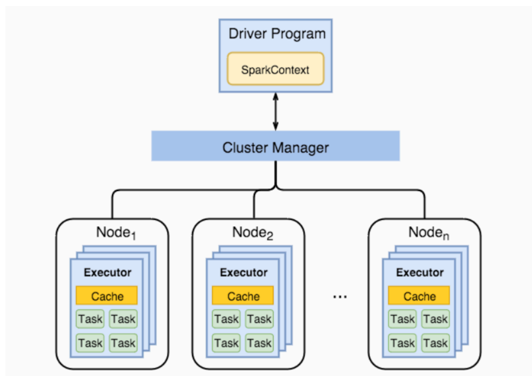
Apache Spark Architecture and Its Core Components

Apache Spark is a fast, general-purpose **distributed data processing framework** designed for large-scale data analytics. It follows a **master–worker architecture** and performs in-memory computation, which makes it much faster than disk-based processing systems



Spark Architecture – Short Explanation

- **Driver Program**
  - Runs the main Spark application
  - Contains **SparkContext**
  - Creates DAG and schedules tasks
  - Communicates with Cluster Manager
- **Cluster Manager**
  - Manages cluster resources (CPU, memory)
  - Allocates resources to the Spark application
  - Launches Executors on worker nodes
- **Worker Nodes**
  - Machines that perform actual computation
  - Host one or more Executors

    - Execute tasks in parallel
- **Executors**
    - Processes running on worker nodes
    - Execute tasks assigned by the Driver
    - Cache data in memory for fast access
    - Exist for the lifetime of the application

2. Core Components of Spark

2.1 RDD (Resilient Distributed Dataset)

RDD is the fundamental data abstraction in Spark.

Description:

An RDD is an **immutable, distributed collection of objects** that can be processed in parallel across a cluster.

Key Characteristics:

- Distributed across multiple nodes
- Immutable (cannot be changed once created)
- Fault-tolerant through lineage
- Can be cached in memory
- Supports parallel operations

Creation of RDDs:

- From data in HDFS or local file system
- From existing RDDs using transformations

Example:

- Reading a text file and splitting it into words

2.2 DAG (Directed Acyclic Graph)

DAG represents the execution plan of Spark jobs.

Description:

A DAG is a **logical execution plan** that shows how transformations on RDDs are connected.

Key Points:

- Created by the Spark Driver
- Nodes represent RDDs
- Edges represent transformations
- Optimizes execution by:
    - Stage division
    - Task scheduling
- Ensures no cyclic dependencies

Importance:

- Enables efficient scheduling
- Reduces unnecessary computations
- Improves performance

2.3 Executors

Executors are worker processes that run tasks and store data.

Description:

Executors are launched on worker nodes and are responsible for executing tasks assigned by the Driver.

Functions of Executors:

- Execute tasks
- Store RDD data in memory or disk
- Return results to the Driver
- Enable parallel processing

Key Features:

- One or more Executors per worker node
- Exist for the lifetime of the Spark application
- Improve speed by caching data

---

3. Role of Other Spark Components (Brief)

- **Driver Program**
  - Controls the application
  - Builds DAG
  - Schedules tasks
- **Cluster Manager**
  - Allocates resources
  - Examples: YARN, Mesos, Standalone

---

4. Advantages of Spark Architecture

- In-memory processing → high speed
- Fault tolerance using RDD lineage
- Supports batch and real-time processing
- Scalable and efficient

Transformations and Actions on RDDs (with Examples)

In Apache Spark, operations on **RDDs (Resilient Distributed Datasets)** are broadly classified into **Transformations** and **Actions**. Transformations define *what* operation to perform, while Actions trigger *execution* and return results.

---

1. Transformations on RDDs

Transformations are operations that create a new RDD from an existing RDD.

They are **lazy**, meaning Spark does not execute them immediately; it only builds a logical execution plan (DAG).

Key Features

- Lazy evaluation
- Return a new RDD
- Do not produce output immediately
- Can be chained together

Common Transformations with Examples

- **map()**

  Transforms each element of the RDD

  ```
  rdd.map(lambda x: x *2)
  ```

- **filter()**

  Selects elements based on a condition

  ```
  rdd.filter(lambda x: x >10)
  ```

- **flatMap()**

  Maps and flattens the result

  ```
  rdd.flatMap(lambda x: x.split())
  ```

- **reduceByKey()**

Aggregates values with the same key

```
rdd.reduceByKey(lambda a, b: a + b)
```

- **union()**

  Combines two RDDs

```
rdd1.union(rdd2)
```

---

2. Actions on RDDs

Actions are operations that trigger the execution of transformations and return results to the driver or write data to external storage.

Key Features

- Trigger job execution
- Return results or save output
- Final step in Spark processing

Common Actions with Examples

- **collect()**

  Returns all elements to the driver

```
rdd.collect()
```

- **count()**

  Returns number of elements

```
rdd.count()
```

- **first()**

  Returns the first element

  ```
  rdd.first()
  ```

- **take(n)**

  Returns first `n` elements

  ```
  rdd.take(5)
  ```

- **saveAsTextFile()**

  Saves output to storage

  ```
  rdd.saveAsTextFile("output")
  ```

Example (Transformation + Action)

```
rdd = sc.textFile("data.txt")
words = rdd.flatMap(lambda x: x.split())# Transformation
count = words.count()# Action
```

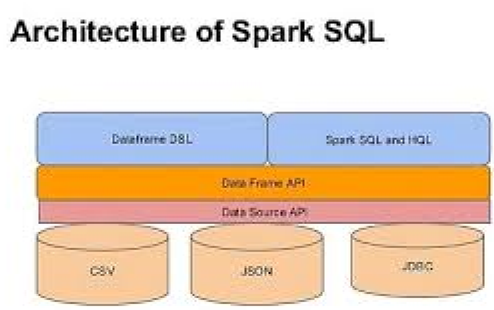
- `flatMap()` → Transformation (lazy)
- `count()` → Action (executes the job)

Difference Between Transformations and Actions

Aspect	Transformations	Actions
Execution	Lazy	Immediate
Output	New RDD	Result / Output
Purpose	Define computation	Trigger computation
Examples	map, filter	collect, count

DataFrames and Spark SQL (with advantages over RDDs)

Apache Spark provides **DataFrames** and **Spark SQL** as higher-level abstractions over RDDs to make data processing **simpler, faster, and more optimized**.



---

1. DataFrames

A DataFrame is a distributed collection of data organized into named columns, similar to a table in a relational database.

Explanation

- Built on top of RDDs
- Data is organized in **rows and columns**

- Supports structured and semi-structured data
- Schema defines column names and data types
- Can be created from:
    - CSV, JSON, Parquet
    - Hive tables
    - Existing RDDs

Example

```
df = spark.read.csv("data.csv", header=True)
df.show()
```

Key Features

- Column-based operations
- Automatic optimization
- Less code compared to RDDs
- Easier to understand and use

---

2. Spark SQL

Spark SQL is a Spark module used to process structured data using SQL queries.

Explanation

- Allows querying DataFrames using **SQL syntax**
- Integrates SQL with Spark programs
- Uses **Catalyst Optimizer** for query optimization
- Supports ANSI SQL

Example

```
df.createOrReplaceTempView("students")
spark.sql("SELECT * FROM students WHERE marks > 80").show()
```

Key Features

- SQL-based querying
- Easy integration with BI tools
- Optimized execution plans
- Supports Hive queries

---

3. Advantages of DataFrames and Spark SQL over RDDs

Aspect	RDDs	DataFrames / Spark SQL
Abstraction Level	Low-level	High-level
Schema	No schema	Schema-based
Optimization	Manual	Automatic (Catalyst Optimizer)
Performance	Slower	Faster
Ease of Use	Complex	Simple
SQL Support	Not supported	Fully supported
Memory Management	Manual caching	Optimized
Code Length	More	Less

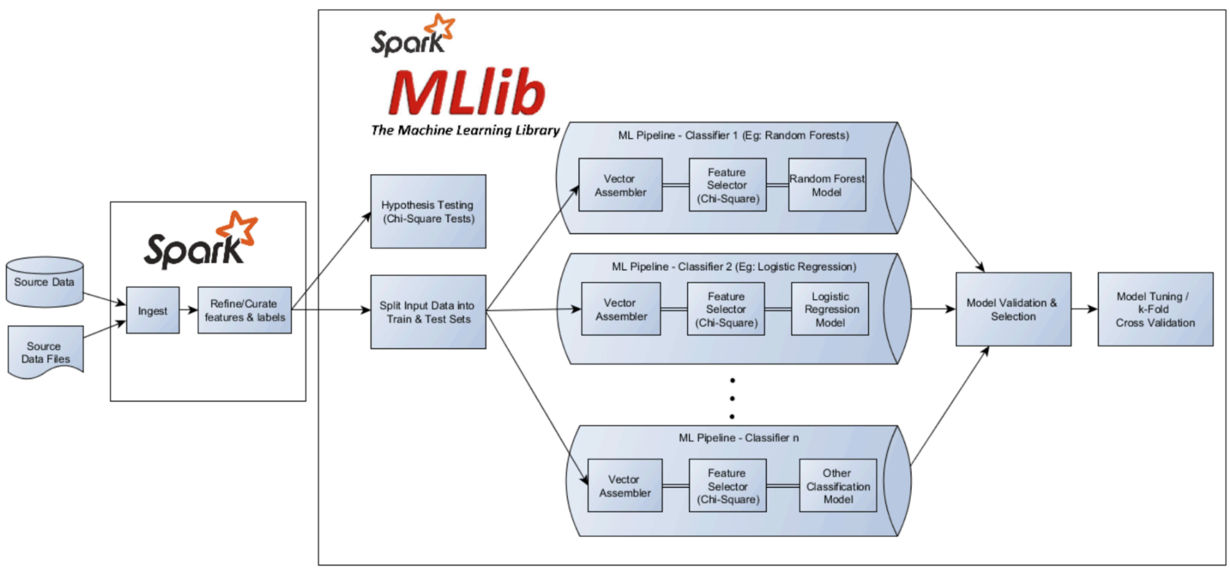
Why DataFrames & Spark SQL are Better than RDDs

- Automatic query optimization
- Better performance due to optimized execution plans
- Easier for developers and analysts

- Supports SQL and structured data processing
- Reduced development time

Spark MLlib and PySpark for Big Data Analytics

Apache Spark provides **MLlib** for scalable machine learning and **PySpark** as a Python API to perform Big Data analytics efficiently. Together, they enable large-scale data processing, analysis, and machine learning on distributed datasets.



---

1. Spark MLlib

Spark MLlib is Spark's **machine learning library** designed to perform ML tasks on large datasets in a distributed manner.

Explanation

- Built on top of Spark's distributed computing engine
- Provides scalable and fault-tolerant ML algorithms

- Works with RDDs, DataFrames, and Spark SQL
- Supports pipeline-based ML workflows

Main Features

- **Classification**: Logistic Regression, Decision Trees
- **Regression**: Linear Regression
- **Clustering**: K-Means
- **Recommendation**: ALS (collaborative filtering)
- **Feature Engineering**: Normalization, tokenization
- **Model Evaluation**: Accuracy, precision, recall

Advantages

- Scales to very large datasets
- Faster due to in-memory processing
- Easy integration with Spark ecosystem

---

2. PySpark for Big Data Analytics

PySpark is the **Python interface to Apache Spark**, allowing users to write Spark applications using Python.

Explanation

- Enables Big Data analytics using Python syntax
- Provides access to Spark core, Spark SQL, MLlib, and Streaming
- Suitable for data analysis, ETL, and machine learning

How PySpark is Used for Big Data Analytics

- **Data Ingestion**: Read large datasets from HDFS, S3, CSV, JSON
- **Data Processing**: Clean, filter, and transform data using DataFrames
- **Analytics**: Perform aggregations and statistical analysis

- **Machine Learning**: Build ML models using Spark MLlib
- **Visualization Support**: Collect results for plotting

Example (PySpark + MLlib)

```
from pyspark.ml.clusteringimport KMeans

model = KMeans(k=3).fit(data)
predictions = model.transform(data)
```

---

3. Advantages of PySpark for Big Data Analytics

- Easy to learn and use (Python-based)
- Handles very large datasets efficiently
- Faster execution compared to traditional Python tools
- Integrates ML, SQL, and analytics in one platform
- Supports interactive data analysis