Spark

Hadoop

Hadoop is a software framework that allows you to store and analyse large amounts of data.

Hadoop is a highly scalable, open-source, distributed computing platform that allows you to store and process large amounts of data. It is used to store and analyze data from a variety of sources, including databases, web servers, and file systems.

Components of Hadoop

HDFS (Hadoop Distributed File System)

Stores large volumes of data across multiple machines. Fault-tolerant and scalable.

HDFS is designed in such a way that it believes more in storing the data in a large chunk of blocks rather than storing small data blocks.

How Data is Stored

1. A file is **split into blocks** (default: 128MB or 64MB).
2. Each block is **replicated** across multiple DataNodes (default replication factor: 3).
3. NameNode keeps a **metadata table** of which block is stored where.
4. DataNodes store the **actual data**.

Suppose a file of 300MB is uploaded to HDFS with a block size of 128MB and replication factor of 3.

The file will be split into 3 blocks:

- Block 1 → 128MB
- Block 2 → 128MB
- Block 3 → 44MB

- Each block is replicated to 3 different nodes, so total storage used = 300MB × 3 = 900MB

Key Features of HDFS

- **Fault Tolerance**: HDFS replicates data blocks across multiple nodes, ensuring data availability even in the event of hardware failures.
- **Scalability**: It supports horizontal scaling, allowing the addition of nodes to handle increasing data volumes.
- **Data Locality**: Computation is moved closer to where the data resides, reducing network congestion and improving efficiency.
- **Cost-Effectiveness**: HDFS uses inexpensive commodity hardware and is open-source, eliminating licensing costs.

Apache Spark

Apache Spark is an open-source, distributed computing system designed for fast and general-purpose big data processing.

At its core, spark a distributed processing framework designed to handle large dataset across multiple machines (or nodes). Spark's key strength lies in its speed — it processes data up to 100 times faster than traditional systems like Hadoop MapReduce by storing intermediate data in memory and processing tasks in parallel.

Spark divides large datasets into smaller pieces and processes them simultaneously across multiple machines, ensuring fast, scalable data processing.

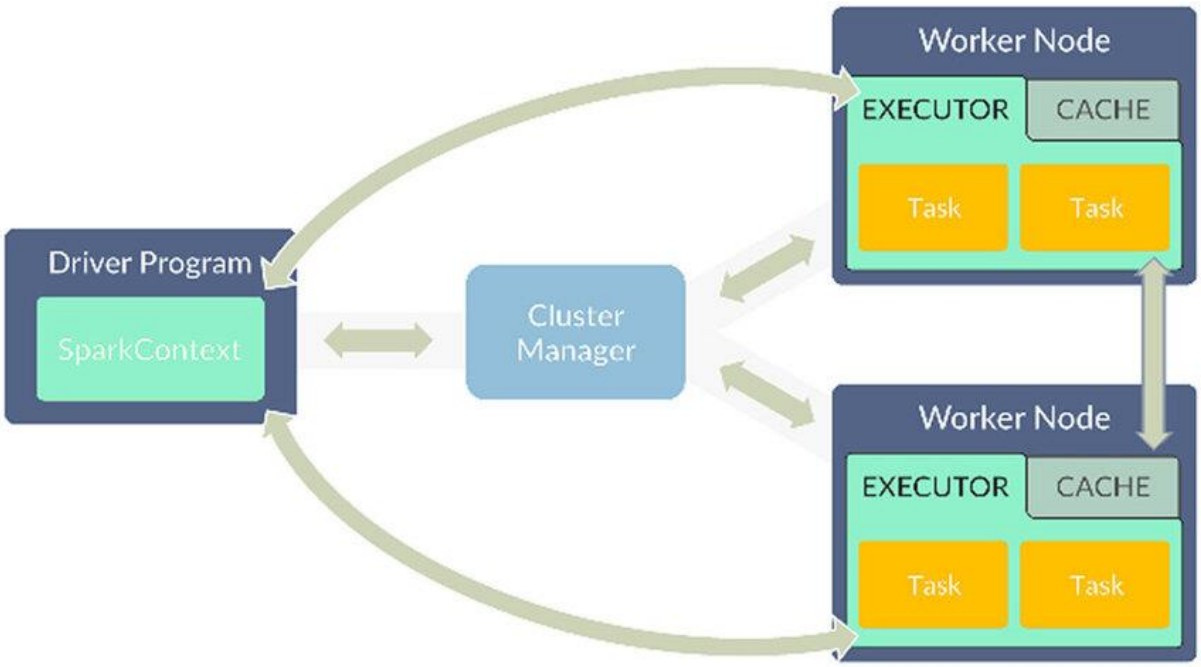
Key Features of Apache Spark

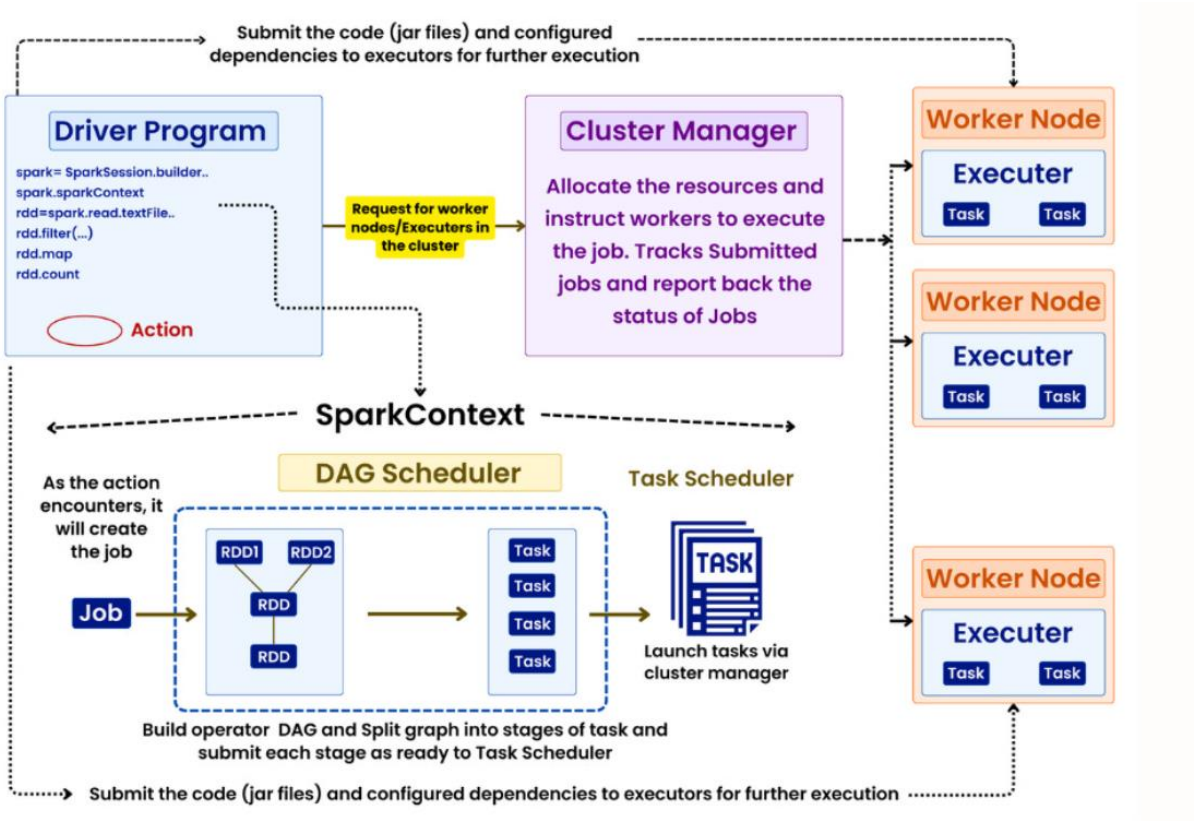
- Speed: Executes workloads 100x faster than Hadoop MapReduce for in-memory data, and 10x faster on disk.
- Ease of Use: Provides high-level APIs in Java, Scala, Python, R, and SQL.
- Advanced Analytics: Supports SQL queries, machine learning (MLlib), graph processing (GraphX), and stream processing (Spark Streaming).
- Flexible Deployment: Can run on Hadoop, Kubernetes, Mesos, or standalone, and access diverse data sources like HDFS, Cassandra, S3, HBase, etc.

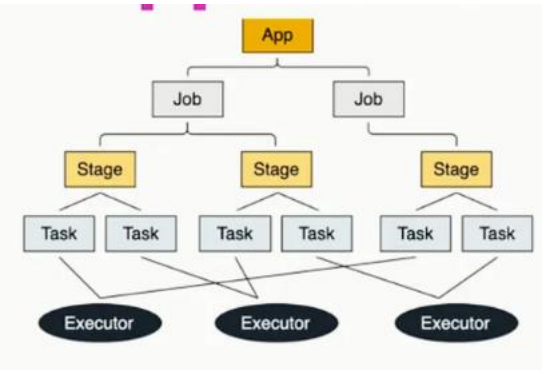
Application

Code submitted to Spark for execution. Starts the job.

Entry point (via SparkSession), runs the code logic.







1. Driver Program

- The **Driver** runs the main application (your Spark code).
- It creates the **SparkContext** (or SparkSession in newer APIs).
- Responsible for:
  - Coordinating the execution
  - Building the DAG
  - Requesting resources
  - Assigning tasks
  - Receiving results

2. SparkContext

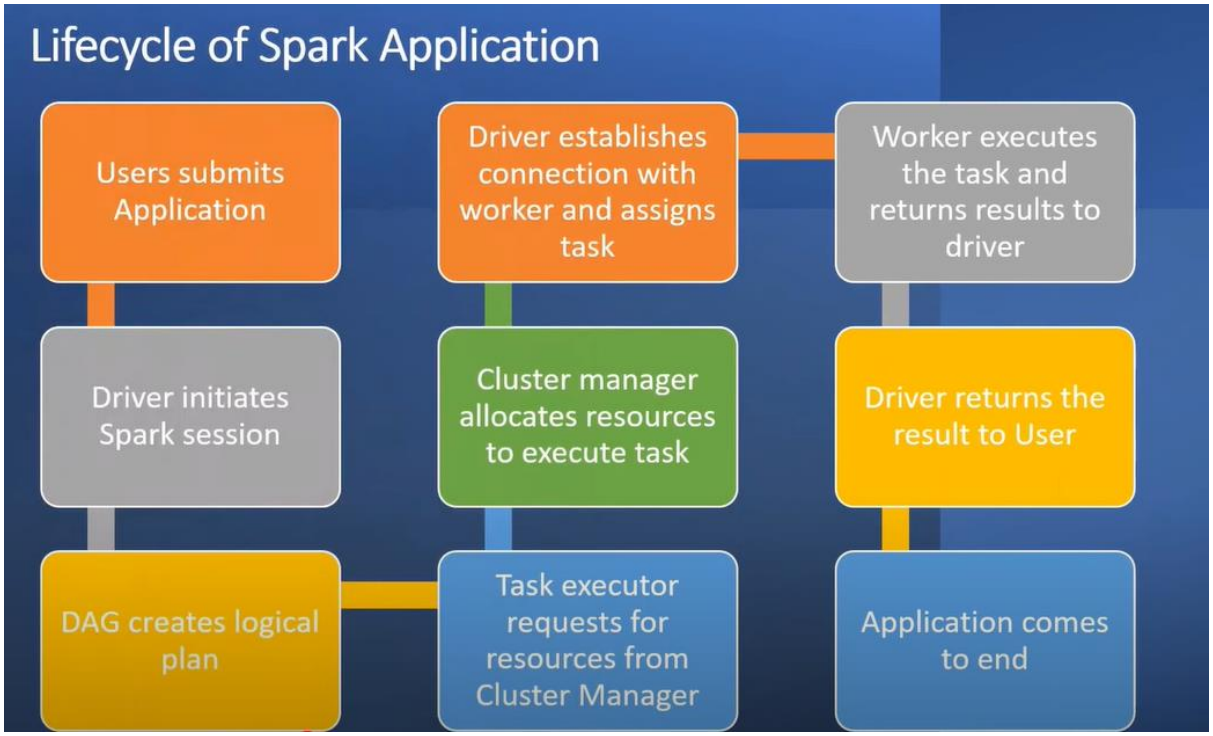
- It's the entry point to Spark functionality.
- Connects the application with the **Cluster Manager** to request resources.

3. Cluster Manager

- Manages resources across the cluster.
- Types: **Standalone**, **YARN**, **Mesos**, or **Kubernetes**.
- Allocates **Worker Nodes** with **Executors** to the Spark job.

4. Worker Nodes

- These are the actual machines that **run the code and process data**.
- Each worker node contains:
    - **Executor**: A JVM process launched for each application.
    - **Cache**: Stores data in memory (for reuse across operations).
    - **Tasks**: Smallest unit of execution, assigned by the driver.



Life Cycle of a Spark Application

---

1. Application Submission

- The user **writes Spark code** (in Python, Scala, Java, or R) and **submits** it.
- This code is packaged and submitted to a **cluster** (YARN, Standalone, Mesos, Kubernetes, etc.).

---

2. Driver Program Starts

- The **Driver** is the main controller of the Spark application.
- It starts a **SparkSession** (or SparkContext).
- Responsible for:
    - Creating the **DAG (Directed Acyclic Graph)**

- Handling task scheduling
- Communicating with the cluster manager

---

3. Job and DAG Creation

- Based on the user code:
  - Spark builds a **Logical Plan** (high-level execution).
  - Optimizes it into a **Physical Plan**.
  - Divides it into **Stages**, and then into **Tasks** — forming a **DAG**.

---

4. Resource Request to Cluster Manager

- The driver requests **executors** (compute resources) from the **Cluster Manager**.
- The cluster manager allocates **worker nodes** with executors.

---

5. Executors Launch

- Executors are launched on the allocated **worker nodes**.
- Executors:
  - Run tasks.
  - Store data in memory (for caching).
  - Report results back to the driver.

---

6. Task Execution

- The driver **distributes tasks** (from the DAG) to the executors.
- Tasks process data in parallel.
- **Shuffling** occurs if tasks need data from other partitions.

---

7. Job Completion

- Once all tasks are finished:
  - Executors return the results to the driver.
  - Driver aggregates and gives final output to the user.

---

8. Application Termination

- The driver shuts down the SparkSession.

- Executors and resources are released.
- The application ends.

If one worker node fails in Spark:

1. **Tasks on that worker fail**
    - The tasks running on the failed executor (worker node) are lost.
    - Driver gets notified that those tasks didn't complete.
2. **Spark reassigns tasks to other workers**
    - Since Spark knows the **lineage** of transformations (thanks to lazy evaluation), it can recompute the lost partitions.
    - Tasks are rescheduled on healthy worker nodes.
3. **Data safety** depends on storage:
    - If you're reading from **HDFS/Delta Lake/S3**, data isn't lost (because of replication in storage).
    - If you cached data **in memory** on the failed worker, Spark will recompute it from original source.
4. **Executor replaced**
    - If cluster manager (YARN/K8s) detects a failure, it can start a **new executor** on another machine.

Example Scenario

- Suppose your dataset has **100 partitions**, spread across **5 worker nodes** (20 partitions each).
- Worker 3 fails → its **20 partitions are lost**.
- Driver sees failed tasks → reassigns those 20 partitions to Worker 1, 2, 4, or 5.
- Job still finishes (maybe slower, since fewer workers now).

Spark Components

1. **Spark Core:** All the functionalities being provided by Apache Spark are built on the highest of the Spark Core. It delivers speed by providing in-memory computation capability. Spark Core is the foundation of parallel and distributed processing of giant dataset. It is the main backbone of the essential I/O functionalities and significant in programming and observing the role of the spark cluster. It holds all the components related to scheduling, distributing and monitoring jobs on a cluster, Task dispatching, Fault recovery. The functionalities of this component are:

    1. It contains the basic functionality of spark. (Task scheduling, memory management, fault recovery, interacting with storage systems).
    2. Home to API that defines RDDs.

2. **Spark SQL Structured data:**

    The Spark SQL component is built above the spark core and used to provide the structured processing on the data. It provides standard access to a range of data sources. It includes Hive, JSON, and JDBC.

    It supports querying data either via SQL or via the hive language. This also works to access structured and semi-structured information.

    It also provides powerful, interactive, analytical application across both streaming and historical data. Spark SQL could be a new module in the spark that integrates the relative process with the spark with programming API.

    The main functionality of this module is:

    1. It is a Spark package for working with structured data.
    2. It Supports many sources of data including hive tablets, parquet, json.
    3. It allows the developers to intermix SQK with programmatic data manipulation supported by RDDs in python, scala and java.

3. **Spark Streaming:**

    Spark streaming permits ascendible, high-throughput, fault-tolerant stream process of live knowledge streams. Spark can access data from a source like a flume, TCP socket. It will operate different algorithms in which it receives the data in a file system, database and live dashboard. Spark uses Micro-batching for real-time streaming. Micro-batching is a technique that permits a method or a task to treat a stream as a sequence of little batches of information. Hence spark streaming groups the live data into small batches. It delivers it to the batch system for processing.

The functionality of this module is:

1. Enables processing of live streams of data like log files generated by production web services.
2. The API's defined in this module are quite similar to spark core RDD API's.

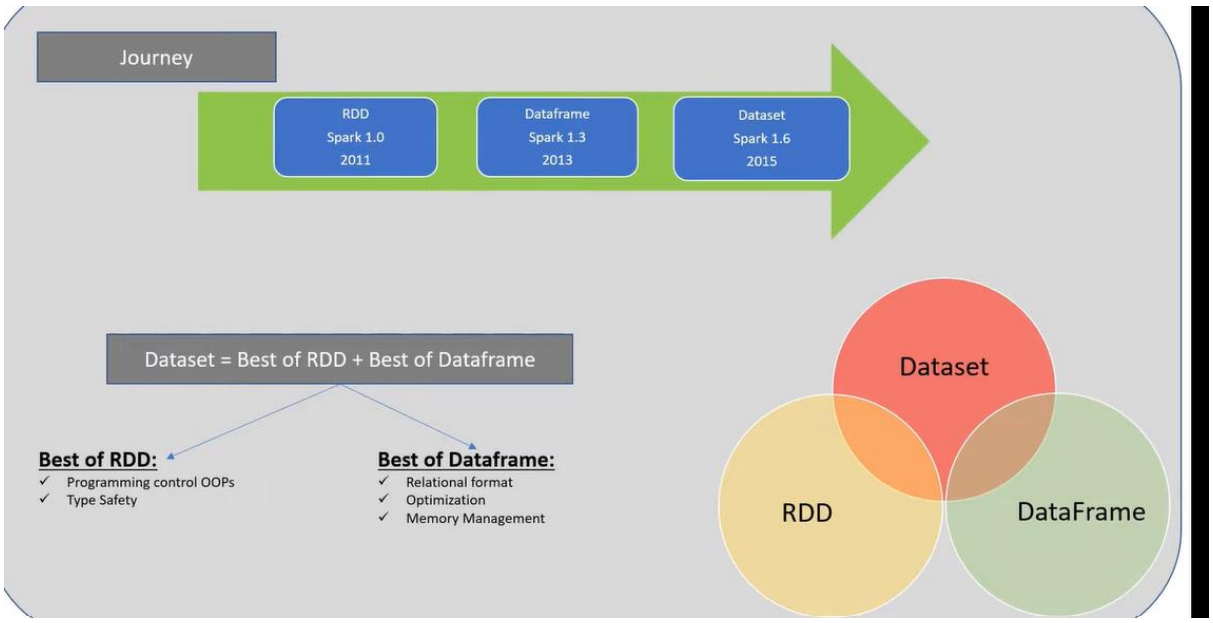
4. **MIlib Machine Learning:**

   MLlib in spark is a scalable Machine learning library that contains various machine learning algorithms. The motive behind MLlib creation is to make the implementation of machine learning simple. It contains machine learning libraries and the implementation of various algorithms

5. **GraphX graph processing:**

   It is an API for graphs and graph parallel execution. There is network analytics in which we store the data. Clustering, classification, traversal, searching, and pathfinding is also possible in the graph. It generally optimizes how we can represent vertex and edges in a graph. GraphX also optimizes how we can represent vertex and edges when they are primitive data types. To support graph computation, it supports fundamental operations like subgraph, joins vertices, and aggregate messages as well as an optimized variant of the Pregel API.

RDD, Dataframe, Dataset



(general explanation about the three Spark APIs)

- They all are APIs provided by spark for developers for data processing and analytics.
- In terms of functionality, all are same and returns same output for provided input data. But they differ in the way of handling and processing data. So, there is difference in terms of performance, user convenience and language support etc.
- Users can choose any of the API while working with spark.

RDD → Dataframe → Dataset

Dataset = best of RDD + best of Dataframe

RDD (Resilient Distributed Dataset) is the core (original) and fundamental data structure in spark. It represents an **immutable**, distributed collection of objects that can be processed in parallel across a cluster.

Resilient: it is fault tolerant and is capable of rebuilding data on failure.

Distributed: data is distributed among the multiple nodes in a cluster.

Dataset: collection of partitioned data with values

Key features of RDD

- Supports low-level transformations and actions.
- Works with both unstructured and semi-structured data.
- In-Memory computation.
- Automatically recovers from node failures (fault tolerance).
- Automatically distributes data across multiple nodes in a cluster.
- Immutable

When to use RDD?

- When working with unstructured or complex data types.
- You need custom transformations not supported in DataFrame/Dataset APIs.

DataFrame

DataFrame are structured distributed collections of data organized into named columns (similar to a table in a relational database). They are essentially a structured version of RDDs, providing a more abstract model for working with data.

Data frames are easy to use and are suitable for a wide range of data processing tasks such as handling missing values, removing duplicates, data warehousing and ETL, and real-time data processing.

Key features:

- Data is represented in a row-column format.
- Less memory consumption
- Supports low-level transformations and actions
- Advanced query optimization and efficient execution plans.

When to use DataFrame?

- You want faster execution via optimizations.
- You're working with structured data
- You prefer sql-style queries or need less complex code.

Dataset

Datasets are a powerful data structure in Apache Spark that combines the best of both worlds: the type safety and object-oriented programming capabilities of RDDs and the optimized performance and ease of use of DataFrame.

Key characteristics:

- Datasets are strongly typed, ensuring that data operations are performed on the right data types.
- Allows to work with objects directly, making it easier to define complex data structures and perform operations on them.
- Efficient data processing and query optimization.
- Supports complex, custom object mappings.

Datasets are particularly useful when you need strong type safety, functional programming capabilities, and optimized performance.

Use Dataset when:

- You need compile-time safety (type-safe code).
- You're using Scala or Java.
- You want both performance (like DataFrame) and type-checking (like RDD).
- You prefer working with custom objects/classes

Python does not support Dataset API, because Python is dynamically typed and doesn't support compile-time type checking.

Similarities

1. Fault Tolerant - can recover from node failures using lineage.
2. Distributed in nature - Data is split across multiple nodes in the cluster
3. In-memory parallel processing - Operations are performed in memory for speed.
4. Immutable - Once created, RDDs cannot be changed (you create new RDDs with transformations).
5. Lazy evaluation - Execution happens only when an action is called, not immediately.
6. Internally processing as RDDs for all APIs

RDD	DataFrame	Dataset
Fault Tolerant	Fault Tolerant	Fault Tolerant
Distributed	Distributed	Distributed
Immutabilty	Immutabilty	Immutabilty
No schema	Schema	Schema
Slow on Non-JVM languages	Faster	Faster
No Execution optimization	optimization Catalyst optimizer	optimization
Low Level	High Level	High Level
No SQL Support	SQL Support	SQL Support
Type Safe	No type Safe	Type Safe
Syntax Error detected at Compile Time	Syntax Error detected at Compile Time	Syntax Error detected at Compile Time
Analysis Error Detected at Compile time	Analysis Error Detected at Run time	Analysis Error Detected at Compile time
JAVA,SCALA, Python,R	JAVA,SCALA, Python,R	JAVA, SCALA
Higher memory is used	Higher memory is used	Low memory is used. Tungsten encoders provide great benefits

RDD is very slow in python and scala because it no JVM

Why DataFrame is faster than RDD?

1. Optimization by Catalyst (catalyst optimization)

- RDD: You just write Scala/Python code → Spark executes it step by step. No optimization.
- DataFrame: It uses the **Catalyst optimizer**. Spark analyses your query, reorders operations, and removes unnecessary steps.
  Example: If you filter before joining, Catalyst will automatically **push down the filter** to reduce data earlier.

---

2. Tungsten Execution Engine

- DataFrame uses **Tungsten** (memory and code generation optimizations).
- Spark converts your DataFrame query into **optimized JVM bytecode** (very close to machine code).
- RDD just runs regular JVM objects (slower, more GC overhead).

---

3. Efficient Memory Usage

- RDD stores data as **Java/Python objects** → heavy memory usage.
- DataFrame stores data in a **binary (off-heap) format** → compact, less garbage collection, faster access.

---

4. Query Optimization (Predicate Pushdown & Column Pruning)

- RDD loads **all data** then applies your function.
- DataFrame can **skip unnecessary columns** and **push filters to the data source** (like SQL WHERE).
  👉 Example: Reading Parquet with DataFrame:

```
df.select("name").filter("age > 30")
```

Only loads the name and age columns, not the entire dataset.
RDD will load all columns and then filter.

---

5. Integration with SQL

- DataFrame API is **declarative** (you tell *what* to do, not *how*).
- Spark SQL engine finds the **best execution plan**.
- RDD is imperative (you control *how* to do everything), which means **no automatic optimization**.

Catalyst Optimizer in Spark

Definition:
Catalyst Optimizer is the **query optimization engine** built into Spark SQL. Whenever you run a query using **DataFrame** or **SQL**, Catalyst automatically decides the **most efficient way** to execute it.

How It Works (Steps)

When you write something like:

df.filter("age > 30").select("name")

Catalyst does these steps:

1. **Parse** → Converts your SQL/DataFrame code into an **unoptimized logical plan**. Example: "Filter age > 30 then select name".
2. **Analyze** → Checks if columns (age, name) exist in schema.
3. **Optimize** → Applies rules like:
    - Predicate Pushdown → Push filter (age > 30) to data source (read less data).
    - Column Pruning → Read only name and age, not all columns.
    - Constant Folding → Replace constant expressions (2+3) with result (5).
4. **Physical Planning** → Generates multiple **physical execution plans** (different strategies to run).
   Example: sort-merge join vs broadcast join.
5. **Select Best Plan** → Chooses the most efficient one based on cost.

Transformation

Transformation is kind of operation which will transform Dataframe from one form to another. We will get new Dataframe after execution of each transformation operation. Transformation is lazy evaluation based on DAG. (filter, union, etc) [means when we execute transformation the actual execution will not happens immediately, spark engine only creates a logical plan, so called Lazy evaluation].

Action: - used to trigger some work(job)

When we want to work with actual data, call the action. Action returns the data to driver for display or storage into storage layer. (count, collect, save, etc)

Example: if there are 10 commands → 9 is transformation and final is action command

For the first 9 command it will just produce the logical plan and DAG, 10^{th} step is action it refers DAG and goes to step 1 & started executing from there.

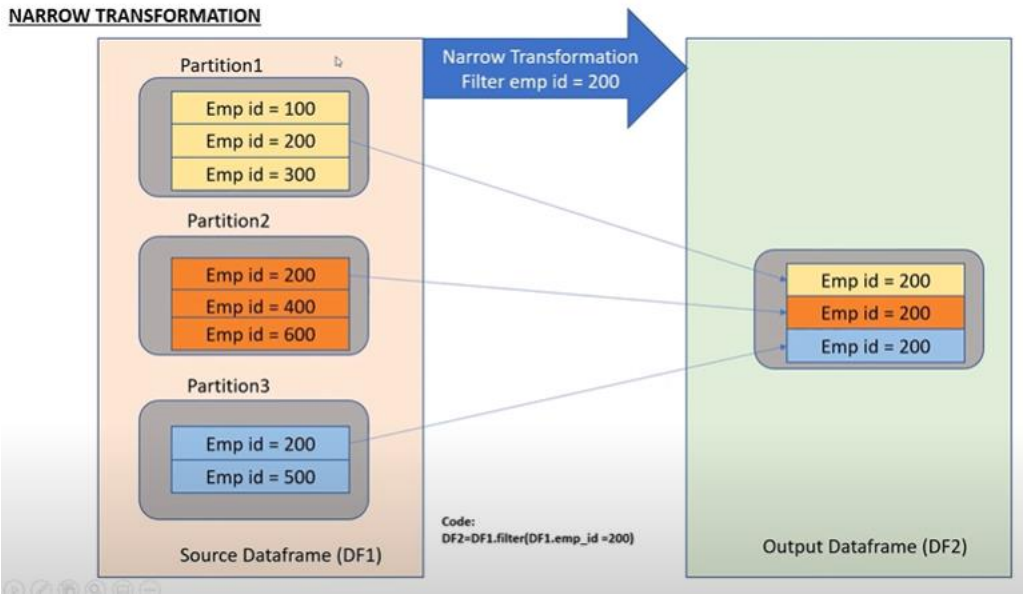
Narrow Transformations

- **Definition**: Each input partition contributes to only **one output partition**.
- **They run in**-parallel
- No data shuffling between executors.
- Fast & efficient.
- No partition dependency.

Examples:

- map(), select(), withColumn(), drop()
- filter()
- flatMap()
- union()
- sample()

Use case: Simple operations where data doesn't need to move across the cluster.



In Narrow Transformation, we don't need to shuffle the data across nodes, its simple transformation and in-expensive. They run in-parallel.

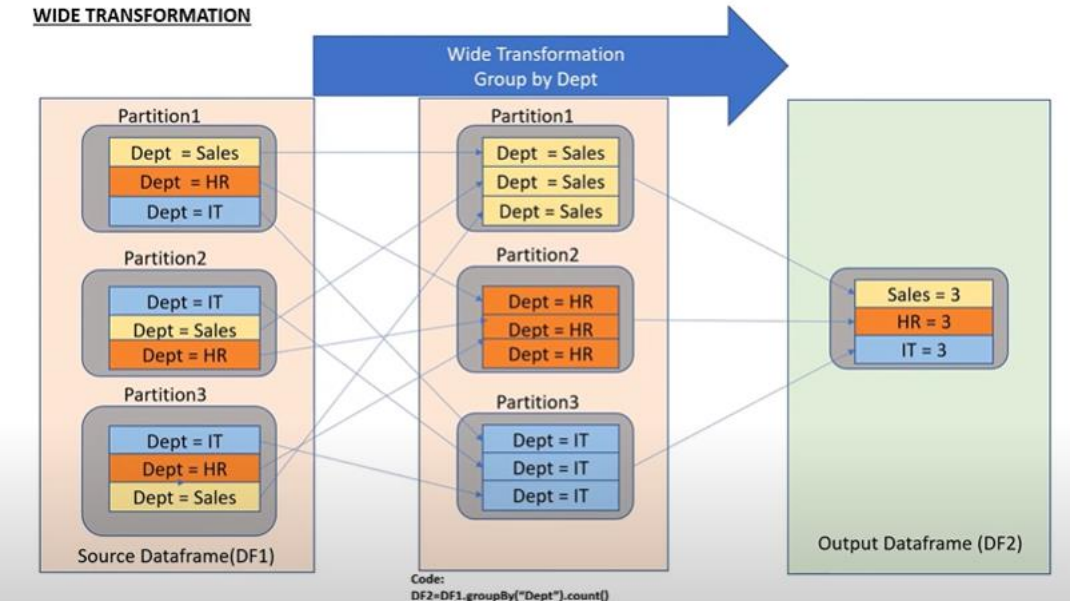
Wide Transformations

- **Definition**: An input partition may contribute to **multiple output partitions**.
- Requires **shuffle** (data movement between executors).
- Slower, but necessary for grouping/aggregating.

Examples:

- groupBy () , agg(), repartition()
- reduceByKey()
- join()
- distinct()
- repartition()

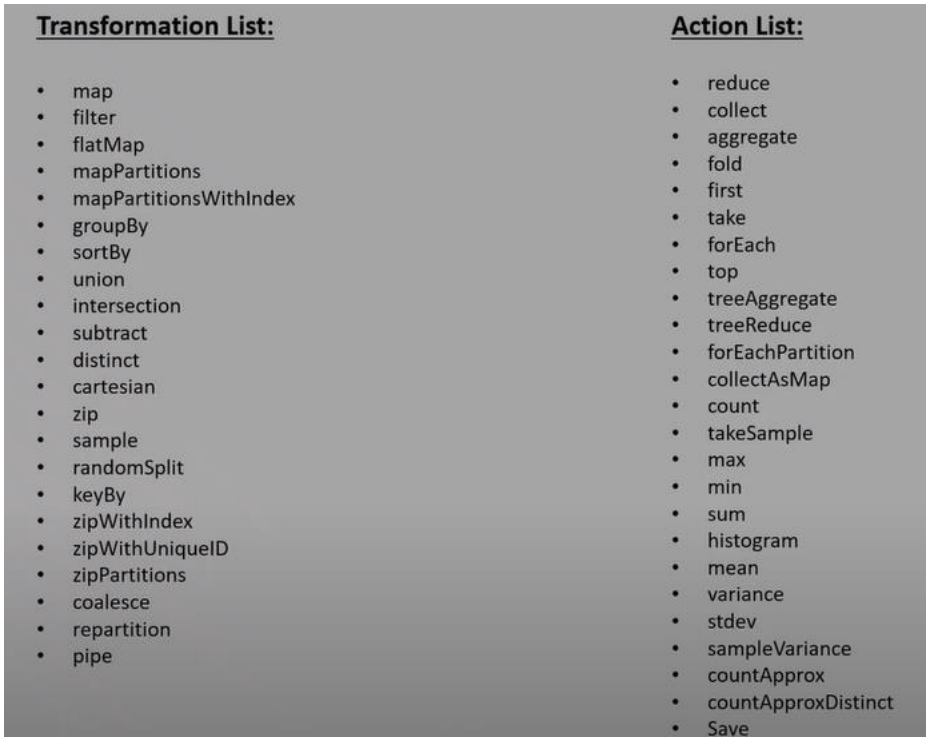
Use case: Aggregations, joins, and operations needing data from multiple partitions.



Data is getting shuffled by the nodes, so it is expensive, and it will hit the performance.

Why need shuffling? Each and every partition is depending on other partition to produce the output. (group by is wide transformation).

Output return to driver.



Lazy Evaluation

Lazy evaluation in Spark means transformations don't execute immediately. Instead, Spark builds a logical plan (DAG) of operations, and execution starts only when an action is called. This helps Spark optimize execution and improve performance.

Why Lazy Evaluation is Important?

1. **Optimization** → Spark can analyze the entire DAG and remove unnecessary steps.
2. **Fault Tolerance** → Spark can recompute lost partitions using lineage. (If there is any failure we can recreate it easily with the lineage graph).
3. **Efficiency** → Avoids computing intermediate results that are never used.

What is a Partition in Spark?

- A **partition** is the **smallest logical chunk of data** in Spark.
- Each partition is processed by **a single task** in an executor.
- Spark achieves **parallelism** by processing multiple partitions at the same time across the cluster.

How is partition created?

In Spark, partitions are created depending on the source. For files, partitions are based on HDFS block size (e.g., a 640 MB file with 128 MB block size gives 5 partitions).

Example:

- A file = **640 MB**
- Block size = **128 MB**
- Spark creates **5 partitions** (640 ÷ 128 = 5).

Each partition is then processed by one task in parallel.

For collections, partitions default to number of cores but can be specified in parallelize ().

If you create an RDD from a Python list using parallelize ():

- Default partitions = **number of CPU cores** available.
- You can also manually specify partitions.

For databases, partitions are defined by range conditions using partition columns. Additionally, transformations like repartition or coalesce can change the number of partitions.

Some operations naturally create or change partitions:

- repartition(n) → reshuffles data into n partitions.
- coalesce(n) → reduces partitions without shuffle.
- groupBy, join, reduceByKey → cause a **shuffle**, creating new partitions.

What is shuffling in spark processing?

- Shuffling is fundamental operation of redistributing or reorganizing of partitions.
- Wide tranformation --> new stage created, and data shuffled. (groupby, join, orderby)

What are job vs stage vs task?

Spark execution hierarchy or blueprint.

Whenever there is an action in the spark application then driver will start executing and create a execution plan based on that it will divide the entire operation into job, stage, task.

- Job: when spark engine reads physical data (external storage) or partition in the ram.

Job

- A **job** is triggered by an **action** (like collect(), count(), saveAsTextFile()).

- It represents the **entire computation** Spark must perform to produce the final result.
- A job consists of **multiple stages**.

Stages: when there is shuffling operation new stages created.

Stage

- A job is divided into **stages** by Spark's **DAG Scheduler**.
- **Shuffle boundaries** decide where one stage ends and another begins.
- Two types of stages:
  - **ShuffleMapStage** (before shuffle)
  - **ResultStage** (final stage returning result to driver or storage)

Note: Even if there is no shuffle, **at least one stage will always be created** in a job.

Task

- Tasks: Atomic unit of work done
- Each task operates on **one partition of data**.
- Tasks are scheduled by the **TaskScheduler** and executed by **executors** in parallel.

If you have 100 partitions in a stage → Spark launches **100 tasks** in that stage.

(Wrong statement: DataFrame is mutable, not immutable E.g: we can add new columns or modify the columns data or data types, so DataFrame is mutable)

When you add new column or changing the data type, it is internally creating new dataframe based on the previous one. It does not change the existing one by adding new column.

What is data skew?

- Uneven partition of data (i.e., in one partition contain more records (100 rows) and another partition contain (50 rows)) is Thin vs Fat partitions.
- This hits the performance.
- Avoid this by Salting Technique.

Salting means adding a random number (or extra value) to a skewed key so that Spark can split its records into multiple partitions instead of putting them all into one. This avoids data skew and speeds up joins/aggregations.

Hadoop (HDFS) Fault Tolerance

- Hadoop Distributed File System (HDFS) achieves fault tolerance by **replicating data blocks**.
- Default replication factor is **3** (each block of data is stored on 3 different nodes).
- If one node fails, the block can still be retrieved from another node.

Spark Fault Tolerance

- Apache Spark achieves fault tolerance **differently**.
- Spark does not replicate data; instead, it uses **RDD lineage (chain of transformations)**.
- If one partition of RDD is lost, Spark just re-applies the **chain of transformations** on the source data to rebuild it.
- For better performance, users can use **checkpointing** or **caching** to avoid recomputation.

Key Point: Fault tolerance is **lineage-based recomputation**.

Why Spark and not other frameworks?

- We use Spark because it is much faster than older frameworks like Hadoop MapReduce (writes intermediate results to disk) due to its in-memory processing.
- It's also easier to use since it provides high-level APIs in Python, Scala, Java, and SQL. Provides High-level abstractions: **RDD, DataFrame, Dataset**
- Spark is a unified framework that supports batch, streaming, ML, and graph processing in one engine, while other frameworks usually focus on just one area.
- It's fault-tolerant, integrates with existing Hadoop and cloud data sources, and has strong community support.

That's why Spark became the standard big data processing framework.

Resource allocation means assigning computing resources like CPU cores and memory(ram) from the cluster to a Spark job.

What is a DAG in Spark?

DAG = Directed Acyclic Graph
It is a **graph (flowchart)** that represents the sequence of **transformations** applied to data in Spark.

- **Directed** → Each step flows in one direction (no backward loops).
- **Acyclic** → No cycles/loops; you can't come back to a previous step.a
- **Graph** → Consists of nodes (RDD/DataFrame) and edges (transformations).

Why Spark uses DAG?

Earlier frameworks like **Hadoop MapReduce** only had **2 steps** (Map → Reduce).

- For multi-step jobs, it had to **write intermediate data to disk** (slow).

Spark improved this by:

1. Building a **DAG of all transformations**.
2. Optimizing the execution plan (reordering, combining).
3. Splitting it into **stages and tasks** only when an **action** is triggered.

This gives **lazy evaluation** and **performance optimization**.

Broadcast variable

A broadcast variable in Spark is a read-only variable that is cached on all worker nodes. It reduces network I/O and improves performance by avoiding sending the same data with every task. Typically used for small lookup tables, configuration maps, or reference data needed across multiple tasks.

Spark Optimization Techniques

1. Partitioning
2. Caching & Persistence
3. Broadcast Variables & Broadcast Joins (Avoid shuffling large tables during joins by broadcasting small tables.)
4. Reduce Shuffles
5. Predicate Pushdown & Column Pruning
6. Transition from RDD to DataFrame/Dataset
7. Implement dynamic resource allocation
8. Avoid wide transformations where possible

Apache Spark uses a **master–worker architecture**.

- The **Driver program** runs your application, driver converts the code in logical plan, builds a **DAG** of transformations, and splits it into **stages** and **tasks**.

  Responsible for - converting code into a DAG of stages, - requesting resources, -Tracking the task.

- the driver **requests (CPU and memory) resources** from a **Cluster Manager** (like Standalone, YARN, or Kubernetes).
- The cluster manager allocates those resources and launches **Executor processes** on worker nodes.
- **Executors** run the tasks and store data in memory or on disk. Each task handles one data **partition**, so tasks run **in parallel** for speed.
- When you trigger an action like count or write, the driver schedules the tasks, collects results, and retries failures if needed.
- **Fault tolerance** is provided by RDD lineage: if a partition is lost, Spark recomputes only that piece.

Spark vs Hadoop

Hadoop	Spark
Hadoop's MapReduce model reads and writes from a disk, thus slowing down the processing speed.	Spark reduces the number of read/write cycles to disk and stores intermediate data in memory, hence faster-processing speed.
Fault tolerance – Hadoop is a highly fault-tolerant system where Fault-tolerance achieved by **replicating** blocks of data.	Lineage - Fault-tolerance achieved by storing chain of transformations If data is lost, the chain of transformations can be recomputed on the original data

It uses Java for MapReduce apps.	It uses Java, R, Scala, Python, or Spark SQL for the APIs.
Hadoop is designed to handle batch processing efficiently.	Spark is designed to handle real-time data efficiently.
Hadoop is a cheaper option available while comparing it in terms of cost	Spark requires a lot of RAM to run in-memory, thus increasing the cluster and hence cost.

We choose Spark because it's far faster, supports streaming and machine learning in the same engine, and is easier to program. Hadoop MapReduce is disk-heavy and batch-only, while Spark leverages memory and a DAG optimizer for real-time and complex analytics.

Pyspark	Pandas
Operation on Pyspark DataFrame run parallel on different nodes in cluster	Pandas will be using ony core of your CPU
Operations in PySpark DataFrame are lazy in nature	but, in case of pandas we get the result as soon as we apply any operation.
In PySpark RDD, we can't change the DataFrame due to it's immutable property, we need to transform it.	Pandas DF are not immutable in nature
PySpark API Supports less type of operations. But it is getting developed	Pandas API support more operations than PySpark DataFrame. Still pandas API is more powerful than Spark.
Complex operations in PySpark is not easier to perform	Complex operations in pandas are easier to perform
Pyspark DataFrame Access is Slower but processing is fast	Pandas dataframe access is faster but limited to available memory, but processing is slow
setting up Cluster is Required	no need for a cluster
Complicated,	simpler, more flexible, more libs, easier to implement