UNIT III – Data Processing with Apache Spark

Spark Architecture (RDD, DAG, Executors)

Introduction

Apache Spark is an open-source, distributed, cluster-computing framework designed for fast and scalable processing of large volumes of data. It supports batch processing, real-time stream processing, machine learning, graph analytics, and SQL-based data analysis. Spark overcomes the limitations of Hadoop MapReduce by enabling in-memory computation, thereby achieving significant performance improvements.

The Spark architecture is built around three fundamental pillars:

Resilient Distributed Dataset (RDD) – data abstraction

Directed Acyclic Graph (DAG) – execution model

Executors – task execution engine

Together, these components provide fault tolerance, scalability, and high-speed processing.

High-Level Overview of Spark Architecture

The main components of Apache Spark architecture are:

1. Driver Program
2. SparkContext / SparkSession
3. Cluster Manager
4. Worker Nodes
5. Executors
6. RDD
7. DAG Scheduler and Task Scheduler

Spark applications run in a master–slave architecture, where the driver acts as the master and worker nodes act as slaves executing tasks.

Driver Program

The Driver Program is the central control unit of a Spark application.

Responsibilities of the Driver:

Executes the main() method of the application

Creates SparkContext (or SparkSession)

Converts user code into a logical execution plan

Builds RDD lineage and DAG

Schedules jobs, stages, and tasks

Collects results from executors

Internal Components of the Driver:

DAG Scheduler – splits jobs into stage

Task Scheduler – sends tasks to executor

Block Manager – manages cached data and shuffle blocks

The driver remains active throughout the lifetime of the Spark application.

Cluster Manager

The Cluster Manager is responsible for managing cluster resources.

Types of Cluster Managers:

Spark Standalone

Hadoop YARN

Apache Mesos

Kubernetes

Functions:

Allocates CPU cores and memory

Launches executor processes on worker nodes

Monitors resource usage

Handles dynamic allocation of executors

Worker Nodes

Worker nodes are the machines that perform the actual computation.

Role of Worker Nodes:

Host executor processes

Execute tasks sent by the driver

Store RDD partitions in memory or disk

Return computed results to the driver

Each worker node can run multiple executors depending on available resources.

Executors

Executors are distributed processes running on worker nodes.

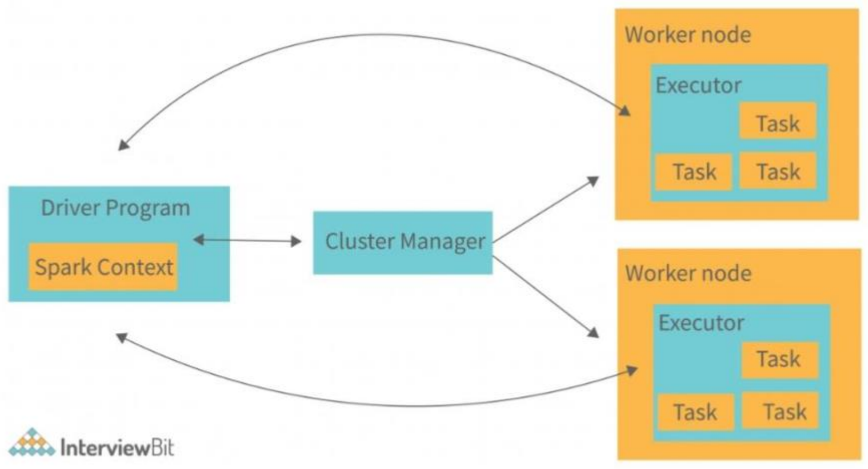
Responsibilities of Executors:

Execute tasks assigned by the driver

Cache RDD partitions for reuse

Perform parallel computation

Report task status back to the driver



Characteristics:

Executors live for the entire duration of the application

Each executor has a fixed number of cores and memory

Executors increase parallelism and performance

Multiple tasks run concurrently within an executor

Executors are crucial for Spark's in-memory and high-speed processing.

Resilient Distributed Dataset (RDD)

Definition:

An RDD is an immutable, distributed collection of objects partitioned across nodes in a Spark cluster.

Key Characteristics:

Resilient – Recovers automatically from failures

Distributed – Data is split into partitions

Immutable – Cannot be modified after creation

Lazy Evaluation – Computation starts only when an action is triggered

Fault-Tolerant – Uses lineage information

RDD Operations:

1. Transformations

Create new RDDs

Lazy operations

Examples: map(), filter(), flatMap()

2. Actions

Trigger execution

Return results to the driver

Examples: count(), collect(), saveAsTextFile()

RDD Lineage:

RDD maintains a lineage graph that tracks how each RDD is derived. In case of failure, lost partitions are recomputed instead of being replicated.

Directed Acyclic Graph (DAG)

Definition:

A Directed Acyclic Graph is a logical execution plan that represents a sequence of transformations applied to RDDs.

DAG in Spark:

Built by the driver using RDD lineage

Nodes represent RDDs

Edges represent transformations

No cycles exist

DAG Scheduler:

Converts jobs into DAGs

Splits DAGs into stages

Identifies shuffle boundaries

Optimizes task execution

Benefits of DAG:

Reduces unnecessary data shuffling

Optimizes execution plan

Improves performance over MapReduce

Enables pipelined execution

Spark Job Execution Flow

1. User submits a Spark application

2. Driver creates SparkContext

3. RDD transformations build a DAG

4. DAG Scheduler divides jobs into stages

5. Task Scheduler assigns tasks to executors

6. Executors execute tasks in parallel

7. Results are returned to the driver

Modes of Spark Execution

1. **Cluster Mode**

Driver runs inside the cluster

Recommended for production

2. Client Mode

Driver runs on the client machine

Suitable for debugging

3. Local Mode

Runs on a single machine

Used for testing and learning

Advantages of Spark Architecture

High-speed processing due to in-memory computation

Fault tolerance using RDD lineage

Scalable and distributed execution

Efficient DAG-based scheduling

Supports multiple languages (Scala, Java, Python, R)

Suitable for batch and real-time processing

Applications of Spark

Big data analytics

Real-time stream processing

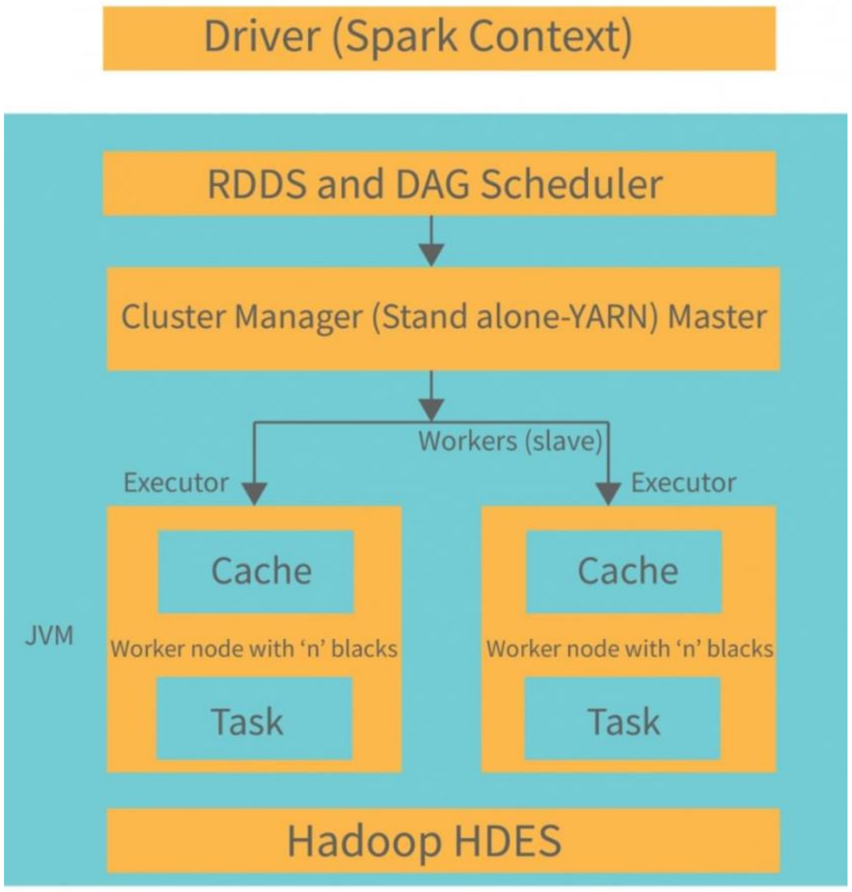
Machine learning (MLlib)

Graph processing (GraphX)

Interactive SQL queries (Spark SQL

Conclusion

Apache Spark architecture integrates RDD for data abstraction, DAG for optimized execution planning, and Executors for parallel task execution. These components work together under the control of the driver and cluster manager to deliver high-performance, fault-tolerant, and scalable big data processing. Due to these features, Spark has become one of the most widely used frameworks in modern data analytics.



Spark SQL and Data Frames

Introduction

Apache Spark is a fast and scalable distributed data processing framework. Spark SQL and DataFrames are high-level abstractions provided by Spark to process structured and semi-structured data efficiently. Spark SQL enables SQL-based querying, while DataFrames provide a table-like structure with powerful programmatic operations. Both improve performance, scalability, and ease of development compared to low-level RDDs.

Spark SQL

Definition

Spark SQL is a Spark module that allows users to process structured data using SQL queries, DataFrames, and Datasets. It introduces schema awareness and automatic query optimization.

Characteristics of Spark SQL

SQL-Based Declarative Interface – Allows users to write ANSI SQL queries easily.

Schema Awareness – Understands data structure, enabling validation and optimization.

Catalyst Optimizer – Optimizes queries using logical and physical optimization techniques.

Tungsten Execution Engine – Improves memory and CPU efficiency.

Hive Compatibility – Supports HiveQL, Hive Metastore, and UDFs.

Operations in Spark SQL

DDL Operations – CREATE, DROP, ALTER

DML Operations – SELECT, INSERT

Aggregation Operations – GROUP BY, COUNT, SUM, AVG

Join Operations – INNER, LEFT, RIGHT, FULL JOIN

Applications of Spark SQL

Interactive data analysis

Data warehousing

Business intelligence reporting

Querying structured big data

Integration with BI tools

Advantages of Spark SQL

Ease of Use

SQL syntax allows users with database knowledge to work with big data easily.

High Performance

Uses Catalyst optimizer and Tungsten engine to improve query execution speed.

Automatic Optimization

Queries are optimized automatically without manual tuning.

Multiple Data Source Support

Can query data from Hive, JSON, CSV, Parquet, ORC, and relational databases.

Scalability

Efficiently processes large datasets across distributed clusters.

DataFrames

Definition

A DataFrame is a distributed, immutable collection of data organized into named columns, similar to a table in a relational database. It is built on Spark SQL and automatically optimized.

Characteristics of DataFrames

Structured Schema – Data organized into columns with defined data types.

Distributed Nature – Data is partitioned across cluster nodes.

Immutability – Transformations create new DataFrames.

Lazy Evaluation – Execution occurs only when an action is triggered.

Fault Tolerance – Lineage information enables recovery from failures.

Language Independent – Supported in Python, Scala, Java, and R.

Operations in DataFrames

Transformation Operations

select(), filter(), withColumn(), drop(), join()

Aggregation Operations

groupBy(), count(), sum(), avg(), min(), max()

Action Operations

show(), collect(), count(), take(), write()

SQL Operations

Querying DataFrames using Spark SQL after creating temporary views

Applications of DataFrames

Big data analytics

ETL pipelines

Machine learning preprocessing

Log and event analysis

Batch and streaming processing

Advantages of DataFrames

Better Performance than RDDs

Optimized execution and efficient memory management.

Less Code and Simplicity

High-level APIs reduce complexity and development time.

Automatic Query Optimization

Catalyst optimizer improves execution without user effort.

Efficient Memory Usage

Uses optimized in-memory representation.

Interoperability

Works seamlessly across multiple languages and data sources.

Diagram

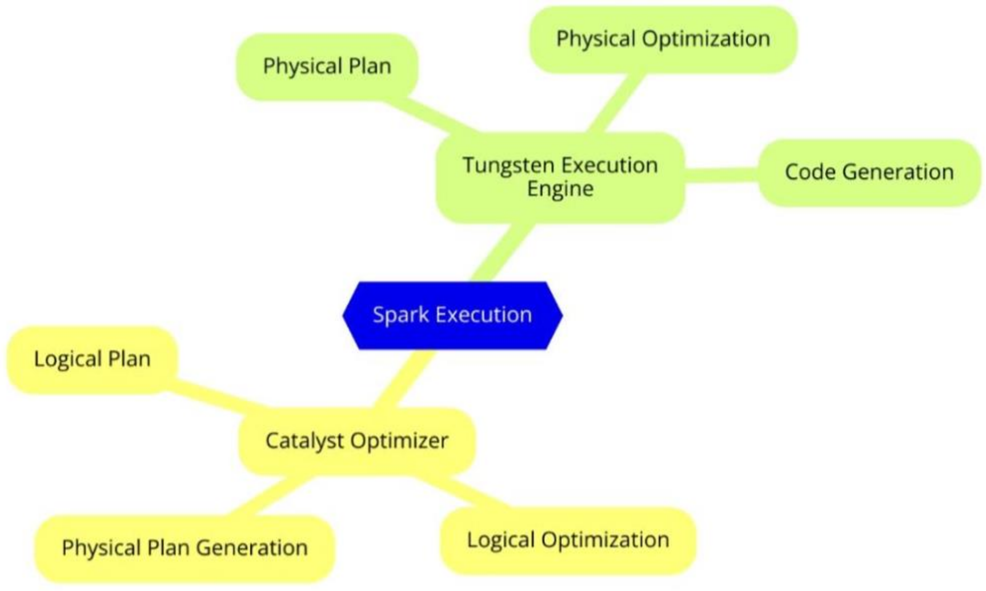


Diagram Explanation

The diagram shows how SQL queries or DataFrame operations are converted into a logical plan, optimized using the Catalyst Optimizer, converted into a physical plan, and executed efficiently using the Tungsten Execution Engine over distributed data sources.

User Interface Layer

The user submits queries using SQL queries or DataFrame APIs.

Spark SQL Engine

Spark SQL converts the query into a logical plan.

Catalyst Optimizer

The logical plan is optimized using rule-based and cost-based optimizations such as predicate pushdown and column pruning.

Physical Plan Generator

Optimized logical plan is converted into a physical execution plan.

Tungsten Execution Engine

Executes the plan efficiently using optimized memory management and CPU execution.

Data Sources

Data is read from various sources like HDFS, Hive, CSV, JSON, Parquet, ORC, JDBC databases.

Result Output

Final results are returned to the user or stored back in external storage.

Example for Spark SQL

Scenario

Assume we have an Employee table with the fields: emp_id, name, department, salary

SQL

```
SELECT department, AVG(salary) AS avg_salary

FROM employee

GROUP BY department;
```

Explanation

The query groups employees based on department.

It calculates the average salary for each department.

Spark SQL converts this query into a logical plan, optimizes it using the Catalyst Optimizer, and executes it efficiently using the Tungsten Execution Engine.

Example for DataFrames (Pyspark)

Scenario

Reading employee data from a CSV file and performing aggregation.

Python

```
df = spark.read.csv("employee.csv", header=True, inferSchema=True)

df.groupBy("department") \
  .avg("salary") \
  .show()
```

Explanation

The DataFrame reads structured data from a CSV file.

groupBy() groups records by department.

avg() computes the average salary.

Execution happens lazily and is optimized automatically.

Same Operation: Spark SQL vs DataFrame (Comparison Example)

Spark SQL

Sql

```
SELECT department, COUNT(*)
FROM employee
GROUP BY department;
```

DataFrame

Python

```
df.groupBy("department").count().show()
```

Both Spark SQL and DataFrames perform the same operation, but Spark SQL uses a SQL interface, whereas DataFrames use a programmatic API.

Comparison: Spark SQL vs DataFrames

Feature	**Spark SQL**	**DataFrames**
Interface	SQL-based	API-based
User Type	SQL users / Analysts	Developers
Query Style	Declarative	Functional
Flexibility	Limited to SQL	More flexible
Optimization	Automatic	Automatic
Performance	High	High
Language Support	SQL	Python, Scala, Java, R

Spark for ETL: joins, filters, aggregations

Introduction

ETL (Extract, Transform, Load) is a fundamental process in big data and data warehousing systems. It involves extracting data from multiple heterogeneous sources, transforming it into a clean and structured format, and loading it into a target system such as a data warehouse or data lake. Apache Spark is widely used for ETL because of its in-memory processing, distributed architecture, and high-level APIs such as Spark SQL and DataFrames, which simplify complex data transformations at scale.

Role of Spark in ETL

Spark is mainly used in the Transform phase of ETL, where large volumes of raw data are cleaned, integrated, and summarized. Spark supports ETL operations through:

DataFrames and Spark SQL

Optimized execution using Catalyst Optimizer

Efficient memory management using Tungsten

The most common transformation operations in Spark-based ETL are filters, joins, and aggregations.

1. **Filters in Spark ETL**

Meaning

Filtering is the process of selecting required records and eliminating unnecessary or invalid data during ETL. Filters help in improving data quality and reducing data size.

Why Filters are Important in ETL

Remove duplicate or invalid records

Apply business rules

Reduce processing and storage cost

Improve performance of downstream operations

Common Filtering Scenarios

Removing null values

Selecting recent transactions

Applying threshold conditions (age, salary, amount, etc.)

DataFrame Example

DataFrame Example

```
filtered_df = df.filter(df.salary > 30000)
```

Spark SQL Example

```
SELECT *
FROM employee
WHERE salary > 30000;
```

Explanation

Only employee records with salary greater than 30,000 are retained. Spark applies predicate pushdown during optimization, ensuring faster execution.

2. **Joins in Spark ETL**

Meaning

Joins are used to combine data from multiple sources based on a common key. In ETL pipelines, data often comes from different systems such as customer databases, transaction logs, and product catalogs.

Why Joins are Important in ETL

Data integration and enrichment

Creation of fact and dimension tables

Building a unified view of data

Types of Joins in Spark

Inner Join

Left Outer Join

Right Outer Join

Full Outer Join

DataFrame Example

```
joined_df = orders.join(customers, "customer_id", "inner")
```

Spark SQL Example

```
SELECT o.order_id, c.customer_name
FROM orders o
INNER JOIN customers c
ON o.customer_id = c.customer_id;
```

Explanation

Order data is enriched with customer information using a common customer_id. Spark optimizes joins using techniques like broadcast joins and join reordering.

3. Aggregations in Spark ETL

Meaning

Aggregation is the process of summarizing detailed data into meaningful metrics. Aggregations are used to generate reports, KPIs, and analytical summaries.

Why Aggregations are Important in ETL

Reduce large datasets into summarized form

Generate business insights

Prepare data for dashboards and analytics

Common Aggregation Functions

COUNT

SUM

AVG

MIN

MAX

DataFrame Example

```
agg_df = sales.groupBy("region").sum("revenue")
```

Spark SQL Example

```
SELECT region, SUM(revenue) AS total_revenue
FROM sales
GROUP BY region;
```

Explanation

Sales data is grouped by region and total revenue is calculated for each region. Spark performs this efficiently using distributed aggregation.

Complete ETL Workflow Using Spark

1. **Extract**

Read data from CSV, JSON, Hive, relational databases, or cloud storage.

Python

```
df = spark.read.csv("sales.csv", header=True, inferSchema=True)
```

2. **Transform**

Apply filters to clean data

Use joins to integrate multiple datasets

Perform aggregations to summarize data

Python

```
clean_df = df.filter(df.revenue > 0)
final_df = clean_df.groupBy("product_id").sum("revenue")
```

3. **Load**

Write processed data to HDFS, Hive tables, or data warehouse.

Python

```
final_df.write.mode("overwrite").parquet("output/")
```

Advantages of Using Spark for ETL

High Performance

In-memory processing significantly reduces execution time compared to disk-based systems.

Scalability

Spark can process terabytes or petabytes of data across distributed clusters.

Ease of Development

High-level APIs reduce code complexity and development time.

Automatic Optimization

Catalyst optimizer improves query performance without manual tuning.

Fault Tolerance

Lineage-based recovery ensures reliability in case of failures.

Support for Multiple Data Sources

Easily integrates with structured and semi-structured data.

Real-World ETL Use Cases

Sales data aggregation for dashboards

Customer behavior analysis

Log and clickstream processing

Financial reporting systems

Data lake and data warehouse pipelines

Conclusion

Apache Spark is a powerful and efficient platform for ETL workloads. Its support for filters, joins, and aggregations enables scalable data cleaning, integration, and summarization. By using Spark SQL and DataFrames, Spark-based ETL pipelines achieve high performance, reliability, and flexibility, making Spark a preferred choice for modern big data processing systems.

Use cases in AI/DS pipelines

Apache Spark is widely used in AI (Artificial Intelligence) and DS (Data Science) pipelines because it efficiently handles large-scale structured and semi-structured data, which is the foundation for machine learning and analytics.

1. Data Preprocessing for Machine Learning

Filters: Remove null, inconsistent, or outlier records to ensure clean input for ML models.

Example: Filter out customers with missing age or invalid purchase history.

Joins: Combine datasets from multiple sources (e.g., customer info + transaction logs) to create a feature-rich dataset.

Aggregations: Generate aggregated features like total purchases, average revenue, or user activity count for each user.

Impact: Clean, enriched, and structured data improves model accuracy and reduces training errors.

2. Feature Engineering

Aggregations: Summarize raw data into features such as average session time, total clicks, or monthly spending.

Filters: Remove irrelevant events or low-variance features.

Joins: Integrate external datasets (e.g., demographic, social media, weather) to enhance predictive power.

Impact: Produces a high-quality feature matrix suitable for training ML algorithms like regression, classification, or clustering.

3. **Real-Time Analytics and AI Pipelines**

Spark Structured Streaming + ETL operations enables real-time data pipelines.

Filters can discard irrelevant or corrupted events instantly.

Joins combine streaming data with reference datasets (e.g., user profiles).

Aggregations generate metrics like rolling averages, counts, or anomaly scores in real-time.

Impact: Supports AI models for recommendation systems, fraud detection, predictive maintenance, and dynamic pricing.

4. Model Monitoring and Post-Processing

After deploying models, Spark ETL pipelines can filter and aggregate incoming data to monitor model performance.

Joins allow integration of predicted outputs with ground truth to compute accuracy metrics.

Aggregations summarize metrics across regions, time periods, or product categories.

Impact: Ensures robust AI pipelines with continuous evaluation and improvement.

5 .Use Cases in Data Science Workflows

AI/DS Stage	**Spark ETL Role**	**Example**
Data Cleaning	Filters	Remove invalid customer transactions
Feature Engineering	Joins + Aggregations	Compute total spending per user across multiple datasets
Model Training	Aggregations	Aggregate past interactions for supervised learning
Real-Time Analytics	Filters + Joins + Aggregations	Fraud detection, recommendation engine
Model Evaluation	Joins + Aggregations	Compare predictions with actual outcomes to compute metrics

Introduction to PySpark

1. What is PySpark?

PySpark is the Python API for Apache Spark, a distributed, in-memory big data processing framework. It allows Python developers to leverage Spark's scalable and high-performance capabilities for processing large datasets. PySpark supports structured data processing, machine learning, streaming analytics, and graph processing through a unified framework.

2. Key Features of PySpark

Distributed Computing – Processes data across multiple nodes in a cluster, enabling scalability for large datasets.

In-Memory Processing – Data is processed in RAM, making operations much faster compared to disk-based frameworks.

High-Level APIs – Provides Python-friendly APIs for RDDs, DataFrames, and Spark SQL.

Fault Tolerance – Automatically recovers lost data using lineage information in RDDs and DataFrames.

Integration with Python Ecosystem – Works with NumPy, Pandas, Scikit-learn, and other Python libraries for machine learning and analytics.

Supports Multiple Workloads – Batch processing, real-time streaming, machine learning (MLlib), and graph processing (GraphX).

3. Components of PySpark

Spark Core – The foundation of Spark for distributed task scheduling, memory management, and fault tolerance.

Spark SQL – Enables structured data processing using DataFrames and SQL queries.

Spark Streaming – Processes real-time streaming data.

MLlib – Machine learning library for scalable algorithms.

GraphFrames/GraphX – Libraries for graph analytics and network processing.

4. PySpark Architecture

Driver Program – Runs the main Python script and coordinates tasks.

Cluster Manager – Allocates resources across worker nodes (e.g., YARN, Mesos, or Spark Standalone).

Worker Nodes – Execute tasks on partitions of data in parallel.

RDD/DataFrame Operations – Transformations and actions are applied, optimized using Catalyst and executed efficiently with Tungsten.

5. Advantages of PySpark

Ease of Use – Python syntax simplifies distributed data processing.

Scalability – Handles terabytes or petabytes of data efficiently.

Performance – Optimized execution with Catalyst and Tungsten.

Versatility – Supports ETL, machine learning, streaming, and graph analytics.

Integration – Works seamlessly with Hadoop, Hive, and cloud storage systems.

6. Simple PySpark Example

Python

```
from pyspark.sql import SparkSession

# Create SparkSession

spark = SparkSession.builder.appName("PySparkExample").getOrCreate()

# Load CSV into DataFrame

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

# Show first 5 rows

df.show()

# Perform aggregation

df.groupBy("department").avg("salary").show()
```

Explanation:

The script creates a Spark session, reads data into a DataFrame, displays the first rows, and performs an aggregation grouped by department.

This demonstrates PySpark’s ease of use, DataFrame operations, and distributed processing capabilities.

7. Applications of PySpark

Large-scale data analytics and ETL pipelines

Machine learning pipelines using MLlib

Real-time data processing with Spark Streaming

Graph analysis using GraphFrames

Integration with cloud data lakes and BI tools

Conclusion

PySpark combines the simplicity of Python with the power of Apache Spark’s distributed computing. Its high-level APIs, fault tolerance, and optimized execution engine make it ideal for processing large-scale data, building AI/ML pipelines, and handling real-time analytics in modern big data ecosystems.