307401
Big Data and Data Warehouses

Introduction to Apache Spark

Introduction to Spark

- Apache Spark is a distributed data processing engine designed for large-scale analytics and machine learning.
- It started in 2009 at UC Berkeley's AMPLab as a research project. In 2010, it was made open source, allowing developers worldwide to contribute and adopt
- Unlike Hadoop MapReduce, which writes intermediate results to disk after every operation, Spark performs **in-memory computation**, drastically reducing disk I/O. This makes iterative tasks like machine learning and graph processing much faster.
- Spark follows a **Directed Acyclic Graph (DAG)** execution model instead of a rigid two-stage Map → Reduce pipeline. DAG scheduling allows Spark to optimize task execution, reuse intermediate data (RDD caching), and reduce redundant computations.
- Spark supports multiple high-level APIs (RDDs, DataFrames, Datasets) and integrates with SQL, streaming, MLlib, and GraphX within the same framework.
- Spark can run standalone or on cluster managers like YARN, Mesos, or Kubernetes, and supports diverse storage backends (HDFS, S3, Cassandra, etc.).

Overall, Spark is faster than Hadoop MapReduce because of:

- **In-memory caching** of data.
- **DAG execution model** for optimization.
- **Reduced disk writes** during intermediate stages.
- **Rich APIs** for complex workflows beyond simple map and reduce.

Thus, Spark is widely adopted for ETL, analytics, real-time processing, and AI-driven pipelines.

Spark Core Components

- The Spark Core Engine forms the foundation for all Spark applications. It manages memory, fault recovery, scheduling, and task distribution.
- High-level APIs such as the DataFrame API, RDD API, and SQL API provide access to the core engine, and they are available in multiple languages including Python, Scala, Java, and R.
- On top of the core, specialized libraries like Spark SQL, MLlib, GraphX, and Structured Streaming extend Spark's capabilities, offering advanced functionalities for diverse use cases.

© Databricks 2025. All rights reserved. Apache, Apache Spark, Spark, the Spark Logo, Apache Iceberg, Iceberg, and the Apache Iceberg logo are trademarks of the Apache Software Foundation.

Spark Core Modules

1. Spark Core

RDDs (Resilient Distributed Datasets): The fundamental data structure in Spark, RDDs are immutable, fault-tolerant collections of objects distributed across nodes. They support operations like map, filter, and reduce.

Transformations and Actions: Transformations (e.g., map, filter) create a new RDD, while actions (e.g., count, collect) execute computations and return results.

2. Spark SQL

Spark SQL allows querying structured data using SQL or the DataFrame API. It's optimized for performance and commonly used for analyzing large datasets.

3. MLlib (Machine Learning Library)

MLlib is Spark's library for scalable machine learning algorithms, including regression, classification, clustering, and collaborative filtering.

4. Spark Streaming

Enables real-time data processing for applications needing continuous, live data feeds.

5. GraphX

A graph processing library in Spark for graph-parallel computations, useful for tasks like social network analysis.

Spark Data Objects: 1- The Resilient Distributed Data Object (RDD)

Definition: RDD is the original Spark abstraction introduced in Spark 1.0. An RDD is a distributed collection of objects that can be processed in parallel.

It represents an **immutable, partitioned collection of objects** that can be processed in parallel across a cluster. RDDs were the original API of Spark, providing low-level control over data and transformations.

Characteristics:

- Provides **low-level transformations** (map, filter, flatMap) and actions (collect, count) – gives more control.
- Works with **any type of data** (structured, semi-structured, unstructured).
- Does **not have a schema**, so Spark does not understand the structure of the data.
- No built-in query optimization → all transformations are executed as you write them (Catalyst optimizer not used).

Three key features of RDDs:

1. **Resilient (Fault-tolerant)**: If part of an RDD is lost due to node failure, it can be recomputed using lineage information (a record of transformations applied).
2. **Distributed**: RDDs are split into partitions, and each partition is processed in parallel across Spark executors, ensuring scalability on big datasets.
3. **Immutable**: Once created, an RDD cannot be modified. Transformations produce new RDDs rather than altering existing ones, ensuring consistency and reliability.

Additional features:

- Lazy evaluation: Transformations are only executed when an action is triggered.
- In-memory computation: RDDs can be cached for reuse in iterative algorithms.
- Although DataFrames and Datasets are now preferred for optimization, RDDs remain important for low-level operations and fine-grained control over distributed processing.

Best for: Complex data processing, custom algorithms, ML preprocessing (when DataFrame API is not enough).

Resilient Distributed Data (RDD)

Spark Data Objects: 2- DataFrame

A **DataFrame** in Spark is introduced in Spark 1.3, it is a **distributed collection of data organized into named columns**, much like a table in a relational database or a Pandas DataFrame. It builds on top of RDDs but adds **schema information** and is optimized by the Catalyst query engine.

Main Features:

- **High-level API** → like SQL tables (rows + named columns), supports **SQL queries** and DataFrame API operations.
- Schema-based, meaning Spark knows the data types of each column.
- Backed by **Catalyst Optimizer** → Spark automatically optimizes the query execution plan.
- Integrated with **Tungsten Execution Engine** for faster in-memory computation.
- Transformations are **declarative** (select, filter, groupBy, agg).

Advantages:

Much faster than RDDs for structured data, more user-friendly, and integrates easily with BI tools.

Use Cases: Structured and semi-structured data, ETL pipelines, analytics, machine learning feature engineering.

Disadvantages: Less control compared to RDDs (but usually enough).

Best for: 90% of modern Spark workloads (ETL, analytics, BI, ML preprocessing). RDD is considered **low-level legacy API**, used only when DataFrame/Dataset can't solve the problem.

Spark Data Objects: 3-Dataset

Dataset was introduced in Spark 1.6, Datasets combine the **benefits of RDD (type safety)** and **DataFrame (optimized execution)**.
Characteristics:

- Available in **Scala and Java only**, not in PySpark (Python users rely on DataFrames).
- Uses **Encoders** to serialize objects more efficiently than Java serialization.
- Offers both **functional transformations** (like RDDs) and **declarative queries** (like DataFrames).

Use Cases: Developers who want both compile-time type safety and query optimization.

Comparison

Feature	RDD	DataFrame	Dataset (Scala/Java)
Data Type Awareness	No schema	Schema-aware (columns, types)	Schema-aware + type-safe
Optimization	No (manual only)	Catalyst + Tungsten	Catalyst + Tungsten
Language Support	Java, Scala, Python	Java, Scala, Python, R	Java, Scala (not Python)
Performance	Slower	Faster (optimized)	Faster + type safety
Best Use Case	Unstructured, raw data	Structured/semi-structured data	Type-safe structured data

Spark Local mode vs Cluster Mode

Apache Spark can run in **two broad deployment modes**:

1. Local Mode:

- Spark runs on a single machine using the local JVM.
- Executors are simulated as threads.
- Used mainly for learning, development, debugging, and small-scale data.

Example:

```
spark = SparkSession.builder.master("local[*]").getOrCreate()
```

Here, local[*] means Spark will use all available cores on the local machine.

2. Cluster Mode:

- Spark runs on a cluster with a **cluster manager** (YARN, Mesos, Kubernetes, or Spark Standalone).
- The Driver runs on one node, and Executors run on worker nodes.
- Suitable for production workloads on large datasets.
- Provides scalability, fault tolerance, and resource management.

Key Difference: Local mode is single-machine and good for testing, while cluster mode distributes computation across many nodes for big data processing in production.

Languages supported by Apache Spark

Apache Spark is designed to be **polyglot**, supporting multiple programming languages so developers can use their language of choice. The primary supported languages are:

1. **Scala** – Spark's native language, offering the richest and most efficient API access.
2. **Java** – Well-supported but more verbose compared to Scala.
3. **Python (PySpark)** – The most popular choice for data scientists, leveraging Python's ecosystem.
4. **R (SparkR, sparklyr)** – Provides DataFrame APIs for statisticians familiar with R.
5. **SQL** – Through Spark SQL for querying structured/semi-structured data. Each language API supports Spark's

While Scala and Java provide the most control and efficiency, **PySpark dominates in industry adoption** because of Python's widespread use in analytics, AI, and machine learning.

Spark in Python (PySpark)

PySpark is the **Python API for Apache Spark**, enabling developers to access Spark's distributed computing power using Python code.

It allows Python developers to work with Spark's core components— RDDs, DataFrames, SQL, MLlib, and Structured Streaming—without switching to Scala or Java, Spark's native languages.

PySpark relies on **Py4J**, a bridge library that facilitates communication between the Python runtime and the underlying JVM where Spark actually executes. When a Python command is written in PySpark, it is translated into a JVM call, executed on Spark's cluster, and results are returned back to Python.

The role of Py4J in PySpark

Py4J is a **Python-Java bridge library** that enables seamless interaction between Python applications and Java-based systems. Since Apache Spark is built on Scala (running on the JVM), Py4J is the key enabler that allows Python users to work with Spark via PySpark.

Here's how it works in PySpark:

1. **Python process**: The user writes commands in Python.
2. **Py4J bridge**: PySpark sends these commands to the JVM through Py4J.
3. **JVM (Spark engine)**: The JVM executes the tasks on the Spark cluster.
4. **Results returned**: Py4J sends back the results from JVM to Python.

For example, if you create a DataFrame and run .show(), Py4J translates the Python method call into a JVM operation that executes in Spark's distributed environment.

Without Py4J, PySpark would not be able to use Spark's JVM-based APIs.

Spark Architecture

1. **Driver Program** → Runs main function, creates SparkSession.
2. **Cluster Manager** → Allocates resources (Standalone, YARN, Mesos, Kubernetes).
3. **Executors** → Worker nodes that run tasks.
4. **Tasks** → Smallest execution units, scheduled on executors.
5. **SparkSession** → Entry point for Spark APIs.

- **Driver:** The brain of a Spark application, responsible for planning and coordinating execution.
- **Cluster Manager/Master:** Manages cluster resources and allocates them to the Driver (internal).
- **Workers:** Nodes in the cluster that host Executors.
- **Executors:** Processes on Worker nodes that execute tasks assigned by the Driver.

© Databricks 2025. All rights reserved. Apache, Apache Spark, Spark, the Spark Logo, Apache Iceberg, Iceberg, and the Apache Iceberg logo are trademarks of the Apache Software Foundation.

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

print("App Name:", spark.sparkContext.appName)
print("Master:", spark.sparkContext.master)
```

This prints details about the Spark driver and cluster manager.

Spark Driver

- A **Spark Driver** is the **central coordinator** of a Spark application.
- It is the process that runs the **main program** and is responsible for orchestrating the execution of tasks across the cluster.
- The driver translates user code into tasks, schedules them on executors, and monitors their execution. Without the driver, a Spark job cannot run, as it acts like the **brain** of the application.
- **Key Responsibilities of the Spark Driver**

1. **Creating SparkSession/SparkContext**
   - The driver creates the entry point (SparkSession in PySpark), which initializes the Spark environment.
2. **Building the Logical Plan**
   - When you perform operations (e.g., df.groupBy().count()), the driver builds a **logical execution plan**. This plan is optimized using the **Catalyst Optimizer**.
3. **Generating the DAG (Directed Acyclic Graph)**
   - The optimized plan is broken down into **stages** and **tasks**. The driver converts them into a DAG and submits tasks to executors.
4. **Task Scheduling**
   - The driver communicates with the **cluster manager** (YARN, Mesos, Kubernetes, or Standalone) to allocate resources and launch executors.
5. **Monitoring & Coordination**
   - It tracks task progress, retries failed tasks, and provides metrics via the **Spark UI**.

Spark Driver

Python Example:

- The **driver program** runs this code.
- It builds the DAG for the filter operation.
- Tasks are scheduled on executors to compute the result.
- Finally, the driver collects results and prints them.

Driver vs. Executor (Quick Comparison)

- **Driver** = Brain → Creates DAG, optimizes, schedules tasks.
- **Executor** = Muscles → Executes tasks, stores data, returns results.

Example in PySpark

```python
from pyspark.sql import SparkSession

# Driver program
spark = SparkSession.builder \
    .appName("DriverExample") \
    .getOrCreate()

data = [("Alice", 25), ("Bob", 30)]
df = spark.createDataFrame(data, ["name", "age"])

# Actions and transformations
result = df.filter(df.age > 26)
result.show()
```

Spark Executor

A **Spark Executor** is a **distributed worker process** launched on each node of a Spark cluster.

Executors are responsible for **executing tasks** assigned by the Spark Driver, storing data in memory/disk, and returning results back to the driver. They are the actual computation units in Spark's architecture.

Key Responsibilities of Executors

1. **Task Execution:** Each executor runs multiple **tasks** (units of work) in parallel. Tasks are derived from the **DAG (Directed Acyclic Graph)** built by the driver and scheduled onto executors.
2. **Data Storage (Caching & Shuffling):** Executors hold **intermediate data** in memory (if cached/persisted) and handle **shuffle operations** by exchanging data across nodes during wide transformations like groupBy or join.
3. **Communication with Driver:** Executors report the status of tasks back to the driver and send computed results. If an executor fails, Spark can reschedule the tasks on another executor to ensure **fault tolerance**.

Executor Lifecycle

- Executors are launched when a Spark application starts.
- They remain alive for the duration of the application unless **dynamic allocation** is enabled (then executors can scale up/down based on workload).
- Once the Spark job completes, executors are terminated.

Configuring Executors in PySpark

- spark.executor.instances = 4 → 4 executors launched.
- Each executor has 2 cores and 2 GB RAM.
- Tasks (groupBy operations) will be distributed across these executors.

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("ExecutorExample") \
    .config("spark.executor.instances", "4") \
    .config("spark.executor.memory", "2g") \
    .config("spark.executor.cores", "2") \
    .getOrCreate()

df = spark.read.csv("data.csv", header=True, inferSchema=True)
df.groupBy("category").count().show()
```

Cluster Manager in Spark

A **Cluster Manager** in Spark is the component responsible for resource allocation and management across the nodes in a cluster.

Spark itself does not manage resources directly—it relies on a cluster manager to provide CPU, memory, and scheduling capabilities. Once Spark applications are submitted, the cluster manager ensures that executors are launched with the required resources and monitors their lifecycle.

Key Responsibilities of a Cluster Manager:

1. Resource Allocation: The cluster manager decides how many executors to launch, how much memory and CPU each executor gets, and which machines they will run on.
2. Executor Management: It starts, stops, and monitors Spark executors on worker nodes. Executors live as long as the Spark application runs (unless dynamic allocation changes them).
3. Communication with Driver: The driver requests resources from the cluster manager. Once resources are granted, the driver schedules tasks onto executors.
4. Isolation of Applications: In a multi-tenant environment, the cluster manager ensures fair resource sharing among different Spark applications.
5. Fault Recovery: If an executor fails, the cluster manager provisions a new one, allowing Spark to re-execute failed tasks and maintain fault tolerance.

Cluster managers supported by Spark.

Spark supports multiple cluster managers:

1. **Standalone Cluster Manager** (built-in).
2. **YARN** (Hadoop cluster manager).
3. **Mesos**.
4. **Kubernetes** (modern container-based).

```
print("Cluster Manager:", spark.sparkContext.master)
```

Depending on setup, it shows local[*], yarn, mesos, or k8s.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("ClusterManagerExample") \
    .master("yarn") \   # Or "local", "mesos", "k8s://", "spark://"
    .config("spark.executor.instances", "4") \
    .getOrCreate()
```

Here:

- .master("yarn") tells Spark to request resources from YARN.
- YARN allocates executors, and Spark runs tasks on them.

- **In summary**: The cluster manager is like the **resource allocator** in Spark's ecosystem. While the **driver plans tasks** and **executors execute them**, the **cluster manager provides the infrastructure** (CPU, RAM, containers) to make execution possible. Without a cluster manager, Spark cannot scale beyond a single machine.

SparkSession

A SparkSession is the entry point to programming with Spark using the Dataset and DataFrame API. Introduced in Spark 2.0, it unifies multiple contexts—SQLContext, HiveContext, and SparkContext—under a single API.

It provides:

- Access to Spark's cluster via SparkContext (spark.sparkContext).
- Methods to read data from various sources (CSV, JSON, Parquet, Hive).
- The ability to run SQL queries (spark.sql).

Configuration of Spark settings (.config() builder). Example:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("MyApp") \
    .config("spark.sql.shuffle.partitions", 10) \
    .getOrCreate()
```

- appName names the application.
- .config() sets parameters.
- .getOrCreate() returns an existing session or creates a new one.

Without SparkSession, you cannot create DataFrames or use SQL APIs. It acts as the "gateway" to all Spark functionalities, simplifying application development.

SparkContext

A **SparkContext** is the entry point for using the **RDD API** in Spark. It represents the connection between a Spark application and the Spark cluster, handling resource allocation and task scheduling.

When you create a SparkSession, it automatically creates a SparkContext accessible as spark.sparkContext. Functions of SparkContext:

- Establishes a connection to the **cluster manager** (YARN, Mesos, Kubernetes, or Standalone).
- Requests resources (CPU, memory) for executors.
- Distributes data across the cluster as RDD partitions.
- Schedules tasks for parallel execution. Example:

```
sc = spark.sparkContext
rdd = sc.parallelize([1, 2, 3, 4, 5])
print(rdd.map(lambda x: x * 2).collect())
```

SparkContext creates an RDD from a local list, parallelizes computation, and executes tasks on the cluster.

While SparkSession is preferred for DataFrame APIs, SparkContext remains essential for **low-level RDD operations** and understanding Spark's execution model.

SparkSession vs SparkContext

SparkSession and **SparkContext** are both entry points in Spark, but they serve different purposes:

Feature	SparkSession	SparkContext
API	High-level (DataFrame, Dataset, SQL)	Low-level (RDD)
Introduced	Spark 2.0	Earlier Spark versions
Scope	Unified entry point for SQL, Streaming, ML, Hive	Connects to cluster for RDD-based operations
Access	spark	spark.sparkContext
Ease of Use	Simplifies data loading, queries	Requires functional programming

In summary:

- **SparkContext** is the engine for **RDD-based computation**.
- **SparkSession** is a unified entry point that builds on SparkContext but makes working with structured data much easier.

Thus, SparkSession is preferred for most modern applications.

Tungsten (Execution Engine)

Tungsten is a **back-end execution engine** for Spark.

It optimize **physical execution & memory**

Optimizations in Tungsten:

1. **Memory Management**
   - Off-heap allocation (bypasses JVM GC).
   - Binary format for rows = less serialization overhead.
2. **Cache-friendly execution**
   - Compact data layout → better CPU cache utilization.
3. **Whole-Stage Code Generation (WSCG)**
   - Dynamically generates optimized Java bytecode.
   - Reduces virtual function calls & interpreter overhead.

PySpark Example:

```
df = spark.range(1, 1000000)
df_filtered = df.filter(df.id % 2 == 0)
df_filtered.explain(True)
```

Example: Instead of iterating row by row in JVM, Spark compiles entire query execution into a single tight Java loop.

Tungsten = **Muscle of Spark** (makes execution super fast).

How Tungsten Improves Apache Spark Performance

Tungsten enhances Spark's performance by optimizing CPU and memory usage through several key techniques:

- **Whole-Stage Code Generation:** Tungsten converts whole query stages into optimized bytecode during runtime, cutting down on overhead from function calls and interpretation.
- **Off-Heap Memory Management:** By manually managing memory and using off-heap data structures, Tungsten reduces JVM overhead and minimizes garbage collection pauses.
- **Cache-Aware Computations:** Tungsten organizes data to fit efficiently into CPU caches, reducing memory access latency and boosting throughput.
- **Binary Processing:** It operates directly on binary data, bypassing costly Java/Scala object serialization, which enhances execution speed. These optimizations ensure that Spark applications run faster and more efficiently, particularly for DataFrame and SQL operations.

The Catalyst Optimizer

Catalyst is Spark SQL's **query optimization framework**. It converts SQL/DataFrame queries into optimized execution plans.

Optimization steps:

1. **Analysis** → Resolve column names and types.
2. **Logical Optimization** → Filter pushdown, projection pruning.
3. **Physical Planning** → Choose best execution strategy (e.g., broadcast join).
4. **Code Generation (Tungsten)** → Compiles to JVM bytecode.

PySpark Example:
The explain() command shows how Catalyst transforms the query.

```
df = spark.range(1, 100)
optimized = df.filter("id > 50").select("id")
optimized.explain(True)
```

How Catalyst Optimizes Queries in Apache Spark

- Catalyst is responsible for optimizing the logical query plans in Spark SQL. It employs various techniques to ensure efficient query execution:
  - **Rule-Based Optimization:** Catalyst applies transformation rules such as predicate pushdown and column pruning to simplify logical plans.
  - **Cost-Based Optimization:** Catalyst uses statistics and cost models to select the most efficient execution plan, balancing performance and resource usage.
  - **Logical Optimizations:** Techniques like join reordering and constant folding further refine the logical plans.
  - **Query Planning:** Catalyst converts high-level SQL queries into sequences of optimized physical operations, preparing them for efficient execution by Tungsten.
- These optimizations enable faster and more accurate data analysis, ensuring high performance in Spark applications.

The DAG (Directed Acyclic Graph) in Spark

A DAG represents the sequence of transformations applied to data. Each node is an operation, and edges show dependencies. When an action is triggered, Spark optimizes the DAG into stages and tasks.

- Spark jobs are broken down into stages (groups of tasks that can run in parallel)
- Computations flow in one direction through stages (Directed)
- Stages never loop back, ensuring jobs terminate (Acyclic)
- Stages are organized into a dependency graph for execution flow (Graph)

Importance:

- Fault tolerance via lineage.
- Efficient execution by minimizing shuffles.

PySpark Example:

```
df = spark.range(1, 10)
result = df.filter("id % 2 == 0").groupBy().count()
result.explain()
```

Here, the DAG: range → filter → groupBy → count. Execution starts when count() is triggered.

https://medium.com/@ashutoshkumar2048/dag-in-apache-spark-a3fee17f7494

The DAG (Directed Acyclic Graph) in Spark

The diagram illustrates how a **Spark job** is executed internally:

1. **Stages in Execution**

The rectangles represent **stages**, each made up of multiple tasks.

These stages are created based on shuffle boundaries (when data needs to be moved across nodes).

2. **Directed Flow**

The arrows show that computations move **forward only** from one stage to the next (hence ***Directed***).

There are no backward arrows, which means Spark jobs can't loop infinitely.

3. **Acyclic Nature**

The stages form a chain without cycles (hence ***Acyclic***).

Once a stage finishes, execution moves to the next stage until the job completes.

4. **Dependency Graph**

The curved arrows represent **dependencies** between stages.

For example, the output of Stage 1 becomes input to Stage 2.

In short, the image **visualizes how Spark breaks down a job into stages and tasks, executes them in order, and ensures progress moves forward without looping**.