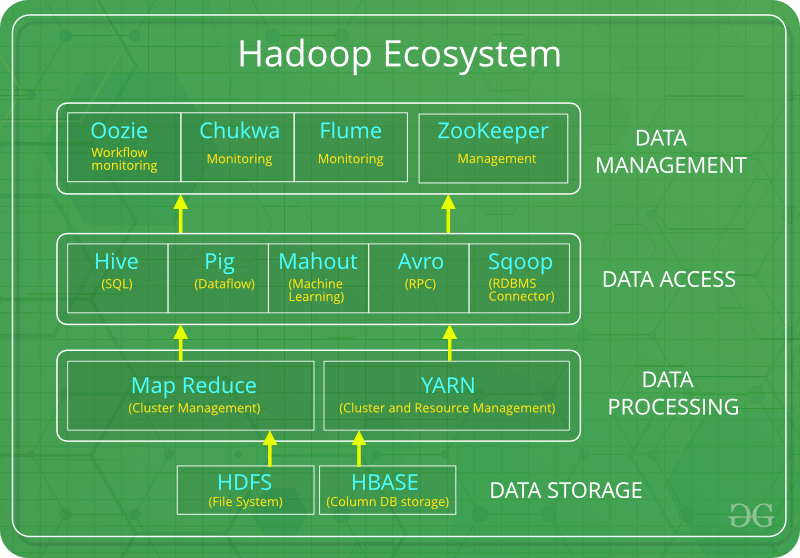
**Day 8 – (16-06-25)**

**Hadoop ECOSYSTEM**

What is the **Hadoop Ecosystem**?

* The **Hadoop Ecosystem** is a **collection of tools and frameworks** that work together to **store, process, manage, and analyze big data** in a distributed computing environment.
* Think of it like an **operating system for big data**. Just like your computer OS has different programs for file storage, editing, networking, etc., Hadoop has different tools for different big data tasks.



Core Components of Hadoop Ecosystem (With Examples):

**1. Storage Layer**

This is where data is stored.

**HDFS (Hadoop Distributed File System)**

* **Definition**: A **distributed file system** that stores huge files by splitting them into blocks and saving them across multiple machines.
* **Example**: If you upload a 100GB file, HDFS splits it into blocks (say, 128MB each) and stores them on different machines.
* **Purpose**: Handles **large-scale storage** reliably.

**2. Resource Management Layer**

**YARN (Yet Another Resource Negotiator)**

* **Definition**: A **resource management** and **job scheduling** system in Hadoop.
* **Example**: If multiple users submit data jobs, YARN decides **which job runs where and when**.
* **Purpose**: It allocates resources to different tasks in the cluster.

**3. Processing Layer**

Used to **analyze or process data** stored in HDFS.

**a. MapReduce**

* **Definition**: A programming model to **process large data sets** with parallel computation.
* **Example**: Count how many times each word appears in a huge text file.
* **Purpose**: Traditional and reliable batch processing.

**b. Apache Spark**

* **Definition**: A **faster processing engine** compared to MapReduce; supports in-memory computing.
* **Example**: Real-time log analysis.
* **Purpose**: For high-speed batch or streaming data processing.

**4. Data Access & Query Tools**

To make it easier to **read/write/query** big data without writing complex code.

**a. Hive**

* **Definition**: A **SQL-like tool** to run queries on Hadoop data.
* **Example**: SELECT \* FROM big\_sales\_data WHERE region = 'South';
* **Purpose**: Query big data using simple **SQL-style commands**.

**b. Pig**

* **Definition**: A scripting language for data analysis.
* **Example**: Count total sales grouped by product.
* **Purpose**: Easier than Java (used in MapReduce), used for transformation pipelines.

**c. HBase**

* **Definition**: A **NoSQL database** that works on top of HDFS.
* **Example**: Store sensor data with fast read/write access.
* **Purpose**: Real-time read/write access to large data sets.

**d. Sqoop**

* **Definition**: A tool to **transfer data between Hadoop and RDBMS** like MySQL.
* **Example**: Import customer data from MySQL to Hadoop.
* **Purpose**: Bridges traditional databases and Hadoop.

**e. Flume**

* **Definition**: Used to **collect and transfer streaming data** to HDFS.
* **Example**: Ingest logs from servers in real-time.
* **Purpose**: Handle real-time data ingestion.

**Other Useful Tools:**

| **Tool** | **Purpose** | **Example Use** |
| --- | --- | --- |
| **Zookeeper** | Coordination & configuration | Keeps track of nodes in the Hadoop cluster |
| **Oozie** | Workflow Scheduler | Run a MapReduce job every night at 1 AM |
| **Mahout** | Machine Learning | Build recommendation systems |
| **Ambari** | Hadoop Cluster Management (UI) | Monitor and manage all Hadoop services easily |

**Final Summary Table:**

| **Layer** | **Component** | **Description** |
| --- | --- | --- |
| **Storage** | HDFS | Stores large data across machines |
| **Resource Manager** | YARN | Manages jobs and resources |
| **Processing** | MapReduce | Traditional batch data processing |
|  | Spark | Fast in-memory processing |
| **Query/Access** | Hive, Pig | SQL or scripts for data analysis |
|  | HBase | NoSQL database on Hadoop |
|  | Sqoop, Flume | Import/export and data collection tools |
| **Support Tools** | Zookeeper, Oozie, Ambari | Monitoring, coordination, scheduling |

What is **Apache Spark**?

* Apache Spark is a **unified analytics engine** for **large-scale data processing**, with built-in modules for **batch processing, real-time streaming, machine learning, and graph processing**. It runs **100x faster** than Hadoop MapReduce in memory.
* **Apache Spark** is a **powerful open-source big data processing engine** used to **analyze huge amounts of data quickly**.

Hadoop vs Spark: Side-by-Side Table:

| **Feature** | **Hadoop (MapReduce)** | **Spark** |
| --- | --- | --- |
| **Basic Idea** | Processes data in **batches**, one step at a time | Processes data **in memory**, much faster |
| **Speed** | **Slower**, due to disk read/write between tasks | **Faster**, as it keeps data in **memory (RAM)** |
| **Ease of Use** | **Complex**, mostly written in Java | Easier with APIs in **Python, Scala, Java, R** |
| **Processing Type** | **Batch processing only** | Supports **batch**, **stream**, **ML**, **graph** |
| **Fault Tolerance** | High – uses **HDFS replication** | High – uses **RDD lineage** |
| **Use Case Example** | Monthly report generation | Real-time Twitter sentiment analysis |
| **Component** | Based on **MapReduce** programming model | Based on **RDD** (Resilient Distributed Datasets) |
| **Speed** | Data is read from disk every time | Data is cached in memory (RAM) for fast access |
| **Resource Management** | YARN (Yet Another Resource Negotiator) | Can use YARN, Mesos, or its own cluster manager |
| **Cost Efficiency** | Cheaper for huge batch jobs on low RAM systems | Needs more RAM → can be **costlier** |

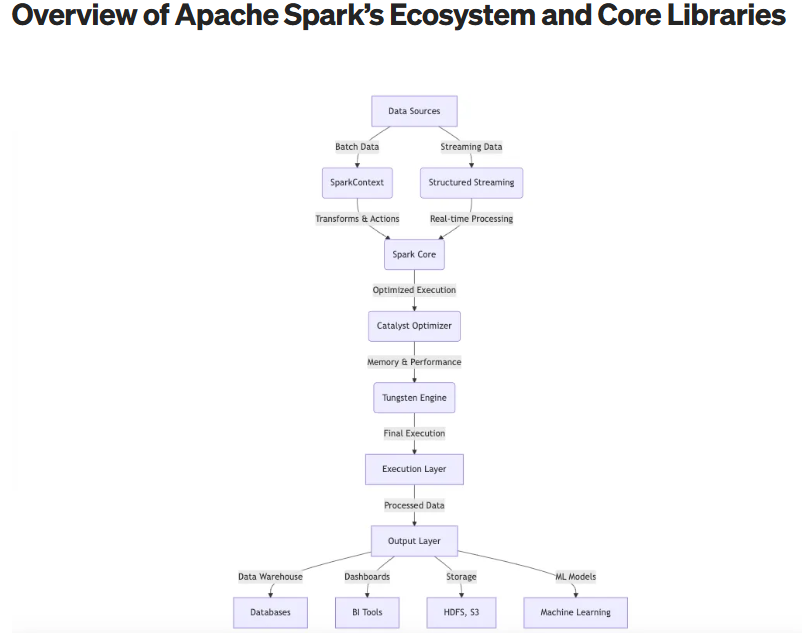
**When to Use What?**

| **Situation** | **Choose** |
| --- | --- |
| Very large batch jobs, limited RAM | **Hadoop** |
| Real-time analytics or fast processing | **Spark** |
| Machine Learning or Stream processing | **Spark** |
| Lower cost, disk-based | **Hadoop** |

**Why Use Spark Instead of MapReduce?**

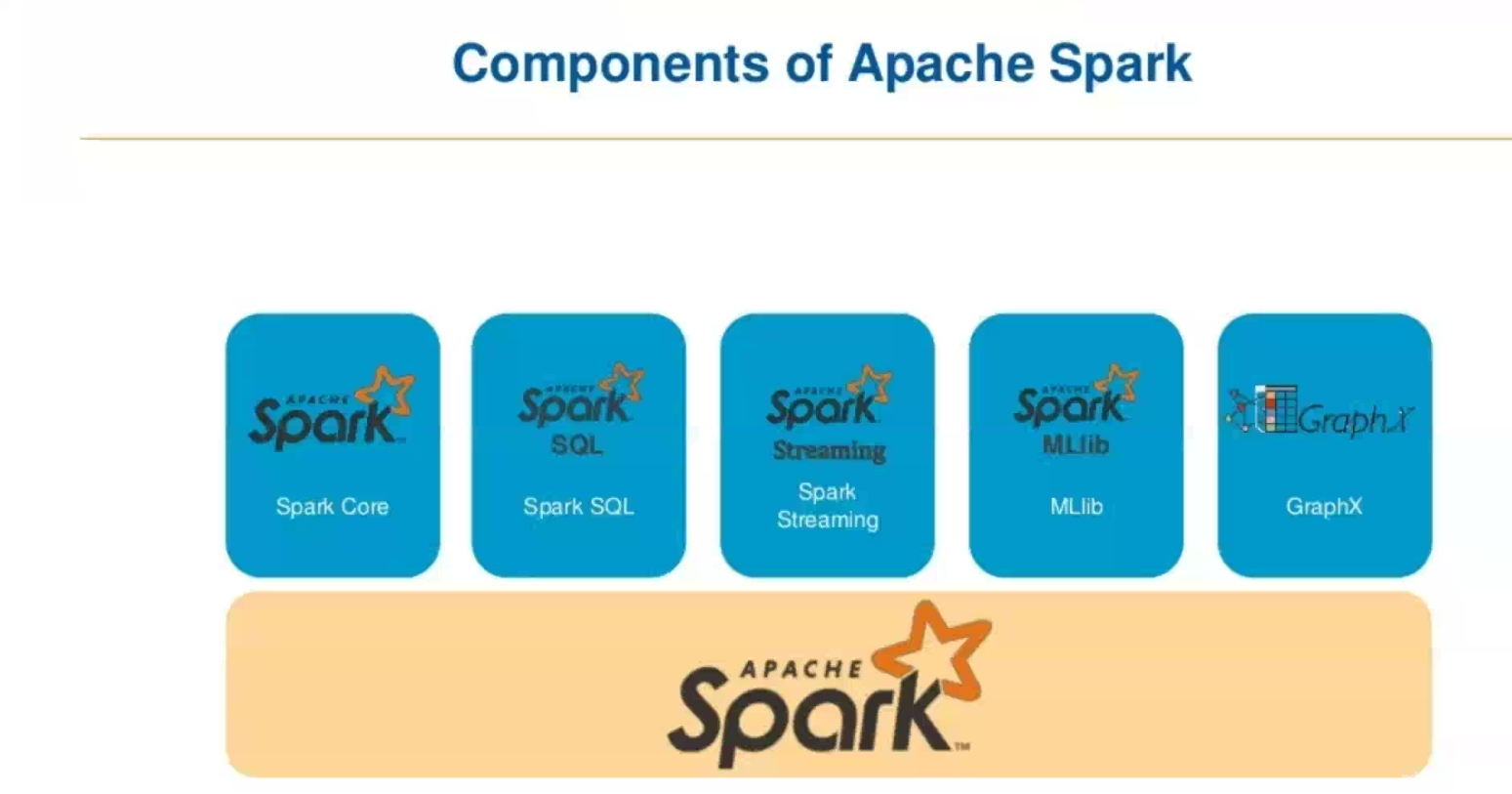
| **Feature** | **Apache Spark** | **Hadoop MapReduce** |
| --- | --- | --- |
| **Speed** | **Very fast (in-memory processing)** | **Slower (disk-based)** |
| **Ease of Use** | **Easy APIs in Python, Java, Scala** | **More complex (Java code needed)** |
| **Real-time Support** | **Yes (via Spark Streaming)** | **No** |
| **Use Cases** | **Batch + Streaming + ML + Graph** | **Mostly batch** |

**Overview of Apache Spark’s Ecosystem and Core Libraries:**



**Components of Apache Spark:**

| **Component** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| **Spark Core** | Base engine for Spark functionality | Reads and processes data |
| **Spark SQL** | Use SQL queries on big data | SELECT \* FROM sales WHERE region='East' |
| **Spark Streaming** | Real-time data processing | Analyze live website logs |
| **MLlib** | Machine learning library | Build recommendation engines |
| **GraphX** | Graph processing | Analyze social networks |



Apache Spark Features:

| **Feature** | **Simple Explanation** | **Technical Explanation** |
| --- | --- | --- |
| **In-Memory Computing** | Keeps data in **RAM** to avoid slow hard disk read/write | Uses **RDDs** to cache intermediate results, reducing I/O and speeding up computations |
| **High Speed** | 10–100x faster than Hadoop for many tasks | Optimized DAG execution engine + in-memory processing |
| **Real-Time Processing** | Can handle **live data streams** like Twitter or sensors | Supports **Spark Streaming** for real-time data with micro-batches |
| **Fault Tolerance** | Automatically recovers from crashes or errors | RDDs track lineage to **recompute lost data** instead of re-running whole job |
| **Machine Learning** | Built-in support for ML tasks like classification, clustering | Includes **MLlib** – scalable machine learning library |
| **Multi-language Support** | Can write code in **Python, Java, Scala, R** | APIs available for all major programming languages |
| **Easy Integration** | Works well with other tools like Hadoop, Hive, Kafka, Cassandra | Supports Hadoop HDFS, YARN, Hive, JDBC, Kafka, and more |
| **Graph Processing** | Analyze data with connections (e.g., social networks) | Built-in **GraphX** engine for graph-parallel computations |
| **DataFrame & SQL** | Query big data using **SQL-like syntax** | Supports **Spark SQL** and DataFrames (structured data processing) |
| **Flexible Deployment** | Can run on **laptop**, **cluster**, or in **cloud** | Works with YARN, Mesos, Kubernetes, or Spark Standalone Cluster |
| **Modular Architecture** | Choose what you need: streaming, SQL, ML, etc. | Components: Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX |

**Spark Core:**

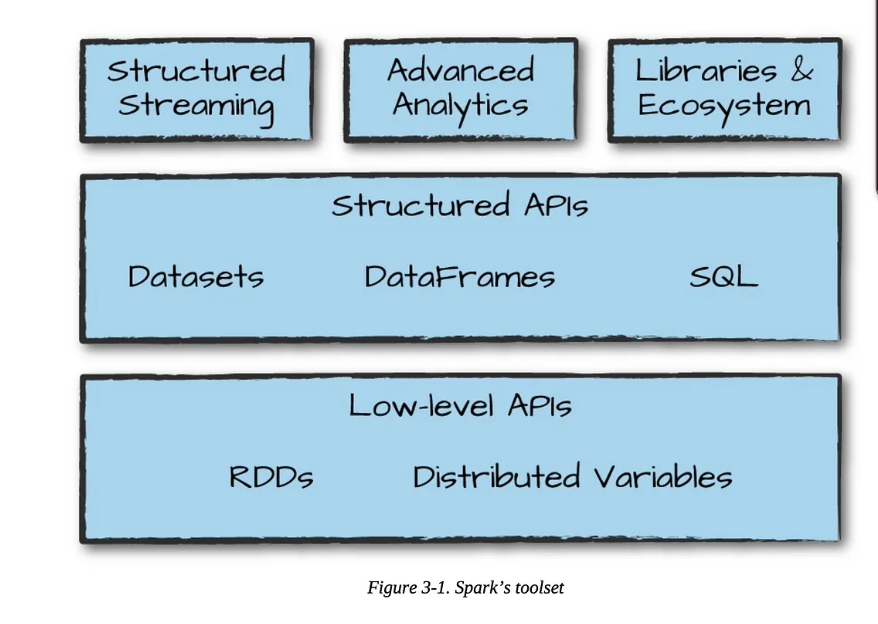
* **Spark Core** is the **base engine** of Apache Spark.
* It handles basic tasks like **memory management**, **task scheduling**, **fault recovery**, and **interacting with storage**.

A screenshot of a computer program

AI-generated content may be incorrect.

**Main Responsibilities of Spark Core,**

| **Feature** | **Explanation** |
| --- | --- |
| **Task Scheduling** | Decides how to run your jobs on clusters or nodes |
| **Memory Management** | Efficiently uses **RAM** to cache data and avoid disk usage |
| **Fault Tolerance** | If a worker fails, Spark Core can **recalculate lost data** from the lineage |
| **Distributed Computing** | Splits your job across many machines and coordinates the results |
| **Storage Interaction** | Connects with **HDFS, S3, local file systems, HBase, Cassandra**, etc. |
| **Support for RDD** | Provides API to create and process **RDDs (Resilient Distributed Datasets)** |



| **Feature** | **Description** |
| --- | --- |
| Core engine | Manages memory, scheduling, execution, and fault tolerance |
| RDD support | Immutable distributed collections with transformations/actions |
| Storage integration | Works with HDFS, S3, Cassandra, HBase, local files |
| Execution environment | Cluster management and task scheduling |
| Basis for other modules | Required for Spark SQL, MLlib, GraphX, Streaming |

RDD (**Resilient Distributed Dataset**):

* RDD is an **immutable**, **distributed collection** of objects that can be processed in **parallel**.  
  It offers **fault tolerance** using **lineage** (i.e., how it was created).
* RDD is like a **big list of data** that is **split across multiple machines**, and **Spark can work on it in parallel**.

A screenshot of a computer

AI-generated content may be incorrect.

**Key Features of RDD,**

| **Feature** | **Explanation** |
| --- | --- |
| **Resilient** | Recovers automatically from failures using lineage info |
| **Distributed** | Spread across many machines or nodes |
| **Immutable** | Once created, it **cannot be changed**; but you can create **new RDDs** |
| **In-Memory** | Can **store data in RAM** for faster access |
| **Lazy Evaluation** | Nothing runs until you call an action (like .collect()) |
| **Parallel Processing** | Spark splits work across cores and nodes |

**Types of Operations on RDD**

| **Type** | **Example** | **Purpose** |
| --- | --- | --- |
| **Transformations** | map(), filter(), flatMap() | Creates a **new RDD** |
| **Actions** | collect(), count(), reduce() | Triggers **actual computation** |

**When to Use RDD?**

* You need **fine-grained control** over your data and operations.
* You want to handle **unstructured data** (like logs, text).
* You’re okay with **writing more code** for **more control**.

How RDD works,

Resilient Distributed Datasets are resilient and distributed. That means:

Resilient:

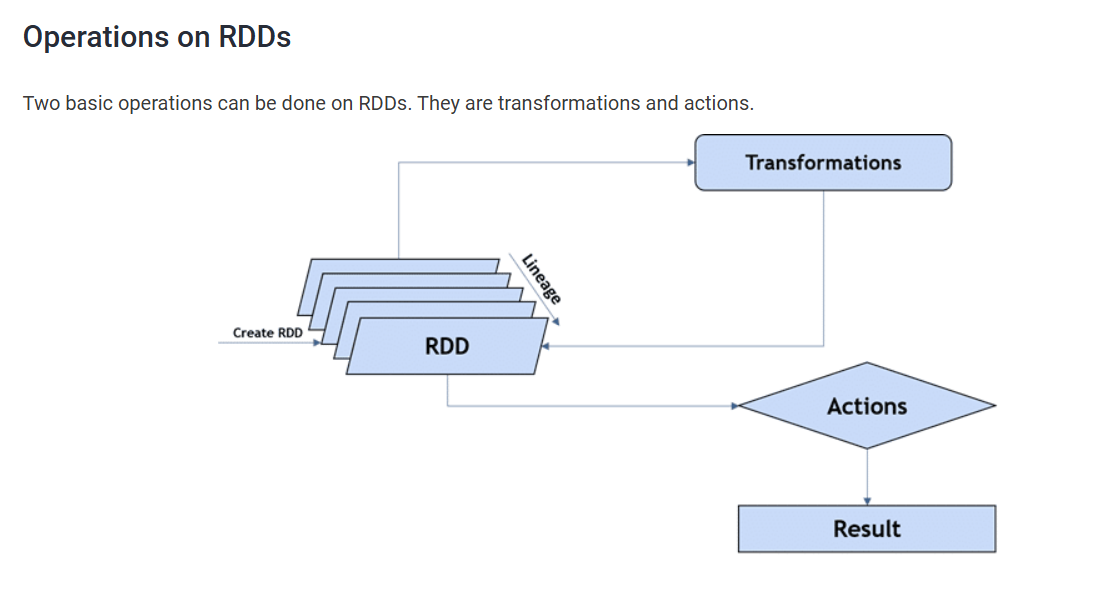
RDDs are called "resilient" because they track [data lineage](https://www.ibm.com/think/topics/data-lineage) information so that lost data can be rebuilt if there is a failure, making RDDs highly fault-tolerant.

As an example of this [data resilience](https://www.ibm.com/think/topics/data-resiliency), consider an executor core that is lost during the processing of an RDD partition. The driver would detect that failure, and that partition would be reassigned to a different executor core.

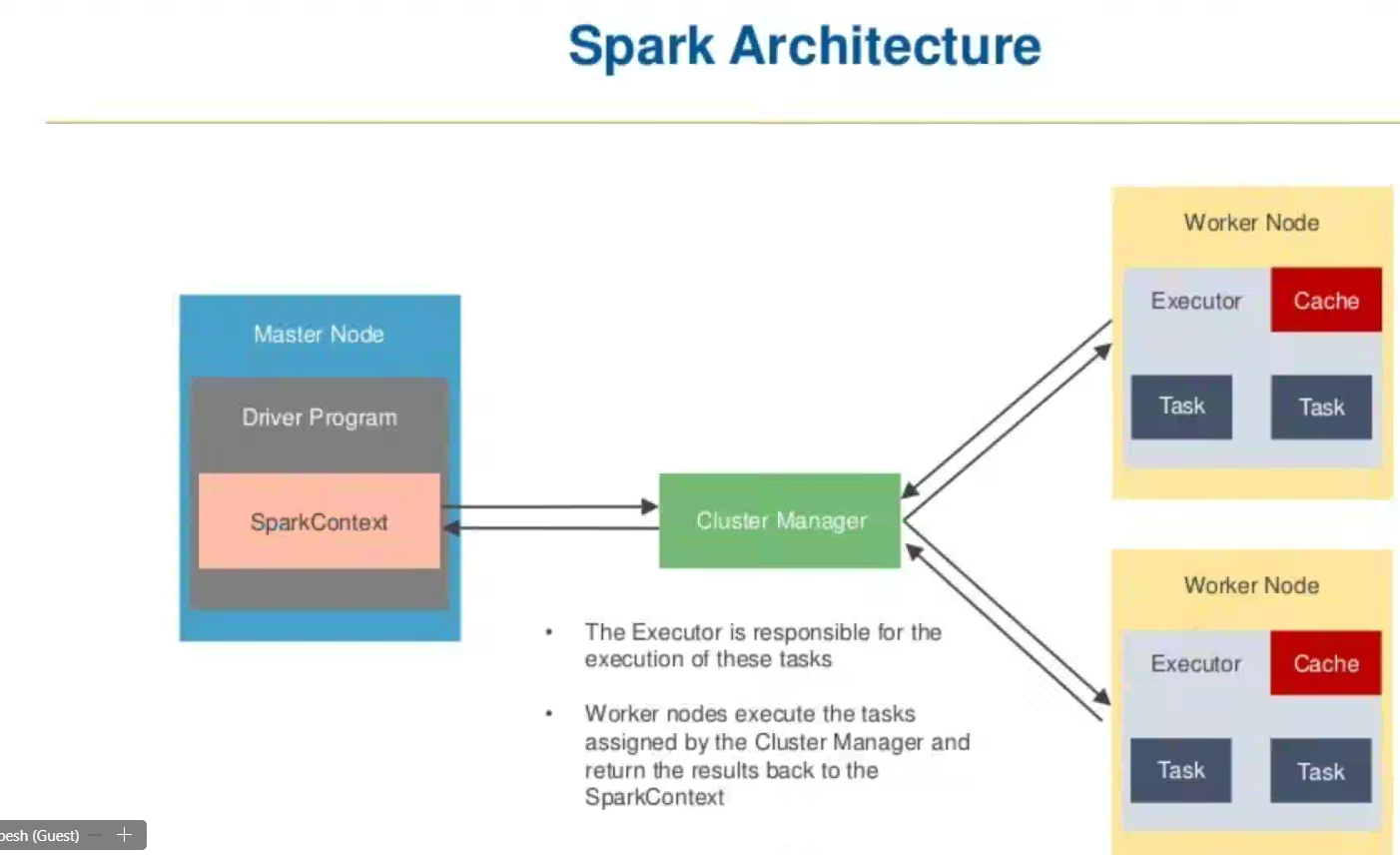
Distributed:

RDDs are called "distributed" because they are split into smaller groups of data that can be distributed to different compute nodes and processed simultaneously.

In addition to these 2 core characteristics, RDD has other features that contribute to its importance and operations in Spark.



**Basic Spark Architecture:**



Apache Spark follows a **master-slave architecture** where the **Driver Program** (on the Master node) coordinates with **Executors** (on Worker nodes) to execute tasks in a distributed fashion. The **Cluster Manager** acts as a resource allocator.

**Main Components**

**1. Master Node**

* **Driver Program**:
  + This is the main application process.
  + It creates the **SparkContext** and handles job scheduling.
  + Divides the job into **tasks** and sends them to executors on worker nodes.
* **SparkContext**:
  + Acts as the entry point to any Spark application.
  + Connects to the **Cluster Manager** to request resources.
  + Communicates with all executors.

**2. Cluster Manager**

* Manages resources across all applications.
* Types of Cluster Managers supported by Spark:
  + **Standalone** (Spark’s own)
  + **YARN** (from Hadoop)
  + **Apache Mesos**
  + **Kubernetes**
* Responsibilities:
  + Allocates resources to SparkContext.
  + Launches **executors** on worker nodes.

**3. Worker Nodes**

Each worker node runs one or more **executors**.

* **Executor**:
  + A process launched for a Spark application.
  + Executes **tasks** assigned by the driver.
  + Stores data in memory (caching).
  + Sends results back to the driver.
* **Cache**:
  + Spark executors can **cache data in memory**.
  + This enables faster repeated computations.
* **Tasks**:
  + Smallest unit of work.
  + Spark breaks jobs into stages, and stages into tasks.
  + Tasks run in parallel across the executors.

**How Spark Works (Step-by-Step)**

1. **SparkContext** is created in the driver program.
2. It connects to the **Cluster Manager**.
3. The Cluster Manager allocates resources on **Worker Nodes**.
4. **Executors** are launched on the worker nodes.
5. SparkContext sends **tasks** to executors.
6. Executors run the tasks and return results to the driver.
7. If caching is used, executors store intermediate results in memory for reuse.

**Advantages of This Architecture**

* **Distributed processing** allows Spark to handle large-scale data efficiently.
* **In-memory computation** via executors provides high performance.
* **Fault tolerance** through task re-execution using lineage info.
* **Scalability** by adding more worker nodes to the cluster.

**Example Scenario**

Imagine you're analyzing a large CSV file with Spark:

* The file is loaded by the driver using SparkContext.
* Spark splits the data into partitions.
* Each partition is assigned as a **task** to worker nodes.
* Executors process the tasks in parallel.
* Results are sent back to the driver or cached for reuse.

Spark Context:

* SparkContext is the **entry point** and **main control gateway** for any Spark application.
* It **connects your application to the Spark cluster** and allows you to create RDDs, accumulate variables, and interact with all Spark functionalities.

**Parameters in SparkContext:**

| **Parameter** | **Description** |
| --- | --- |
| master | The cluster URL (e.g., "local", "yarn", "spark://") |
| appName | Name of your Spark application |
| config | Optional SparkConf object for custom settings |

**What is the Task Scheduler?**

The **Task Scheduler** in Apache Spark is a key internal component responsible for:

* **Distributing tasks** (the smallest units of execution) across executors on worker nodes.
* **Managing task execution** based on data locality and resource availability.
* **Monitoring** the status of each task and **handling failures**.

**Responsibilities of Task Scheduler,**

| **Responsibility** | **Description** |
| --- | --- |
| **Task Allocation** | Assigns tasks to executors based on available resources and data location |
| **Data Locality Optimization** | Tries to run tasks on nodes where the required data is already present |
| **Monitoring Task Execution** | Tracks task progress, retries failed tasks |
| **Failure Recovery** | If a task fails, it re-executes it on another executor |
| **Communication** | Sends updates (success/failure) back to the **DAGScheduler** and driver |

**Summary Execution Flow (Simple View),**

1. Spark application is submitted → SparkContext is created.
2. RDD/DataFrame operations → DAG of stages.
3. DAG Scheduler creates task sets per stage.
4. **Task Scheduler**:
   * Picks executors
   * Assigns tasks
   * Monitors task progress
5. Executors complete tasks → send results back to driver.

Simple Diagram:

Driver

|

DAG Scheduler

|

Task Scheduler  
|

Executors

A diagram of a computer

AI-generated content may be incorrect.

A diagram of a new recap

AI-generated content may be incorrect.

**What is a DAG?**

A **DAG (Directed Acyclic Graph)** in Spark is a graph structure that represents the **logical execution plan** of a Spark job.  
It consists of **vertices (nodes)** and **edges (connections)**:

* **Vertices**: RDDs or stages.
* **Edges**: Operations or dependencies between them.

“Directed” means operations flow in one direction, and “Acyclic” means there are **no loops**.

**Why DAG in Spark?**

In older systems like Hadoop MapReduce:

* Every job was broken into map and reduce phases.
* It used **intermediate disk I/O** and had limited optimization.

**Spark DAG eliminates these limitations** by:

* Combining operations into a **chain of transformations**.
* **Optimizing execution** before it runs.
* Reducing unnecessary shuffling or disk access.

**DAG Execution Flow in Spark,**

1. **User Code**: You write transformations (e.g., map, filter, join, etc.).
2. **Lazy Evaluation**: Spark doesn't execute immediately. It builds a DAG.
3. **DAG Scheduler**:
   * Breaks the DAG into **stages** (based on shuffle boundaries).
   * Each stage is divided into **tasks** (based on data partitions).
4. **Tasks** are scheduled and sent to **executors** for parallel execution.

**Benefits of DAG in Spark**

* **Optimized execution plan** before running jobs
* **Better fault tolerance** (via lineage)
* **Efficient task scheduling**
* **Minimized disk I/O** and **reduced data shuffling**

Simple Visual Summary,

[Stage 1]

RDD1 --map--> RDD2 --flatMap--> RDD3

↓

shuffle (wide)

↓

[Stage 2]

RDD4 --reduceByKey--> FinalRDD

Summary of - [**Apache Spark: core concepts, architecture and internals**](https://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/)

**1. Introduction to Apache Spark,**

Apache Spark is a distributed data processing engine designed for speed and general-purpose analytics. It supports in-memory computation and is compatible with cluster managers like YARN, Mesos, and its own Standalone manager. It works with various storage systems such as HDFS, Cassandra, and Amazon S3, and supports APIs in Scala, Python, Java, and R.

**Spark Architecture:**

**[ Driver Program ]**

**|**

**[ SparkContext ]**

**|**

**[ DAG Scheduler ]**

**|**

**[ Task Scheduler ]**

**|**

**[ Scheduler Backend ] <----> [ Cluster Manager ]**

**|**

**[ Executors (on Worker Nodes) ]**

A diagram of a computer

AI-generated content may be incorrect.

A diagram of a new recap

AI-generated content may be incorrect.

**2. Resilient Distributed Datasets (RDDs)**

**What is an RDD?**

* An RDD (Resilient Distributed Dataset) is an immutable, distributed collection of objects partitioned across nodes in a cluster.
* It supports lazy evaluation and tracks lineage to recover lost data in case of failure.

**Key Methods in RDD:**

1. getPartitions
2. getDependencies
3. compute
4. getPreferredLocations
5. partitioner (optional)

**RDD Operations:**

* Transformations: Lazy operations that create new RDDs (e.g., map, filter, reduceByKey)
* Actions: Trigger execution (e.g., collect, count)
* Persistence: Allows caching of RDDs in memory/disk (cache(), persist())

**RDD Lineage Example:**

RDD1 = sc.textFile("file.txt")

RDD2 = RDD1.map(...)

RDD3 = RDD2.filter(...)

RDD4 = RDD3.reduceByKey(...)

RDD4 maintains the lineage from RDD3 → RDD2 → RDD1.

**3. Directed Acyclic Graph (DAG) and Execution,**

**DAG Basics:**

* A DAG represents the sequence of computations as nodes (RDDs) and edges (transformations).
* Spark constructs a DAG when you define transformations, which is only executed upon an action.

**Dependency Types:**

* **Narrow Dependency**: Child RDD depends on one parent partition (e.g., map, filter)
* **Wide Dependency**: Child RDD depends on multiple parent partitions and requires a **shuffle** (e.g., groupByKey)

**Stage Formation:**

* DAG is split into **stages** at shuffle boundaries.
  + **ShuffleMapStage**: Involves shuffle write operations.
  + **ResultStage**: Returns data to the driver.

**Task Types:**

* **ShuffleMapTask**: Writes intermediate shuffle data to disk.
* **ResultTask**: Computes and returns final result to driver.

**DAG and Stages:**

Job

├── Stage 1 (map, narrow)

│ ├── Task 1

│ ├── Task 2

└── Stage 2 (reduce, shuffle)

├── Task 1

├── Task 2

**4. Shuffle Process,**

**Shuffle Write:**

* Map tasks write data for each reduce partition.
* Involves writing M × R output files (M = map tasks, R = reduce tasks).
* Hash-based or sort-based shuffle is used. Sort-based shuffle is the default since Spark 1.2.

**Shuffle Read:**

* Reduce tasks fetch, optionally sort, and aggregate data from all mappers.

**5. Spark Architecture**

**Key Components:**

**1. Driver Program:**

* Hosts the SparkContext.
* Coordinates the overall execution.
* Breaks jobs into tasks and stages.

**2. SparkContext:**

* Entry point for Spark application.
* Manages configuration, job submission, and connection to cluster manager.

**3. Cluster Manager:**

* Allocates resources on the cluster (YARN, Mesos, or Standalone).

**4. Executors:**

* JVM processes launched on worker nodes.
* Run tasks and store data for in-memory processing.

**6. Spark Internal Execution Flow**

1. **User Code**: Written using Spark APIs to create a DAG of RDDs.
2. **SparkContext**: Initializes Spark and communicates with the cluster manager.
3. **DAGScheduler**:
   * Divides the job into stages based on shuffle boundaries.
   * Maintains stage dependencies.
4. **TaskScheduler**:
   * Schedules individual tasks to be run on executors.
   * Handles retry logic and locality-aware scheduling.
5. **SchedulerBackend**:
   * Integrates with the underlying cluster manager to launch executors and tasks.
6. **Executors**:
   * Execute tasks.
   * Cache RDDs.
   * Handle data shuffle and I/O.

**7. Memory Management in Spark (Spark 1.6+)**

Spark divides the JVM heap into the following regions:

1. **Execution Memory**: Used for runtime operations like shuffles, joins, and aggregations.
2. **Storage Memory**: Used for caching RDDs and broadcast variables. It can borrow from execution memory if needed.
3. **User Memory**: Used by Spark internal metadata and user-defined objects.
4. **Reserved Memory**: A fixed amount (300 MB by default) reserved to prevent JVM crashes

**8. Summary Table,**

| **Component** | **Description** |
| --- | --- |
| **RDD** | Immutable dataset, supports lazy transformations and lineage tracking. |
| **DAG Scheduler** | Splits a job into stages and identifies shuffle boundaries. |
| **Task Scheduler** | Schedules and executes tasks on executors. |
| **Executors** | Run tasks and store data on worker nodes. |
| **Driver Program** | Orchestrates the application and manages SparkContext. |
| **Cluster Manager** | Allocates resources to executors across nodes. |
| **Shuffle** | Redistributes data across partitions; requires disk and network I/O. |
| **Memory Manager** | Manages Spark memory into execution, storage, and user memory regions. |

**Kubernetes: (Own interest research not covered )**

**1. What is Kubernetes?**

* Kubernetes (often abbreviated as **K8s**) is an **open-source container orchestration platform**. It automates the **deployment**, **scaling**, **management**, and **operation** of containerized applications (like Docker containers) across a cluster of machines.
* Kubernetes was originally developed by **Google** and is now maintained by the **Cloud Native Computing Foundation (CNCF)**.
* **Kubernetes** is the industry standard for managing containers at scale.
* It automates infrastructure operations like deployment, scaling, and recovery.
* The architecture is based on **Control Plane** and **Worker Nodes**.
* It supports a wide range of production-grade features like auto-scaling, load balancing, and service discovery.

**2. Why Use Kubernetes?**

Without Kubernetes:

* You have to **manually deploy** containers on each server.
* Scaling, load balancing, and failure recovery are **manual and error-prone**.

With Kubernetes:

* **Auto-scaling** of applications.
* **Self-healing** (restarts failed containers).
* **Service discovery and load balancing**.
* **Declarative configuration** using YAML/JSON files.
* **Rolling updates and rollbacks**.

**3. Where is Kubernetes Used?**

Kubernetes is widely used in:

* **Cloud-native applications**
* **Microservices architecture**
* **DevOps pipelines**
* **Big Data platforms** (like Spark on Kubernetes)
* **Machine Learning pipelines**
* **Hybrid cloud/multi-cloud deployments**

Major companies using Kubernetes: Google, Netflix, Amazon, Spotify, Reddit, Pinterest, and thousands of others.

**4. Kubernetes Architecture**

Kubernetes uses a **master-worker** architecture:

**Cluster = Control Plane (Master) + Worker Nodes**

A screenshot of a computer

AI-generated content may be incorrect.

**Control Plane (Master Components)**

1. **API Server (kube-apiserver)**
   * Entry point for all REST commands.
   * Clients, kubectl, and other components communicate through this.
2. **etcd**
   * Distributed key-value store.
   * Stores the entire state of the cluster (like a database).
3. **Controller Manager**
   * Watches the cluster state and makes changes to reach the desired state.
   * Example: Node Controller, Replication Controller.
4. **Scheduler**
   * Assigns Pods to Nodes based on resource needs and policies.

**Worker Node Components**

1. **kubelet**
   * Agent that runs on every worker node.
   * Communicates with the control plane to manage containers in pods.
2. **kube-proxy**
   * Handles networking and load-balancing.
   * Forwards traffic to the correct pod in a node.
3. **Container Runtime**
   * Runs containers (e.g., Docker, containerd, CRI-O).
4. **Pods**
   * Smallest unit in Kubernetes.
   * A pod can contain one or more tightly coupled containers.

**Kubernetes Workflow:**

1. **Developer** writes a YAML configuration file describing the app.
2. File is submitted to **API Server**.
3. API Server stores desired state in **etcd**.
4. **Scheduler** picks the best node to run the pod.
5. **kubelet** on that node pulls the container image and runs it.
6. **kube-proxy** ensures the pod is network-accessible.
7. If the pod crashes, **Controller Manager** ensures it is restarted.

**Core Kubernetes Concepts,**

| **Concept** | **Description** |
| --- | --- |
| **Pod** | Smallest unit in Kubernetes; wraps one or more containers |
| **Service** | Defines a network policy and load-balancer for pods |
| **Deployment** | Manages rolling updates, replica scaling, self-healing |
| **Namespace** | Isolated environments within a single cluster |
| **Node** | Physical or virtual machine where pods run |
| **Cluster** | Group of nodes managed by Kubernetes |

**Kubernetes in Big Data and Spark,**

Kubernetes can run **Apache Spark** jobs using the native Kubernetes scheduler.

Benefits include:

* Dynamically launch Spark executors as pods
* Better resource isolation
* Native integration with cloud providers (AWS, GCP, Azure)
* Easier DevOps integration with CI/CD pipelines