Ministry of Higher Education and Scientific Research
University of Mila
Institute of Mathematics and Computer Science
Department of Computer Science
Master 2 I2A – Big Data
2025/2026

Chapter 3
Apache Spark

Presented by: Dr. Brahim Benabderrahmane

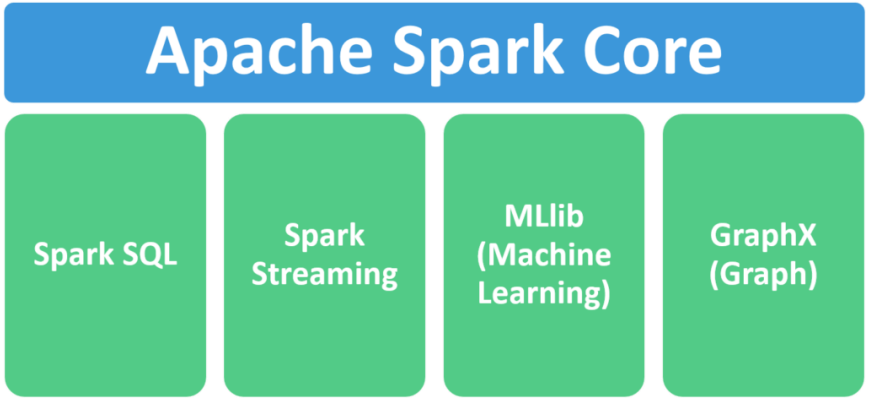
Table of Contents

01

Introduction to Apache Spark

Apache Spark – Unified Big Data Framework

- Developed by UC Berkeley's AMPLab, now an Apache top-level project.
- Open-source, distributed computing engine
- Supports batch, streaming, SQL, ML, and graph processing in one unified platform
- Compatible with Hadoop HDFS, Hive, and NoSQL systems
- Provides APIs in Python (PySpark), Scala, Java, and R



Why Do We Need Spark? The Limits of Hadoop MapReduce

1. Disk I/O Bottleneck:
MapReduce reads and writes to disk between each stage → very slow for iterative tasks (e.g., ML or analytics)

2. Complex Programming Model:
Developers must manually chain multiple MapReduce jobs → code is long and hard to maintain.

3. Limited to Batch Processing:
No built-in support for streaming or interactive analysis; each job must complete before the next starts.

4. High Latency:
Job setup and scheduling overheads make small tasks inefficient.

5. Lack of Flexibility:
Only one computation model (Map → Reduce); no native APIs for SQL, ML, or graphs.

What Is Apache Spark? Apache Spark in a Nutshell

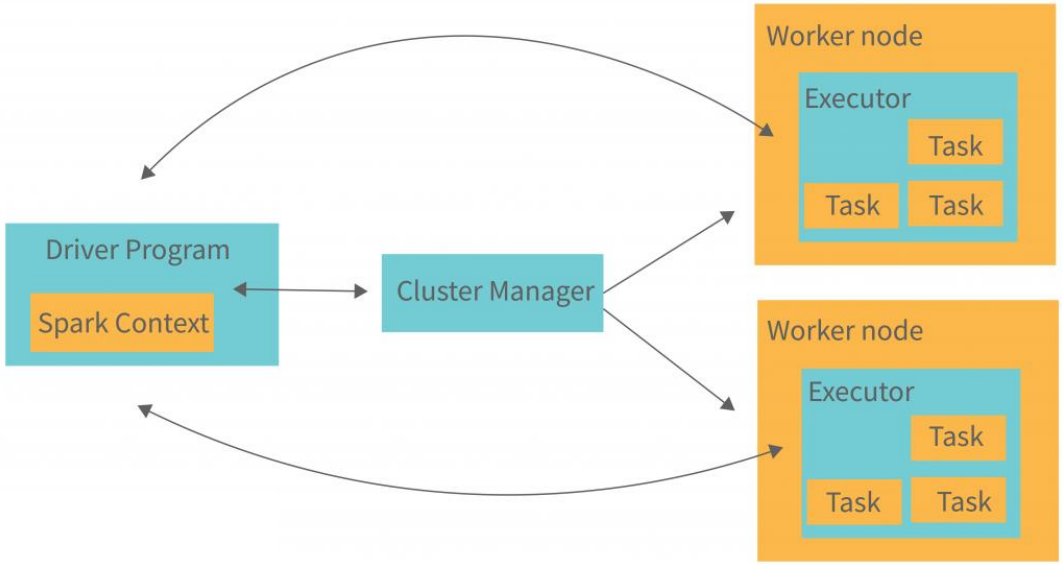
- A unified analytics engine for **large-scale distributed data processing**.
- Built for **speed**, **ease of use**, and **versatility.**
- Processes data **in-memory** instead of writing intermediate results to disk.
- Works with **structured, semi-structured**, and **unstructured** data.
- Runs on **clusters**, locally, or in the **cloud** (AWS EMR, Databricks, GCP Dataproc).

The Spark Ecosystem – One Platform, Many Libraries

Spark Core	**Foundation providing RDDs & distributed execution.**
Spark SQL	**Query structured data using SQL or DataFrames.**
Spark Streaming / Structured Streaming	**Real-time data.**
MLlib	**Scalable machine learning library.**
GraphX	**Graph computation engine (PageRank, network analysis).**

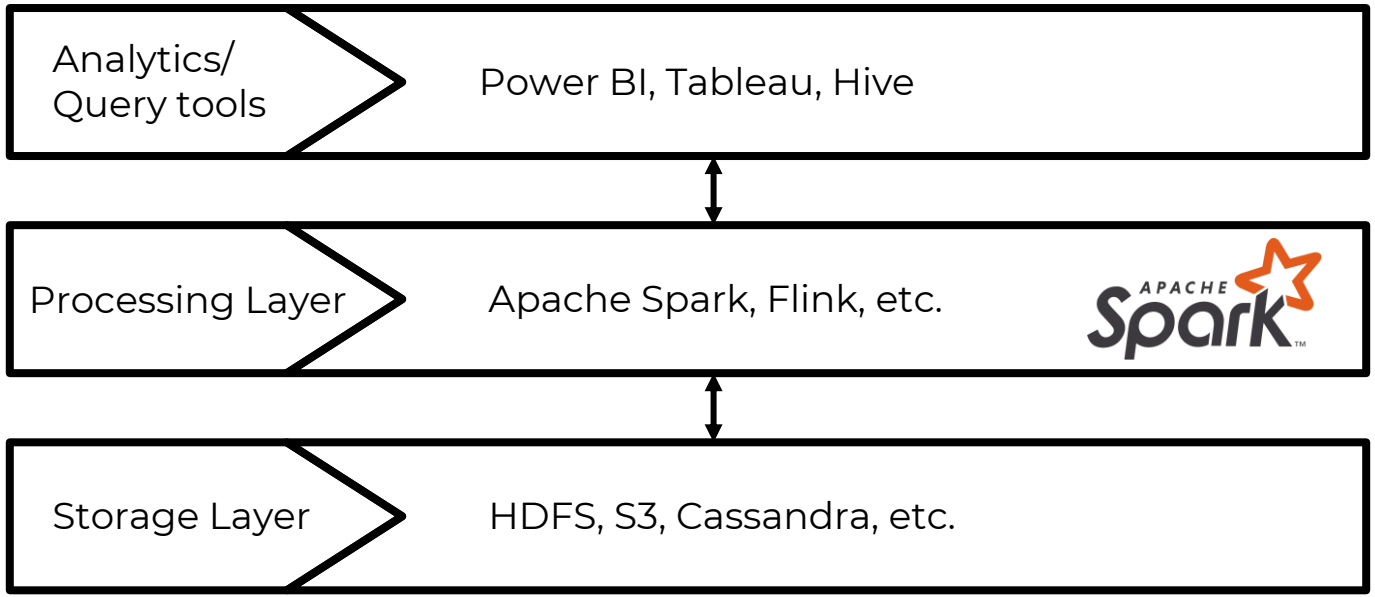
Inside Spark – Driver, Executors, and Cluster Manager

- **Driver Program**: runs the user's main code, creates the execution plan.
- **Cluster Manager**: allocates resources (Standalone, YARN, Kubernetes, Mesos).
- **Executors**: run tasks, store data in memory across nodes.
- **SparkSession** acts as the entry point to this ecosystem.



Spark and the Modern Data Architecture

- **Storage layer:** HDFS, S3, NoSQL (Cassandra, MongoDB).
- **Processing layer:** Spark (Core, SQL, MLlib, Streaming).
- **Query & Visualization layer:** Hive, Presto, Power BI, Tableau.
- Spark acts as the **engine** that connects data at rest



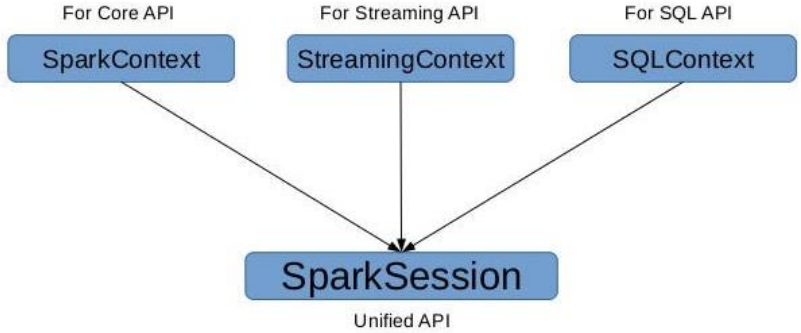
02

Core Concepts and Data Abstractions

SparkSession – The Unified Gateway to Apache Spark

- Introduced in **Spark 2.0** to simplify access to all Spark components.
- Replaces older objects like SparkContext, SQLContext, and HiveContext.
- Acts as the **single entry point** for DataFrame, SQL, Streaming, and ML operations.
- Automatically creates and manages the underlying **SparkContext**.

Through **spark**, we can read data, create DataFrames, run SQL queries, and more.



Commonly created in pythong as:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MyApp").getOrCreate()
```

RDDs (Resilient Distributed Datasets) – Spark's Foundation

RDD = Resilient Distributed Dataset → the fundamental data abstraction in Spark.

Represents an **immutable**, **distributed collection** of objects partitioned across cluster nodes.

Enables **fault tolerance** through lineage (rebuild lost partitions).

Supports **two types** of **operations**:

- **Transformations**: create a new RDD (e.g., map, filter, flatMap).
- **Actions**: trigger computation (e.g., collect, count, saveAsTextFile).

Example:

```
rdd = spark.sparkContext.textFile("data.txt")
words = rdd.flatMap(lambda x: x.split())
counts = words.map(lambda x: (x, 1)).reduceByKey(lambda a, b: a + b)
counts.collect()
```

From RDDs to DataFrames and Datasets – Simplifying Distributed Data

- **RDDs** were powerful but too *low-level* (manual schema, hard to optimize).
- **DataFrames** introduce a tabular structure, like a distributed pandas DataFrame or SQL table.
- **Datasets** (in Scala/Java) add *compile-time* type safety on top of DataFrames.
- In **PySpark**, we mainly use **DataFrames** (Dataset API is implicit).

- Key benefits over RDDs:

1. Automatic **schema** and **column operations**
2. **Catalyst Optimizer** → automatic query optimization
3. **Tungsten Engine** → efficient memory management
4. **Ease of use**: simpler syntax, SQL-like queries

Example:

```
df = spark.read.csv("sales.csv", header=True, inferSchema=True)
df.groupBy("region").sum("revenue").show()
```

Spark Operations – Transformations vs Actions

Spark operations are divided into two main categories:

1. **Transformations (lazy):**
- Create a *new* dataset from an existing one.
- Examples: **map**(), **filter**(), **flatMap**(), **groupBy**(), **select**().
- They *build* the computation plan but don't execute it immediately.

2. **Actions (trigger execution):**
- Return a value to the driver or write data to storage.
- Examples: **collect**(), **count**(), **show**(), **saveAsTextFile**().
- They *trigger Spark's DAG scheduler* to execute all pending transformations.

Example:

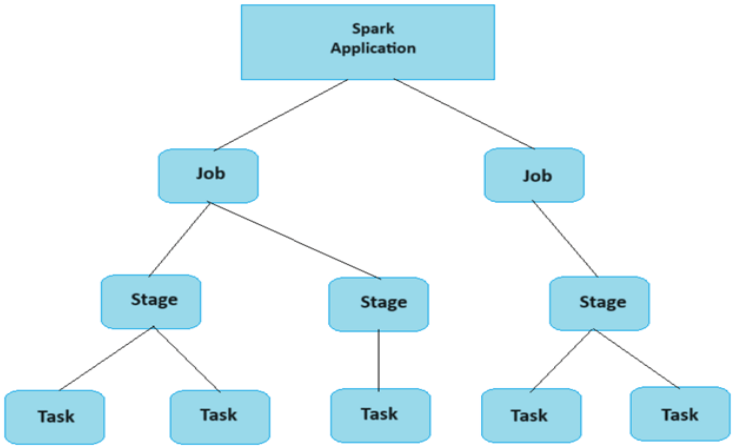
```
rdd = spark.sparkContext.parallelize([1,2,3,4])
rdd2 = rdd.map(lambda x: x * 2)   # Transformation
rdd2.collect()                    # Action
```

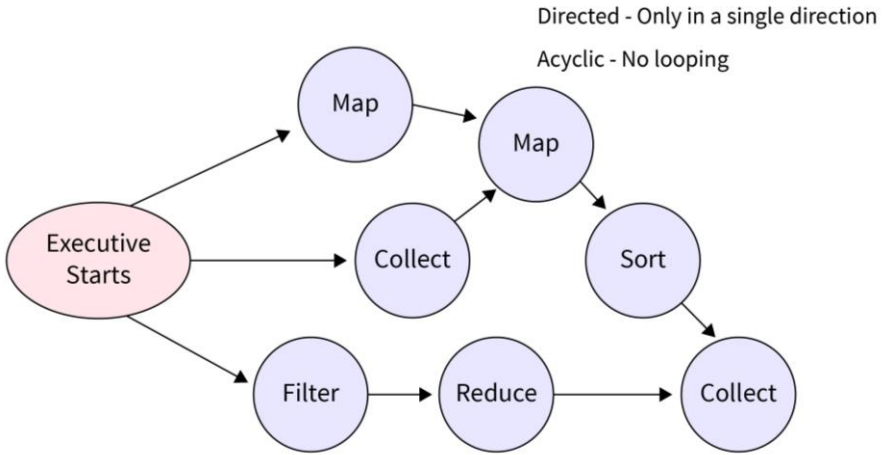
Lazy Evaluation – How Spark Thinks Before It Acts

- Spark doesn't execute operations immediately; it **builds a logical plan** (DAG) first.
- **Transformations** are *recorded*, not run.
- **Actions** trigger Spark to optimize and run everything in one go.
- Benefits:

1. **Optimization:** Spark rearranges and combines operations for efficiency.
2. **Reduced I/O:** Fewer data passes between stages.
3. **Fault Tolerance:** Execution plan can be re-evaluated if something fails.

- The **Directed Acyclic Graph (DAG)** represents this computation plan.





Two Ways to Say the Same Thing: RDD vs DataFrame

Concept	RDD Approach	DataFrame Approach
Abstraction Level	Low – manual transformations	High – structured API
Ease of Use	Code-heavy, functional	SQL-like, concise
Optimization	Manual	Automatic (Catalyst)
Typical Use Case	Custom logic / fine control	Analytics / ML / SQL queries

03

Spark Execution Model

From Code to Execution: Spark's DAG (Directed Acyclic Graph)

Idea:
When you write Spark code (with transformations like map, filter, reduceByKey), Spark doesn't execute them immediately.
Instead, it builds a **Directed Acyclic Graph (DAG)**: a logical plan describing how data flows from the source to the final result.

How It works?

1. **Logical Plan Construction:**
   Spark records transformations as a sequence of nodes and edges, **not executed** yet.
2. **Optimization:**
   The **DAG Scheduler** analyzes the plan, combines narrow dependencies, and splits it into *stages*.
3. **Physical Execution:**
   The Task Scheduler sends each stage's tasks to executors across the cluster.
4. **Action Trigger:**
   When an *action* (like collect(), count(), or saveAsTextFile()) is called, Spark finally executes the DAG.

Key Benefits:
Optimization: Eliminates unnecessary data shuffles and recomputations.
Fault Tolerance: If a node fails, Spark recomputes missing partitions from the DAG lineage.
Efficiency: Executes multiple transformations in memory before writing to disk.

Lazy Evaluation in Action

Spark doesn't execute transformations immediately.
Instead, it **builds a logical plan (DAG)** that records ***what*** to do, **not *how*** *or when*.
Execution only happens when an **action** (like *count(), collect()*, or *saveAsTextFile()*) is called.

This *lazy evaluation* allows Spark to:

- **Optimize** the execution plan (combine transformations, minimize shuffles)
- **Avoid unnecessary computation**
- **Recover efficiently from failures**

1. **Program Definition (Lazy Stage)**

```
rdd = sc.textFile("data.txt")
words = rdd.flatMap(lambda x: x.split(" "))
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```

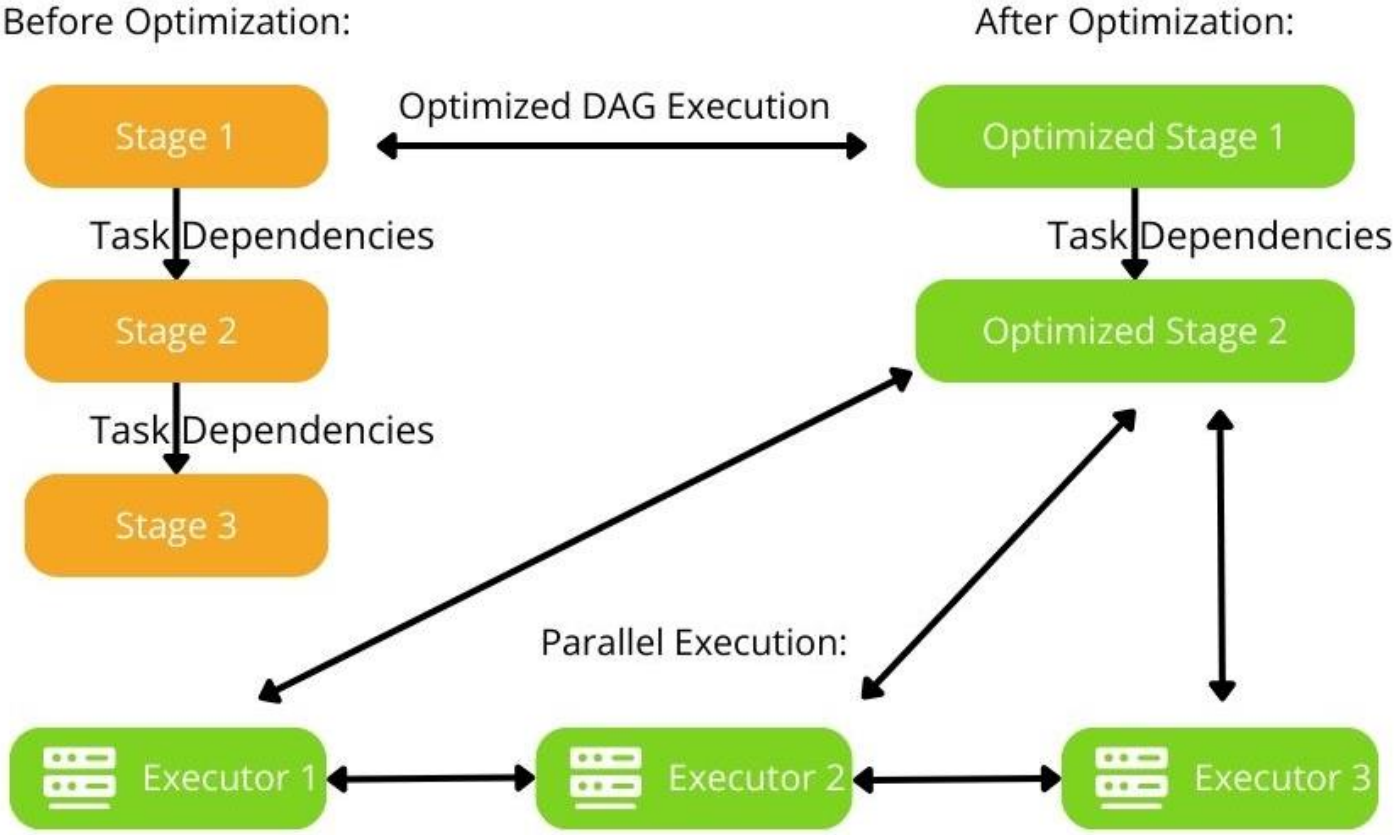
Nothing executes yet! Spark is just **building a DAG**

2. **Action Trigger (Execution Stage)**

```
counts.collect()
```

Only now Spark **executes the DAG**: splits into stages, sends tasks to executors, and optimizes the workflow.

Apache Spark DAG Optimization



Stages and Tasks: How Spark Splits Work

Once Spark's DAG is optimized, the DAG Scheduler breaks the computation into **stages**.
Each stage representing a group of transformations that can be executed **without data shuffling**.
Within each stage, Spark launches **tasks**, one per data partition across the cluster's executors.

How it works (step-by-step):

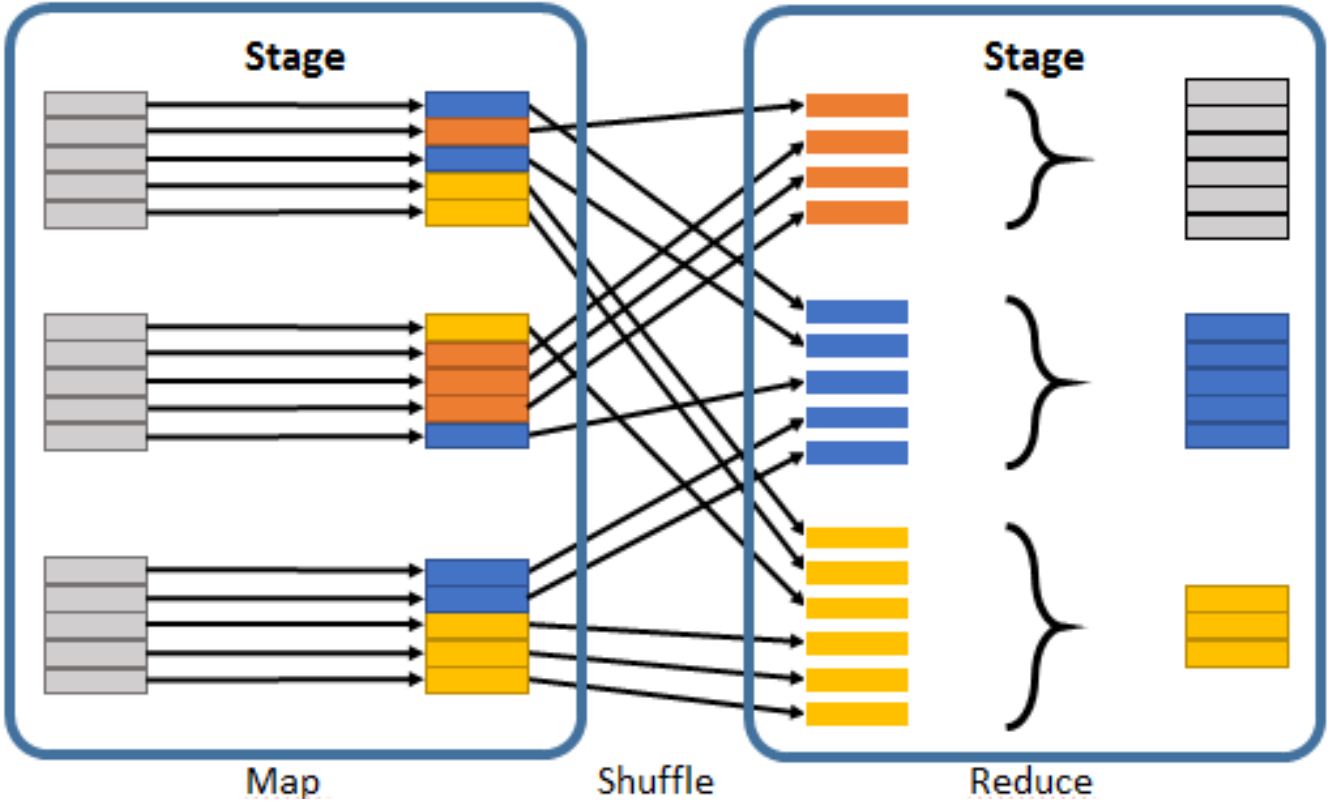
1. **Stage Creation:**
- Each **narrow dependency** (like map, filter) stays within the same stage.
- Each **wide dependency** (like groupByKey, reduceByKey) triggers a **shuffle**, which creates a new stage.

2. **Task Generation:**
- A **task** is the smallest execution unit : one per data partition.
- Each executor runs several tasks in parallel on its cores.

3. **Scheduling:**
- The **DAG Scheduler** handles stage dependencies.
- The **Task Scheduler** sends tasks to executors on worker nodes.

4. **Execution Flow:**
- Stage 1: narrow transformations on local partitions
- Shuffle (data exchange)
- Stage 2: aggregation, joins, or reduce tasks
- Result collected or written to storage

Spark Shuffle



Driver vs Executors: Roles Explained

Spark applications follow a **master–worker architecture**:
- **Driver:** The “brain” of the Spark application, it coordinates everything.
- **Executors:** The “muscles”, they perform the computations on the data.

Together, they handle the full data processing pipeline.

1. Driver Program (Master Node):
- Runs the **main()** function of your Spark application.
- **Creates the SparkContext**, which connects to the cluster manager.
- **Builds the DAG** of transformations (logical plan).
- **Schedules tasks** into stages and distributes them to executors.
- **Collects results** from executors.
- Example: When you call collect(), the driver gathers data from all executors.

2. Executors (Worker Nodes):
- Each executor runs on a worker machine.
- Responsible for **executing tasks** assigned by the driver.
- **Stores data in memory** (RDD partitions) for caching and reuse.
- **Reports progress** and results back to the driver.
- If an executor fails, Spark can **relaunch** tasks on another node (fault tolerance).

Caching and Persistence (memory/disk levels)

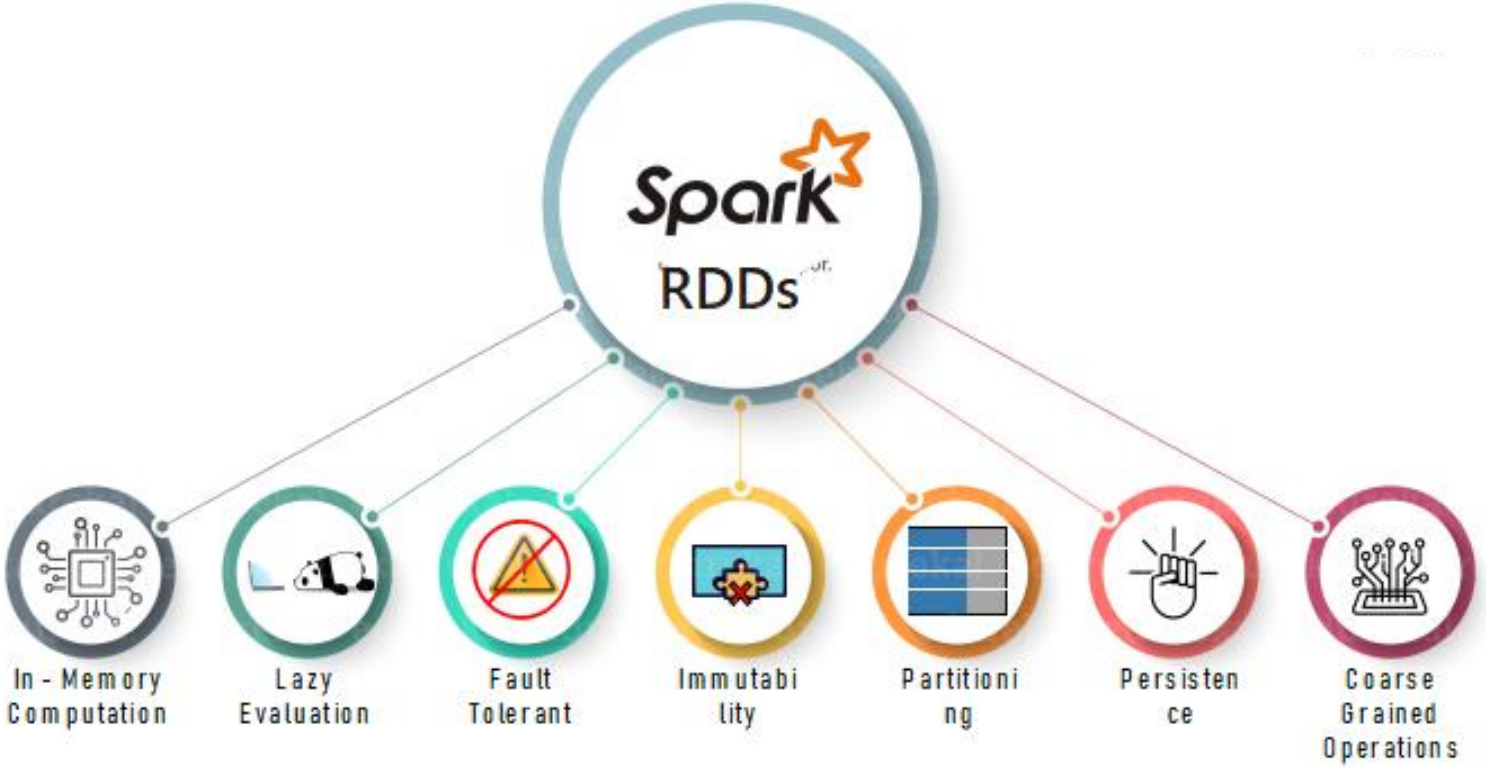
Spark allows you to **cache** or **persist** datasets (RDDs, DataFrames, or Datasets) in memory or on disk. This helps **reuse results** across multiple actions (e.g., count, collect, or join), avoiding recomputation and speeding up iterative jobs like ML training or graph analysis.

Concept	Description
Caching	Quick, default option using memory storage (e.g., *rdd.cache()*)
Persistence	More control over *where and how* data is stored using storage levels
Recomputation Avoidance	Prevents Spark from re-running the entire lineage DAG
Storage Levels	*MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, OFF_HEAP*
Automatic Recovery	If cached data is lost (node failure), Spark recomputes it using lineage

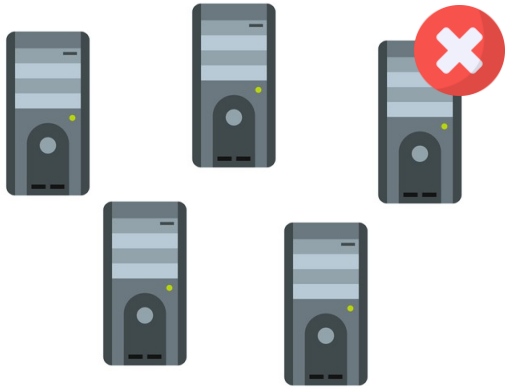
Example:
Once cached, Spark reuses this data for subsequent transformations.

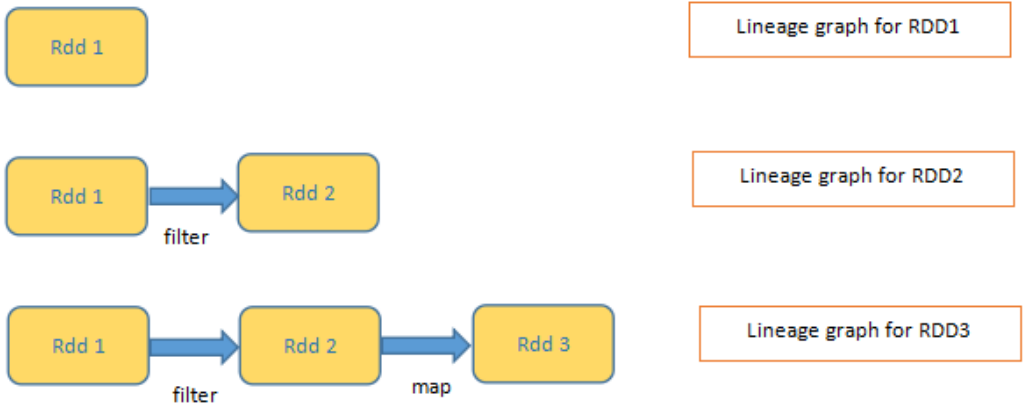
```
rdd = sc.textFile("data.txt")
rdd.cache()  # or rdd.persist(StorageLevel.MEMORY_AND_DISK)
result = rdd.filter(lambda x: "Spark" in x).count()
```

Spark RDDs


Spark
RDDs
In - Memory Computation
Lazy Evaluation
Fault Tolerant
Immutability
Partitioning
Persistence
Coarse Grained Operations

Fault Tolerance: RDD Lineage & Re-computation





When a Node Fails…

- *Lost partition of an RDD*
- *Re-executing the whole job would be expensive*
- *Data replication (like Hadoop) consumes too much storage*

Spark Uses *Lineage* for Recovery:

- Keeps a **lineage graph** of all transformations (*map*, *filter*, etc.)
- When data is lost: recompute only that partition
- No need for full job restart or data replication
- Optional *checkpointing* for very long workflows

04

Practical Spark Programming

Creating an RDD or DataFrame in PySpark

Before performing any operations in Spark, we need to **create a SparkSession** — the main entry point to work with the Spark engine.
From it, we can create **RDDs** (Resilient Distributed Datasets) or **DataFrames**, which are the core data abstractions in Spark.

- **RDD (Resilient Distributed Dataset):**
  The **fundamental Spark data structure** represents a distributed collection of elements across nodes.
  RDDs are **immutable**, **partitioned**, and support **transformations** and **actions**.
  They offer **fine-grained control**, but less optimization.
- **DataFrame:**
  A **higher-level abstraction** built on top of RDDs.
  It represents data in **tabular form** (rows and columns) with named fields.
  DataFrames are automatically optimized by Spark's **Catalyst engine**, making them faster and easier to use.
- **Data Sources:**
  Spark can read data from multiple sources — local files, HDFS, Hive tables, JSON, Parquet, and more.

Abstraction	Format	Main Use	Optimization
RDD	Distributed objects	Low-level transformations	Manual
DataFrame	Structured (tabular)	SQL-like operations	Automatic (Catalyst)

Example of Creating an RDD or DataFrame in PySpark

Example:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("CreationExample").getOrCreate()

# --- Create an RDD from a Python list ---
rdd = spark.sparkContext.parallelize([10, 20, 30, 40])

# --- Create a DataFrame from local data ---
data = [("Alice", 25), ("Bob", 30)]
df = spark.createDataFrame(data, ["Name", "Age"])

# --- Load data from an external CSV file ---
df_csv = spark.read.csv("people.csv", header=True, inferSchema=True)
```

Observation: Spark builds **a logical plan** when these are created, actual computation happens **later**, during actions (lazy evaluation).

Transformations: *map(), filter(), flatMap(), reduceByKey()*

Transformations create a **new RDD or DataFrame** from an existing one.
They are **lazy** : nothing runs until an **action** is called.
Common transformation types:

- **map()** : applies a function to each element
- **filter()** : keeps elements that satisfy a condition
- **flatMap()** : similar to map, but flattens the results (useful for splitting text into words)
- **reduceByKey()** : combines values with the same key (for pair RDDs)

```
# Sample RDD
rdd = spark.sparkContext.parallelize(["apple banana", "banana orange", "apple orange banana"])

# flatMap(): split each line into words
words = rdd.flatMap(lambda line: line.split(" "))

# map(): create (word, 1) pairs
pairs = words.map(lambda w: (w, 1))

# reduceByKey(): count occurrences
word_counts = pairs.reduceByKey(lambda a, b: a + b)

# filter(): keep only words starting with 'a'
filtered = word_counts.filter(lambda x: x[0].startswith('a'))

# Collect results
print(filtered.collect())
```

Actions: collect(), count(), saveAsTextFile()

Actions trigger the **execution** of all previous **transformations**.
They return results **to the driver** or **save output externally**.
Spark evaluates transformations **lazily**, so actions are what **launch the DAG** execution.

Common Actions:

- **collect():** Returns the entire RDD to the driver (⚠ use with small datasets).
- **count():** Returns the number of elements in the RDD.
- **saveAsTextFile(path):** Saves results to HDFS or local storage.

```
# Continuing from the previous example
word_counts = pairs.reduceByKey(lambda a, b: a + b)

# Action 1: collect results
print("Collected results:", word_counts.collect())

# Action 2: count total words
print("Word count:", word_counts.count())

# Action 3: save output to a text file
word_counts.saveAsTextFile("output/wordcount_results")
```

Example 1: WordCount (step-by-step)

High-level steps (what we want Spark to do):

1. **Read** raw text files into an RDD (one line per record).
2. **Tokenize** lines into words (*flatMap*) : produce a stream of words.
3. **Map** each word to a pair (word, 1) (*map*).
4. **Aggregate** counts for each word (*reduceByKey*) : causes a **shuffle**.
5. **Collect or save** the final counts (*collect()* or *saveAsTextFile*) action triggers execution.

Execution notes (DAG → Stages → Tasks):

- **Logical DAG:** textFile → flatMap → map → reduceByKey → action.
- **Stage split:** Narrow ops (*flatMap*, *map*) are in **Stage 1**; *reduceByKey* (wide → shuffle) starts **Stage 2**.
- **Tasks:** One task per partition per stage. If input has 8 partitions: Spark launches ~8 tasks in Stage 1 and ~8 tasks in Stage 2 (post-shuffle).

Optimization tips:

- Use *mapPartitions* for heavy per-partition setup. (instead of *map*)
- Use *persist*() if you reuse intermediate RDDs. (to keep the RDD in cache instead of storing it in memory)
- Prefer *reduceByKey* over *groupByKey* to avoid excessive network I/O. (reduceByKey applies aggregation first, groupByKey sends all values)

Fault tolerance: If a partition is lost, Spark recomputes it from lineage (only the missing partitions are recomputed).

Example 1: WordCount (Code)

```
# SparkSession assumed: spark
# 1: read
rdd = spark.sparkContext.textFile("hdfs://data/corpus/*")

# 2: tokenize
words = rdd.flatMap(lambda line: line.split())

# 3: pair
pairs = words.map(lambda w: (w.lower(), 1))

# 4: aggregate (shuffle)
counts = pairs.reduceByKey(lambda a, b: a + b)

# 5: action -> execute
counts.saveAsTextFile("hdfs://output/wordcount")
```

Example 2: Simple Sales Analytics using DataFrames

Spark DataFrames provide a **structured and optimized** way to handle big data, similar to SQL tables but distributed across the cluster.
They offer:

- **Automatic optimization** via the *Catalyst optimizer*
- **Easy SQL-like syntax** (select, groupBy, agg, etc.)
- **Interoperability** with pandas and databases

Let's explore a small example of computing total and average sales per product.

We start from a CSV file containing: (Product, Category, Price, Quantity)
Goal: to compute:

- total revenue per product
- average price per category

Product	Category	Price	Quantity
TV	Electronics	800	2
Phone	Electronics	500	3
..	..	..	..

Example 2: Simple Sales Analytics using DataFrames (Code)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, avg

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

# Load dataset
df = spark.read.csv("sales.csv", header=True, inferSchema=True)

# Compute revenue per product
revenue = df.withColumn("Revenue", col("Price") * col("Quantity"))
summary = revenue.groupBy("Product").agg(sum("Revenue").alias("TotalRevenue"))

# Average price per category
avg_price = df.groupBy("Category").agg(avg("Price").alias("AvgPrice"))

summary.show()
avg_price.show()
```

Using Caching and Reusing DataFrames

In Spark, **caching** (or **persisting**) allows data to be **kept in memory (or disk)** after the first computation. This avoids recomputing the same DataFrame/RDD multiple times, especially for **iterative algorithms**, **interactive analysis**, or **reused data**.
Key benefits:

- **Performance boost**: Faster access to reused datasets
- **Avoid recomputation**: Spark skips re-running the DAG lineage
- **Control**: Choose where to store (memory, disk, or both)

Method	Description	Use Case
MEMORY_ONLY	Fastest, but may recompute if not enough memory	Small reusable datasets
MEMORY_AND_DISK	Falls back to disk if memory is full	Larger datasets
DISK_ONLY	Stores only on disk	When memory is limited
NONE	No caching (default)	One-time computations

```python
df.cache()                                  # Cache a DataFrame in memory

from pyspark import StorageLevel
df.persist(StorageLevel.MEMORY_AND_DISK) # Or persist with a storage level

df.count()                                  # Trigger computation

filtered = df.filter(df["Price"] > 1000) # Later reuse the same DataFrame
filtered.show()
```

05

MLlib
and Machine Learning Pipelines

What Is MLlib? Overview of Spark ML

MLlib is Spark's machine learning library, designed to scale across very large datasets using Spark's distributed computing engine.
It provides a unified, high-level API that allows developers to build machine learning pipelines for a variety of tasks, without worrying about low-level cluster management or data partitioning.

MLlib supports:

- **Classification:** Assigning labels to data points (e.g., spam detection, sentiment analysis).
- **Regression:** Predicting continuous values (e.g., price forecasting).
- **Clustering:** Grouping similar data points (e.g., customer segmentation).
- **Recommendation:** Suggesting items based on user behavior (e.g., movie or product recommendations).
- **Feature extraction and transformation:** Converting raw data into formats suitable for ML models.

With MLlib, the same API works for both **batch** and **streaming** machine learning, making it flexible for real-time analytics as well as large-scale batch computations.

MLlib Evolution: From RDD-based to DataFrame-based API

MLlib originally offered an **RDD-based API**, which allowed distributed machine learning on Spark, but it had several limitations:

- **Verbose code:** Developers had to manually manage RDDs and feature transformations.
- **Limited optimizations:** Spark couldn't automatically optimize operations, leading to slower execution for large datasets.

To improve usability and performance, Spark introduced the **DataFrame-based API**:

- **Simpler syntax:** Operations are expressed directly on DataFrames, making code more concise.
- **Automatic optimization:** Spark's **Catalyst optimizer** automatically plans query execution for efficiency.
- **Better integration:** Works seamlessly with Spark SQL and Pipelines.

Both APIs still exist, but **new projects are strongly recommended to use the DataFrame-based API**.

Example: Logistic Regression that once required several lines with RDDs can now be implemented in just **3–4 lines** with the DataFrame API, including feature assembly and model training.

Spark ML Workflow: Core Concepts

MLlib organizes machine learning workflows using **three main components**:

1. Transformer:

A Transformer takes a DataFrame as input and **returns a new DataFrame** with one or more additional columns.

Examples:

- VectorAssembler : combines multiple feature columns into a single feature vector.
- StandardScaler : scales features to a standard range.
- A trained model (after fitting) also acts as a Transformer to generate predictions.

2. Estimator:

An Estimator is an **algorithm that can be trained** on a DataFrame to produce a Model (Transformer).

Examples:

- LogisticRegression.fit() : trains a logistic regression model.
- KMeans.fit() : trains a clustering model.

3. Pipeline:

- A Pipeline chains **Transformers and Estimators** in a specific order.
- Allows automation of preprocessing, model training, and prediction in a **single workflow**.
- Reduces errors and simplifies code when applying the same process to new datasets.

Data → Transformer(s) → Estimator (training) → Model (Transformer) → Predictions

Example: Logistic Regression with DataFrame API

This example demonstrates how **simple and concise** the DataFrame-based ML API is in Spark MLlib. Using only a few lines, you can transform features, train a logistic regression model, and make predictions:

Step 1: Feature Assembly

- Combine multiple feature columns into a single feature vector using VectorAssembler.

Step 2: Train the Model

- Create a LogisticRegression estimator specifying the features and label columns.
- Fit the model on the training DataFrame.

Step 3: Make Predictions

- Apply the trained model to the training DataFrame to generate predictions.
- Display the label and predicted values to verify results.

Key Takeaway:

- The DataFrame API **removes boilerplate code**, integrates seamlessly with Pipelines, and makes ML workflows easy to understand and maintain.

```python
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import VectorAssembler

# Assemble features
assembler = VectorAssembler(inputCols=["feature1", "feature2"], outputCol="features")
train_df = assembler.transform(df)

# Train logistic regression model
lr = LogisticRegression(featuresCol="features", labelCol="label")
model = lr.fit(train_df)

# Make predictions
predictions = model.transform(train_df)
predictions.select("label", "prediction").show()
```

Example: KMeans Clustering (Conceptual + Small Code)

KMeans is an **unsupervised learning algorithm** used to group similar data points into clusters. Spark MLlib provides a **consistent DataFrame-based API**, making it simple to switch between different ML algorithms.

Use Case Example:

- Customer segmentation: group customers with similar buying patterns.
- Any dataset with numerical features can be clustered using **KMeans**.

Steps in Spark MLlib:

1. Create a KMeans Estimator:

- Specify the number of clusters (k) and the column containing feature vectors.

2. Train the Model:

- Fit the model on the DataFrame to compute cluster centers.

3. Make Predictions:

- Transform the DataFrame to assign each row to a cluster.

Key Takeaway:

- MLlib uses the **same Transformer / Estimator API** across classification, regression, and clustering, simplifying the workflow for developers.

```
from pyspark.ml.clustering import KMeans
# Define KMeans estimator
kmeans = KMeans(k=3, featuresCol="features")
# Fit the model
model = kmeans.fit(df)
# Assign clusters to each row
predictions = model.transform(df)
predictions.select("features", "prediction").show()
```

Feature Engineering in MLlib

Feature engineering is a **critical step before training any machine learning model**, as it transforms raw data into formats that models can interpret efficiently. Spark MLlib provides **built-in Transformers** to handle most preprocessing tasks, fully compatible with ML Pipelines.

Key Transformers for Preprocessing:

- **StringIndexer:** Converts categorical string columns into numerical indices.
- **OneHotEncoder:** Encodes categorical variables as binary vectors.
- **VectorAssembler:** Combines multiple feature columns into a single feature vector.
- **StandardScaler:** Normalizes features to have zero mean and unit variance.
- **PCA (Principal Component Analysis):** Reduces feature dimensionality while retaining variance.

Integration in Pipelines:

- All these transformers can be chained with estimators in a **Pipeline**, ensuring preprocessing and model training are applied in a single workflow.
- This reduces code complexity and avoids repeated transformations.

```
from pyspark.ml.feature import StringIndexer, VectorAssembler
# Encode categorical feature
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
# Combine features into a single vector
assembler = VectorAssembler(inputCols=["feature1", "feature2", "categoryIndex"], outputCol="features")
```

Full Pipeline Example

1.Preprocessing:

- StringIndexer encodes categorical columns into numerical indices.
- VectorAssembler combines feature columns (numeric + encoded categorical) into a single feature vector.

2.Model Training:

- LogisticRegression estimator trains a classification model on the processed data.

3.Pipeline Execution:

- A Pipeline chains the preprocessing stages and the estimator.
- pipeline.fit(train_df) trains all stages sequentially and produces a model object.

4.Predictions:

- The trained pipeline model is applied to test_df with model.transform(test_df).
- Predictions include the original label, features, and predicted value.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer,
VectorAssembler
from pyspark.ml.classification import LogisticRegression

# Preprocessing
indexer = StringIndexer(inputCol="category",
outputCol="categoryIndex")
assembler = VectorAssembler(inputCols=["feature1",
"feature2", "categoryIndex"], outputCol="features")

# Model
lr = LogisticRegression(labelCol="label")

# Pipeline
pipeline = Pipeline(stages=[indexer, assembler, lr])

# Train pipeline
model = pipeline.fit(train_df)

# Predictions
predictions = model.transform(test_df)
```

How Spark Distributes ML Computation

Spark MLlib leverages **cluster computing** to parallelize machine learning computations, which makes it **fast and scalable** for large datasets.

Key Concepts:

1. Data Partitioning:

- Input data is automatically **split across partitions** distributed to different executors.
- Each partition contains a subset of the dataset for local processing.

2. Local Computation on Executors:

- Executors perform **local computations**, such as calculating gradients or intermediate statistics.
- Reduces network communication since most operations are performed in-memory on local data.

3. Driver Aggregation:

- The **Driver node** collects local results from executors and aggregates them to update the global model.
- Ensures synchronization for iterative algorithms (e.g., gradient descent).

4. In-Memory Speed Advantage:

- Unlike Hadoop MapReduce, Spark keeps data in memory, minimizing disk I/O.
- This dramatically improves training time for iterative ML algorithms.

Model Evaluation in Spark ML

Evaluating machine learning models is **critical** to ensure they perform well on unseen data. Spark MLlib provides a set of **unified evaluators** that work across different types of models.

Key Evaluators:

1. BinaryClassificationEvaluator: for binary classification tasks.

- Computes metrics like **AUC** or **area under ROC**.

2. RegressionEvaluator: for regression models.

- Computes metrics like **RMSE, MAE, R^2**.

3. MulticlassClassificationEvaluator: for multi-class classification.

- Computes metrics like **accuracy**.

Example Code:

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
# Evaluate a binary classification model
evaluator = BinaryClassificationEvaluator(labelCol="label")
auc = evaluator.evaluate(predictions)
print("AUC:", auc)
```

Key Takeaways:

- Spark MLlib evaluators provide a **consistent API** across different models.
- Easy integration into Pipelines for **automatic evaluation**.
- Helps compare different models and tune hyperparameters efficiently.

Integration and Scalability

The technology that fits this description is **Apache Spark MLlib** (or Spark ML), which is Apache Spark's scalable machine learning library.

Key features and integrations include:

- **Works across large clusters**: MLlib is built on top of the Apache Spark engine, which is designed for distributed computing across large clusters and handles massive datasets efficiently.
- **Integrates with Spark SQL and Structured Streaming**: The primary API for modern MLlib is DataFrame-based, allowing seamless integration with Spark SQL for data manipulation and Structured Streaming for real-time model application.
- **MLflow (for tracking experiments)**: MLflow provides a robust integration with Spark MLlib, enabling seamless model tracking, experiment management, and deployment of Spark ML models.
- **Can export models to PMML or ONNX**: MLflow allows for the conversion and export of Spark MLlib models to the ONNX (Open Neural Network Exchange) format for flexible, cross-platform deployment and optimized inference. Spark MLlib also supports export to PMML (Predictive Model Markup Language) for compatibility with other statistical modeling tools.

06

Spark in the Modern Big Data Landscape

Structured Streaming in Spark

Spark Structured Streaming allows you to **process live data streams in near real-time** while using the **same DataFrame API** as batch processing, making it easy to transition from batch to streaming workflows.

Key Features:

1.Continuous Data Processing:

- Handles incoming data in small micro-batches or continuous mode.
- Ensures low-latency processing for time-sensitive applications.

2.Event-Time Processing:

- Supports event-time semantics to handle **late-arriving data**.
- Watermarks are used to manage out-of-order events.

3.Window Operations:

- Allows aggregation over **time windows**, e.g., counting events per minute.
- Useful for metrics like moving averages or anomaly detection.

4.Integration with Streaming Sources:

- Works with **Kafka**, **Kinesis**, or **socket streams**.
- Can write results to files, databases, dashboards, or other sinks.

Example Use Case:

- Real-time **transaction monitoring**: detect fraudulent behavior.
- **Log analysis**: monitor errors or trends in server logs as they arrive.

Spark SQL – Querying DataFrames Like Tables

Spark SQL allows you to **query structured data using SQL syntax or the DataFrame API**, providing a familiar interface for analysts while taking advantage of Spark's distributed computing power.

Key Features:

1.Flexible Querying:

- Supports **SQL queries**, including joins, aggregations, and complex queries.
- DataFrames provide a **programmatic API** that maps directly to SQL operations.

2.Multiple Data Sources:

- Can query structured data in **HDFS, Parquet files, Hive tables**, and external databases.
- Makes it easy to integrate Spark into existing data ecosystems.

3.Performance Optimization:

- **Catalyst optimizer** automatically rewrites and optimizes queries for efficiency.
- Applies **predicate pushdown, constant folding, and physical plan optimizations**.

Example Use Case:

- Analyzing sales data from multiple Hive tables using SQL.
- Performing complex aggregations across datasets stored in Parquet files.

Spark Integration with Big Data Ecosystem(Spark + HDFS + Hive + Kafka)

Spark is designed to work seamlessly within the **big data ecosystem**, enabling both **batch and real-time processing**.

Key Integration Points:

1. HDFS (Hadoop Distributed File System):

- Spark can **read and write large datasets** stored in HDFS.
- Enables distributed storage and high availability for big data.

2. Hive Tables:

- Use **Spark SQL** to query Hive tables directly.
- Supports **joins, aggregations, and complex queries** on existing data warehouses.

3. Kafka Streams:

- Integrates with **Kafka** for real-time streaming pipelines.
- Spark can process streaming events and perform **analytics on-the-fly**.

4. Hybrid Workflows:

- Combine **batch processing** from HDFS/Hive with **real-time streaming** from Kafka.
- Supports end-to-end **data pipelines** in one unified framework.

Example Use Case:

- Real-time analytics dashboard combining historical sales data from Hive and live transactions from Kafka.

Performance Architecture – Catalyst & Tungsten

1. Catalyst Optimizer:

Optimizes **logical and physical query plans** automatically.
Performs transformations such as:

- **Predicate pushdown:** filters applied as early as possible.
- **Expression simplification:** constant folding, eliminating unnecessary computations.
- **Efficient join strategies:** chooses the best join method based on data size and distribution.
- Ensures that both SQL and DataFrame queries are executed **efficiently**.

2. Tungsten Engine:

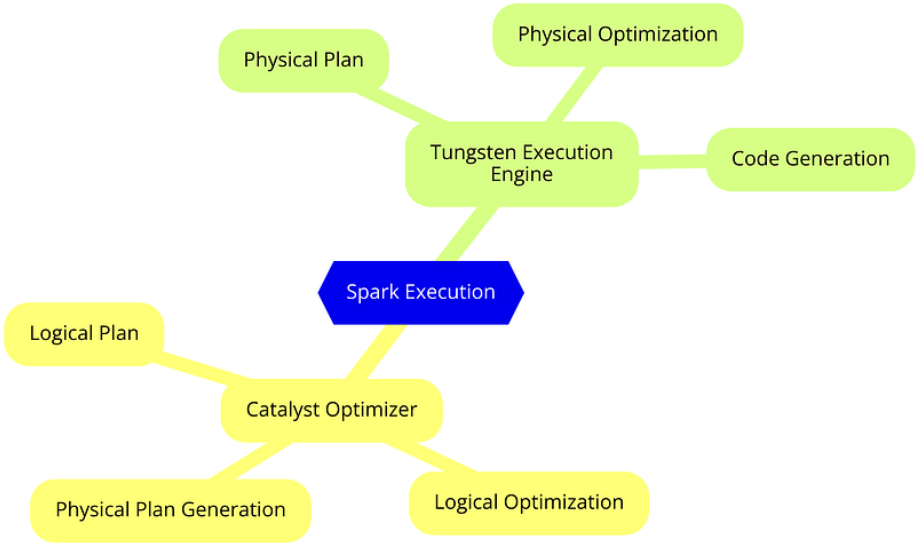
- Handles **memory management** and **binary code generation**.
- Improves **in-memory computation speed** and reduces garbage collection overhead.
- Makes Spark **much faster than traditional MapReduce**, especially for iterative algorithms.

Combined Effect:

- Catalyst + Tungsten enables **highly optimized query execution**, **low-latency transformations**, and **scalable processing** across clusters.

Example Analogy:

- Think of **Catalyst** as the **smart planner** that reorganizes the tasks, and **Tungsten** as the **high-speed executor** that runs them efficiently.



Spark vs Flink vs Dask – Brief Comparison

Feature	Spark	Flink	Dask
API Style	Batch + Streaming	Stream-first	Python-native
Latency	Medium (ms–s)	Low (ms)	Medium
Language	Python/Scala/Java/R	Java/Scala/Python	Python
Ecosystem	MLlib, SQL, GraphX	CEP, streaming apps	NumPy, Pandas, ML

Summary & Future Trends – Unified Analytics

Spark has evolved into a **unified analytics engine** capable of handling **batch processing, streaming, machine learning, and graph analytics** all within one platform.

1.Unified Analytics:
- Spark 4.x integrates **batch, streaming, ML, and graph processing** seamlessly.
- Reduces the need to switch between different engines for different workloads.

2.Delta Lake:
- Provides **ACID transactions** on big data.
- Ensures **reliability and consistency** for large-scale data pipelines.

3.Arrow & Columnar Formats:
- Optimized in-memory formats for faster analytics and **lower serialization costs**.
- Enables **efficient interoperability** with Python (pandas) and other tools.

4.Future Trends:
- Cloud-native Spark deployments.
- Enhanced support for **large-scale ML workflows**.
- Continuous improvements in **performance, scalability, and API simplification**.

End of Chapter 3