadoop?**

**Apache Hadoop** is an open-source framework for distributed storage and batch processing of massive datasets. It was designed to scale up from a single server to

**Limitations of Hadoop Compared to Apache Spark**

| **Hadoop Limitation** | **Apache Spark Advantage** |
| --- | --- |
| **Disk-based Processing**: MapReduce reads/writes from disk, which is slow. | **In-Memory Computation**: Spark keeps intermediate data in memory, making it up to 100x faster. |
| **Complex Programming Model**: Requires writing extensive Java code. | **Easy APIs**: Provides Python, Scala, R, and Java APIs for quick development. |
| **No Real-time Support**: Only suitable for batch jobs. | **Real-time Processing**: Supports streaming data (Spark Streaming). |
| **Limited Libraries**: No built-in tools for ML or graph processing. | **Rich Ecosystem**: MLlib, Spark SQL, GraphX, Streaming. |
| **Latency in Iterative Processing**: Each step writes to disk. | **Fast Iterations**: Ideal for ML and graph algorithms. |
| **Rigid Fault Tolerance**: Relies on replication. | **RDD Lineage**: Recovers lost data through lineage and recomputation. |

💡Spark was **initiated** at **Berkeley** in 2009 to address Hadoop's limitations, and by 2013 became an Apache Top-Level Project.

**💡 Introduction to Apache Spark**

Apache Spark is a unified analytics engine designed for big data processing. It supports in-memory computation and provides APIs for batch, streaming, machine learning, and graph processing.

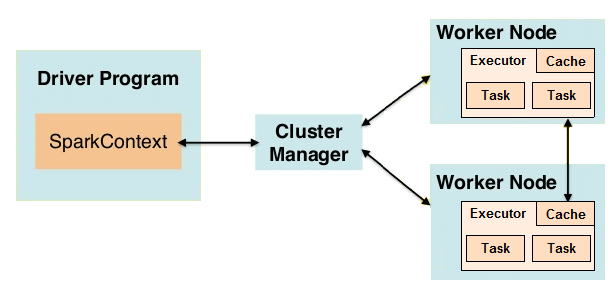
Spark is best suited for **real-time**, **iterative**, and **multi-stage** workflows.

**Features of Apache Spark**

* Spark is faster, more flexible, and more developer-friendly than Hadoop MapReduce.
* Supports various APIs and workloads.
* Integrates with Hadoop ecosystem (YARN, Hive).
* Ideal for data engineering pipelines and real-time analytics.

**Spark Architecture Overview**

Apache Spark follows a **master-slave** architecture with the following core components:

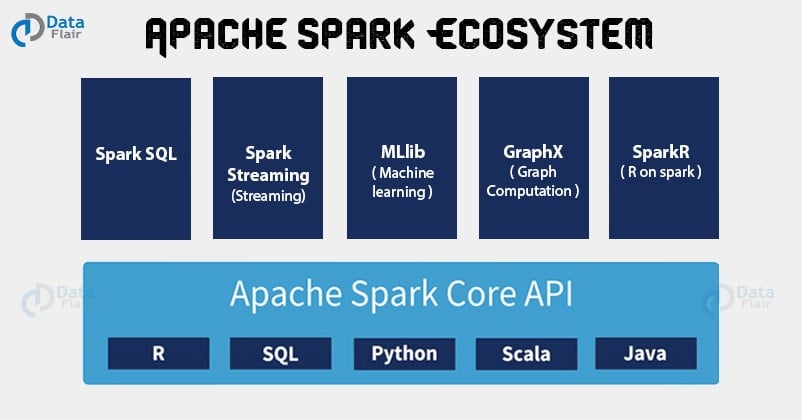
* **Driver Node**: Converts code into tasks and manages execution.
* **Worker Node**: Hosts executors that perform data processing.
* **Executor**: Processes data and stores results.
* **Cluster Manager**: Manages resources; can be YARN, Mesos, Kubernetes, or Standalone.

**Spark Deployment Modes**

| **Mode** | **Description** |
| --- | --- |
| **Client Mode** | Driver runs on the machine submitting the job. |
| **Cluster Mode** | Driver runs within the cluster—ideal for production workloads. |

**Spark Ecosystem and Components**

| **Component** | **Functionality** |
| --- | --- |
| **Spark Core** | Base engine, task scheduling, memory mgmt |
| **Spark SQL** | Structured data access using SQL |
| **Spark Streaming** | Real-time data processing |
| **MLlib** | Machine Learning library |
| **GraphX** | Graph computations |



**Spark Programming APIs**

| **API** | **Description** |
| --- | --- |
| **RDD (Resilient Distributed Dataset)** | Low-level API, immutable and distributed. |
| **DataFrame** | Tabular data abstraction with schema (columns). |
| **Dataset** | Combines RDD and DataFrame benefits (type-safe, JVM only). |
| **Spark SQL** | Enables SQL queries on structured data. |

**Spark Execution Flow: Transformations and Actions**

**Transformations (Lazy):**

* Examples: map(), filter(), flatMap()
* Do not execute immediately.
* Build a **DAG (Directed Acyclic Graph)**.

**Actions (Trigger Execution):**

* Examples: collect(), count(), reduce(), saveAsTextFile()
* Triggers actual computation.

**DAG & Lazy Evaluation**

Spark optimizes execution using DAGs. It applies **lazy evaluation**, deferring execution until an action is called. This improves performance and reduces computation.

**Spark Memory Management**

* **In-memory computation** enables Spark’s speed.
* Components:
  + **Execution Memory**: Temporary storage for shuffles, joins, etc.
  + **Storage Memory**: Caching data (RDD/DataFrame).
  + **User Memory**: For custom objects.
  + **Reserved Memory**: For internal tasks.

**Advantages of Spark**

* **High-speed Processing**: In-memory data processing via RDDs.
* **Real-time Capability**: Outperforms Hadoop for live data.
* **Fault Tolerant**: RDD lineage enables automatic recovery.
* **Unified Engine**: Handles batch, streaming, ML, graph.

**RDD = Resilient Distributed Dataset**

* Immutable
* Fault-tolerant
* Distributed by default

| **Tool** | **Purpose** |
| --- | --- |
| **YARN** | Resource manager; used by Spark as cluster manager |
| **Hive** | Data warehouse; supports SQL-like querying |
| **Zookeeper** | Coordination service for distributed systems |
| **Mahout** | ML on Hadoop (now largely replaced by MLlib in Spark) |

Here's a clear and concise **summary** of the article **"Apache Spark: Core Concepts, Architecture and Internals"** from Datastrophic, broken down into key sections and explained in simple terms:

**🔹Core Concept: RDD (Resilient Distributed Dataset)**

**RDD** is Spark's core abstraction for distributed, immutable datasets.  
Key properties:

* **Partitioned** across nodes.
* **Immutable** and **lazy** (transformations are delayed until an action is called).
* Supports **lineage** for fault tolerance (can recompute lost partitions).
* Includes APIs like:
  + getPartitions, getDependencies, compute, getPreferredLocations.

**Example**:

val data = sc.textFile("hdfs://...")

.map(\_.split(","))

.filter(...)

**🔹 RDD Operations**

| **Type** | **Examples** | **Purpose** |
| --- | --- | --- |
| **Transformations** | map, filter, groupBy | Create new RDDs |
| **Actions** | count, collect, reduce | Trigger job execution |
| **Persistence** | cache, persist | Store RDDs in memory/disk |
| **Checkpointing** | Save stable RDD state to HDFS |  |

**🔹 DAG (Directed Acyclic Graph)**

Spark constructs a DAG of transformations to understand data dependencies.

* **Narrow dependencies**: Data flows from one parent to one child partition (e.g., map).
* **Wide dependencies**: Involve shuffling data across partitions (e.g., reduceByKey).

DAG is divided into **stages**, each consisting of multiple **tasks**:

* **ShuffleMapStage** → writes intermediate data
* **ResultStage** → returns results to driver

**🔹 Shuffle in Spark**

Shuffling redistributes data between stages:

* **Shuffle Write**:
  + Sorts/partitions data and writes to disk.
  + Uses **sort shuffle** (default) or **hash shuffle**.
* **Shuffle Read**:
  + Fetches data blocks from other nodes and optionally sorts them.

**🔹Spark Components (at 10,000 ft)**

| **Component** | **Role** |
| --- | --- |
| **Spark Driver** | Manages job lifecycle, creates SparkContext |
| **Executors** | Run tasks, store intermediate data |
| **Cluster Manager** | Allocates resources (YARN, Mesos, Kubernetes, Standalone) |

**Subcomponents in Driver**:

* SparkContext: Interface to the cluster
* DAGScheduler: Breaks jobs into stages
* TaskScheduler: Sends tasks to executors
* SchedulerBackend: Connects to cluster manager
* BlockManager: Manages memory, disk storage for RDDs

**🔹Memory Management (Spark 1.6+)**

Executor memory is split as follows:

| **Region** | **Purpose** |
| --- | --- |
| **Execution Memory** | For shuffles, joins, aggregations |
| **Storage Memory** | For cached RDDs, broadcast vars |
| **User Memory** | For UDFs and metadata |
| **Reserved Memory** | For executor JVM processes |

Spark dynamically balances between execution and storage memory.

**🔹Execution Workflow Summary**

1. **User code** creates RDDs and transformations.
2. Spark builds **DAG** and breaks it into **stages**.
3. **TaskScheduler** assigns tasks to executors.
4. **Executors** execute tasks and shuffle data if needed.
5. Final **results** are collected or stored.