Spark Framework for Distributed Computing and Big Data Processing

Goian Sergiu-Rares
MSc Student - Distributed Systems in Internet
Babes Bolyai University
Cluj-Napoca, Romania

***Abstract*—The rapid growth of data-intensive applications and AI workloads has created a strong demand for scalable and efficient distributed processing systems. Apache Spark addresses these challenges through an optimized in-memory computation model, resilient fault-tolerance mechanisms, and a unified analytics engine that supports batch processing, SQL querying, streaming and machine learning. This paper presents a concise overview of Spark's architecture and execution model, emphasizing how its design enables high performance in large-scale analytics and distributed environments. Practical examples and use cases illustrate Spark's applicability to modern data platforms, and the paper concludes with an evaluation of its strengths and limitations in comparison to traditional systems such as Hadoop MapReduce.**

I. Introduction

A. *Problem Space, Motivation and Objectives*

Modern workloads such as recommendation systems, forecasting, classification, clustering and anomaly detection require distributed computation models that can deliver low latency, fault tolerance and high bandwidth. Apache Spark addresses these challenges through an in-memory distributed processing engine and a unified ecosystem for batch analytics, SQL querying, streaming, and machine learning.

The objective of this work is to provide a structured overview of the Spark framework within the context of distributed computing and big data processing. Specifically, the paper examines Spark's architecture and execution model, its core components (Spark Core, RDDs, DataFrames, Spark SQL, Structured Streaming, MLlib and GraphX) and its fault-tolerance mechanisms. In addition, practical aspects such as integration with Java, setup procedures and example implementations are discussed too. Finally, the study evaluates Spark's strengths and limitations in comparison with alternative systems such as Hadoop MapReduce and Apache Flink.

II. Overview of Apache Spark

A. *Evolution*

Spark was developed in 2009 as a research project at UC Berkeley AMPLab, primarily to address the limitations of Hadoop MapReduce in handling iterative and interactive workloads. Unlike MapReduce, which writes intermediate data to disk at every computation step, Spark instead introduced in-memory processing, resulting in big improvements for workloads such as ML, graph processing and real-time analytics ([1]).

It has gained fast adoption because of its unified programming model and superior performance. By 2014, it became one of the most active Apache projects, with contributions from large organizations like IBM, Netflix and Intel. Currently, Spark continues to evolve as an open source stable distributed computing engine with a wide environment of modules.

B. *Supported Languages*

Spark has libraries for multiple languages: Java, Python, SQL, Scala or R. A comparative analysis by Borodii et al. [2] investigated the performance of Java, Python and Scala across different ETL scenarios, highlighting clear guidelines for language selection based on workload characteristics:
- Python (PySpark): superior execution time on small and moderately sized datasets (6 seconds for small task, 46 seconds for moderately sized task). These findings suggest PySpark is suited for lightweight Extract-Transform-Load (ETL) tasks and exploratory data analysis due to its high processing efficiency for smaller workloads and ease of implementation;
- Java / Scala (JVM languages): greater stability and performance when processing large datasets and executing complex, multi-step ETL operations involving large joins and partitioned writes;
Spark is implemented in Scala, and this makes lower level Spark programming and code navigation much easier for any Scala developer.

However, the full support for Java is important due to Java's dominance of enterprise distributed systems and ease of integration of the Spark APIs with Java-based microservices and cloud applications, offering all the core functionalities (RDDs, Dataframe and Dataset APIs, Spark SQL, Structured Streaming, integration with Java-based tools like Kafka for stream processing, Spring and Hadoop).

To simplify the integration and connectivity even more, Spark introduced Spark Connect, which decouples the client app from the running Spark cluster, allowing client-side libraries to communicate with Spark through an open-source protocol built on gRPC (protocol for Remote Procedure Calls). This provides flexibility, enabling data scientists and developers to build applications using Spark from anywhere, for instance from a local IDE, a web application or a microservice, without at least needing to run the Spark driver on the client machine.

For practical examples and more details regarding the use of Spark libraries, see chapter V.

III. Architecture

Spark achieves fast big data processing because of important architectural decisions have been made when building it. It utilizes RDDs and the higher level Dataframes and Datasets as its core data abstractions for storing and managing intermediate results in the cluster's RAM, allowing the CPU to have direct access to processed data across multiple transformations, to finally avoid the time consuming IO operations associated with writing and reading from disk at each step.

Besides the in-memory computing, Spark comes with another important feature: a master-worker architecture, built explicitly to handle fault tolerance, which is detailed in a later subchapter (III-G).

The overall execution flow is governed by the interaction between the driver, cluster manager and executors as illustrated in figure 1.

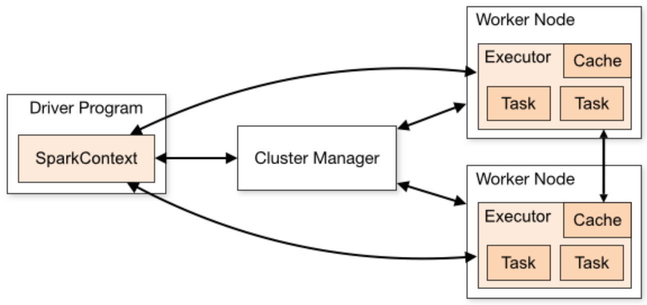

Fig. 1. Key components of the Apache Spark master-worker architecture.

This architecture consists of a Driver, that controls the entire flow of transformation operations, communicates with the user's app and the Cluster Manager that handles the executor nodes / workers that will execute the specific operations on the Dataframes / Datasets.

A. Driver Program (Master Node)

The driver is the main process that runs the Spark application. It hosts the SparkSession (and underlying SparkContext), connects to the cluster manager, and coordinates the entire lifecycle of the application. Its responsibilities include configuration, resource management, job scheduling, metadata tracking, communication with executors and fault recovery.

When the application starts, the driver initializes the SparkSession / SparkContext, resolves configuration parameters, sets up internal RPC endpoints used for communication with executors and the cluster manager. The driver's JVM executes the main() function, which builds a logical plan of operations. This plan is transformed into a Directed Acyclic Graph (DAG) by the DAG Scheduler, which splits the computation into stages based on shuffle boundaries. Each stage is further decomposed into tasks by the Task Scheduler, where each task corresponds to a partition of the input data ([3], [4]).

Once workers are allocated by the cluster manager, the driver distributes the application code (JAR files or Python modules) and distributed tasks across workers. Each task contains the function to be executed, partition metadata, configuration and references to distributed variables (broadcast variables, accumulators). During execution, workers keep reporting status updates, logs and metrics back to the driver through heartbeat messages. The driver aggregates these metrics and detects failures or slow tasks for handling fault tolerance.

The driver must remain reachable by all executors throughout the application's lifetime. For performance and reliability reasons, the driver should be deployed close to the worker nodes, typically within the same LAN, ensuring low-latency scheduling and stable communication. [5]

B. Executors (Worker Nodes)

When the driver submits a task to a cluster, it is divided into smaller units of work called tasks, which are then scheduled to run on the available executors in the cluster, concurrently. At the same time, executors run concurrently and provide the results back to the driver.

An executor is a node that receives a certain amount of memory that it uses to store / process data in memory for faster access during computations. Each executor is able to manage his own memory and can execute at once multiple tasks assigned by the driver (depending on the available slots / CPU cores), so for instance if 4 cores are available, it could process at most 4 tasks in parallel, one task per slot.

These are created when the SparkContext is created and run until the app is terminated. By default, Spark creates one Executor per node in the cluster, but the number of executors can be configured by the user based on application's needs. This number could affect the performance of the Spark application, so it's important to choose the right number based on the available resources and the nature of the data processing tasks. Executors may also communicate to each other, action called shuffle, see a visual representation of it in the figure 2.

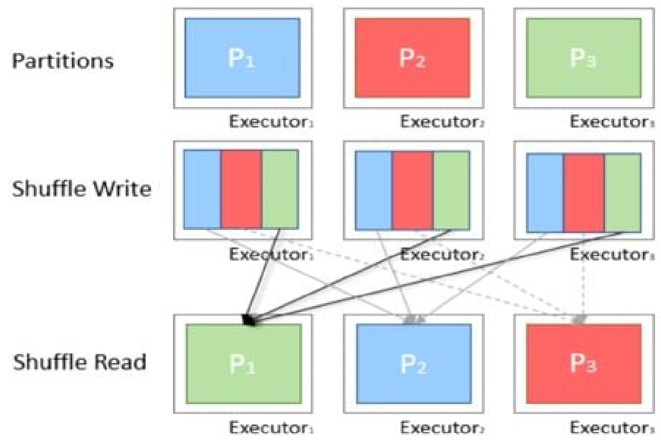

Fig. 2. Visual representation of the shuffling operation

C. Shuffling

A shuffle is when executors exchange data with each other so that the reorganized dataset holds data with the same key on the same executor. Transformations require that all values for a given key end up in the same partition, such as: groupByKey, reduceByKey, join or repartition. A shuffle has 2 main phases, the map phase and the reduce phase. Shuffles are considered an expensive operation, since they include multiple IO operations such as nework IO, disk IO, serialization / deserialization, sorting, creation of thousands of small files.

In the map phase (also called shuffle write), each record is hashed by a partitioner, the hash determines which downstream partition the record must go to. Each executor creates a separate output file per downstream partition, data is serialized into memory buffers, registers shuffle block metadata with the Block Manager (component discussed in III-F).

In the reduce phase (also called shuffle read), each reduce task contacts the Block Manager on every executor that produced shuffled output, requests block corresponding to its partition and finally fetches and aggregates them as required (depending on join, reduceByKey, etc).

1) Executor Types: There are multiple types of executors that can be used based on the requirements of the running app, these are:
- default: used for general-purpose data processing tasks, each node in the cluster runs one default executor by default;
- coarse-grained: for tasks that require more memory, can be configured to have larger amounts of memory than the default; They are also used when the application has large datasets that need to be processed;
- fine-grained: for tasks that require less memory and are used when the application has many small tasks to perform, also useful when data is not distributed evenly across the nodes in the cluster;
- external: when external resources are required for processing, such as GPU processing (an external executor can be used to submit the task for GPU processing);

2) Executor Configuration: Each executor is highly configurable, depending on the application's needs:
- memory: the amount of JVM heap allocated to each executor (commonly 1 to 4 Gb by default, depending on the cluster and distribution);
- cores: the number of CPU cores assigned to a single executor process. By default, Spark assigns *one core per executor*, but this only controls intra-executor parallelism (how many tasks a single executor JVM can run concurrently). The total number of cores available to the application is determined by the cluster manager (YARN, Kubernetes, or Standalone), which governs how many executors are launched and on which nodes;
- number of executors: number of executors that are launched on each node in the Spark cluster, by default 1 executor / node;
- garbage collection: providing 2 different types of algorithms, the default being Concurrent Mark and Sweep (CMS) which minimizes stop-the-world (STW) pauses (JVM halts to perform essential maintenance tasks, such as GC), which provides low latency, predictable use cases being mostly used for workloads with high volume of short-lived objects. The other algorithm is Garbage First (G1) which is designed for processing large heaps, better throughput for long-lived executors that cache large Dataframes / RDDs;
- overhead memory: amount of memory reserved for system processes such as JVM overhead and off-heap buffers, by default 10% of the executor's memory;
- shuffle memory: amount of memory allocated for Spark's shuffle operations, which are used to exchange data between executors;

D. Cluster Manager

The Cluster Manager is the component responsible for allocating and managing the computation resources (CPU, memory, containers, nodes) required by a Spark application. Spark is agnostic to the underlying cluster manager: it can run on Standalone, Hadoop YARN or Kubernetes. Regardless of the implementation, the cluster manager acts as the resource broker between the driver and the physical cluster.

This component has the responsability to decide how many executors the application can obtain, how much memory and CPU each executor will receive, where (on what machines) the executors will run, when to scale up / down the resources. It enforces cluster policies, such as capacity limits, user quotas, FIFO scheduling, node isolation or containerization (ex: Docker in Kubernetes). It provides process / container isolation so that multiple Spark jobs or other applications can run in the same cluster.

Once the driver registers the application, the cluster manager starts executor processes on worker nodes or containers, monitors them (health, memory, CPU, liveness) for fault tollerance III-G, restarts them if allowed (e.g., YARN/K8s) and terminates them when the job ends, dynamic allocation reduces executors, the cluster is under pressure or the node becomes unhealthy.

E. Directed Acyclic Graph (DAG) Model

Spark represents every transformation in the program as a node in a Directed Acyclic Graph, where each node points to the next transformation, there are no cycles, transformations only depend on previous ones, and captures full dataflow dependencies.

The DAG acts as a high level execution blueprint and encodes the complete chain of transformations, enabling the system to: - identify parallelism;
- understand dependencies;
- decide where shuffles are necessary;
- break execution into stages;

By using DAG, all the transformations are recorded and Spark is able to recompute the partitions for fault tolerance. It also allows the underlying optimization tool (Catalyst) to reorder operations, push down predicates, remove unnecessary operations, etc.

This tool has its edges classifies as:
- narrow: one child partition depends on one parent partition (ex: map, filter, etc);
- wide: one child partition depends on multiple parent partitions (ex: groupByKey, join, etc);

1) DAG Scheduler: The DAG Scheduler handles the stage creation, each wide transformation introduces a new stage (topologically sorted stage) and parent stages must finish before children begin. For each stage:
- one task per input partition is created;
- tasks receive partition metadata and serialized function closures;
- failure handling;

DAG Scheduler also retries failed tasks, invalidates shuffle blocks after executor loss, recomputes lost partitions via lineage and rebuild downstream stages as needed.

F. Storage Layer

Spark's storage layer is responsible for managing how data is cached, retrieved, spilled and replicated across the cluster. It provides high-throughput access to intermediate results, minimizes recomputation and supports fault tolerance. The storage subsystem is built around four core components: the Block Manager, the MemoryStore, the DiskStore and the Shuffle Storage subsystem.

1) Block Manager: The Block Manager is the central service responsible for storing and tracking all Spark data blocks. A *block* represents the smallest addressable unit of data in Spark, including:
- RDD partitions;
- DataFrame and Dataset cached partitions;
- shuffle map outputs;
- broadcast variable chunks;
- metadata and spill files;

Each executor hosts its own Block Manager instance, while the driver maintains a Block Manager Master, which stores block-location metadata. Executors communicate with each other via RPC to fetch or push blocks during shuffle or broadcast operations. When an executor fails, the driver invalidates all block references for that node, triggering recomputation through lineage or refetching from replicas.

2) Memory and Disk Storage: Each Block Manager exposes two physical storage tiers:

a) MemoryStore: Cached blocks are stored in the JVM heap or in off-heap memory. The in-heap MemoryStore uses Java objects or serialized byte arrays depending on configuration. Off-heap storage. If memory pressure occurs, the MemoryStore evicts the LRU (least-recently-used) or least-cost blocks, writing them to disk or deleting them if recomputation is cheaper.

b) DiskStore: When blocks cannot fit in memory, they are stored in the executor's local disks (SSD or HDD). The DiskStore writes blocks as serialized byte streams, avoiding the object overhead of deserialized objects in the JVM heap. Disk IO is slower, but provides capacity for large workloads. Spill-to-disk is used both during caching and during shuffle operations to prevent out-of-memory conditions.

3) Serialization Layer: Spark supports two main serializers:
- Java Serialization: slower, high overhead, but universal;
- Kryo Serialization: faster, lower memory footprint, requires class registration for optimal performance;

All cached and shuffle blocks are stored in serialized form unless explicitly configured for deserialized caching (ex: MEMORY_ONLY RDD storage level). Serialized storage:
- reduces heap usage;
- allows contiguous memory layout;
- speeds up shuffle and network transfer;

4) Data Persistence (Caching): Users may persist RDDs, DataFrames or Datasets using storage levels:
- MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY;
- Off-heap levels: MEMORY_ONLY_SER, MEMORY_AND_DISK_SER;
- Replicated levels: MEMORY_ONLY_2, etc;

Persisted blocks accelerate iterative workloads (machine learning, graph processing) and avoid re-executing lineage on each action.

During wide transformations, executors write shuffle map outputs to local disk via the ShuffleWriter. These blocks include partitioned, serialized key-value buckets. Downstream tasks request shuffle blocks through the Block Manager, triggering disk read or memory read depending on spill status.

Shuffle blocks may be:
- kept in memory buffers if space allowsl
- spilled to disk when under memory pressure;
- fetched by multiple downstream tasks (ex: joins, groupByKey);

The storage layer ensures shuffle determinism and recovery: lost shuffle blocks from a failed executor are recomputed by rerunning the corresponding map tasks.

Operators that exceed the shuffle memory threshold (sort, aggregate, join) spill intermediate data to disk automatically. Spark uses a multi-stage spill process:
- sort/aggregate in memory until memory threshold is exceeded;
- spill a sorted run to disk;
- merge multiple spill files at the end;
- emit final shuffle map outputs;

This mechanism prevents out-of-memory errors and enables stable processing of datasets larger than available RAM.

G. Fault Tolerance

The master-worker architecture scales and handles node crashes (hardware failure, network disconnection, OS crash), Java Virtual Machine (JVM) issues (Garbage Collection Pauses due to large heap) or inconsistent data volumes (corrupted input files or uneven data partitions). The current Spark's architecture implements several mechanisms to ensure that distirbuted computations remain correct and progress despite these failures.

1) Heart Monitoring: The workers periodically send heartbeat messages back to the driver, like reporting task metrics or log messages. A message contains the worker id or metrics about tasks running in the executor (run, CPU time, result size). The message is received by the driver to update the last seen time of given worker and tasks metrics, then also check If a worker fails to respond within a timeout window, the master marks it as dead.

2) Lineage in Fault Tolerance: Lineage is a DAG that records the sequence of transformations applied to a dataset, acting as a blueprint: if part of the data is lost, Spark follows the lineage to rebuild it from the original source. When some transformations are applied to an RDD (ex: map), Spark builds a lineage graph instead of computing the result immediately. That is used to track more relevant information: - source data (in-memory collection / file);
- transformations (operations that create new RDDs, ex: mapping or filtering); - dependencies (how each RDD relies on its parents).

To achieve fault tolerance, a lineage is created as transformations are applied (Spark builds a DAG tracking the operations). At failure detection by driver, Spark identifies the lost partition's dependencies using the lineage, then recomputes it from the source data or a checkpoint. Then the recomputed partition is assigned to another worker and the job resumes.

3) Elastic Scaling: This architecture enables scaling out (adding new worker nodes), remove unhealthy nodes or rebalance workloads across workers in case partition skew is detected.

H. Execution Flow

When the developer writes transformations (ex: map, filter, join, select), Spark does not execute them immediately, instead it builds a logical plan representing a deferred computation depending on the underlying Spark data structure:
- for RDDs a lineage graph;
- for Dataframes / Datasets: a Catalyst logical plan;

This phase includes parsing the user program, registering transformations, ensuring schema consistency, creating an initial (unoptimized) representation of the computation. without using any cluster resources or data.

1) Optimization Planning: Spark optimizes the logical plan into a physical plan suitable for distributed execution: Catalyst Optimizer (SQL/DataFrame APIs) performs:
- Predicate pushdown (filter conditions are applied as early as possible in the queries for faster and more efficient queries);
- Projection pruning: query engine identifies and removes the columns that are not required for the final result or any intermediate operation;
- Constant folding: query optimizer pre-calculates and replaces expressions that involve only constant values;
- Join reordering & statistics-based optimization: reorders join operations by precomputing the total cost of operations using statistics;
- Subquery elimination: mering the inner queries into the main query, converting each subquery into a more efficient JOIN operation;

RDD API is optimized by the DAG Scheduler, which creates the physical execution plan (DAG) directly from the explicit user-defined transformations:
- physical execution plan consisting of a DAG of stages;
- operations are classified as narrow (no shuffle) or wide (require shuffle);
- final stage boundaries that determine how tasks will run in parallel;

2) Stage and Task Scheduling: Spark divides the job into stages, each containing many parallel tasks. A stage:
- represents a continuous set of transformations that do not require shuffling;
- ends when a wide transformation occurs (groupByKey, join, repartition, etc);

A task:
- is the smallest schedulable unit;
- processes one partition of data;
- runs inside an executor JVM;

The DAG Scheduler converts the physical plan into stages, then the Task Scheduler assigns individual tasks to executors. Schedulers track executor locality (preferring to run tasks close to data) and failed tasks.

The tasks are triggered inside the executors and they use:
- Block Manager to read data partitions;
- Shuffle Manager for wide transformations;
- Memory Manager for unified memory allocation;

When the driver program finishes, the executors are released, intermediate shuffle files are cleaned, block metadata is discarded and finally the application state is removed from the cluster manager.

IV. Core Components

Spark provides a data processing, streaming, querying SQL, ML and graph analytics under a single system. Its modular design is organized around a powerful execution engine (Spark Core) and a set of libraries that operate on top of a common distributed data abstraction.

A. Spark Core

Spark Core is the fundamental execution engine responsible for scheduling, task dispatch, memory management and fault-tolerant distributed computation by offering a set of functionalities:
- APIs for distributed computation across large clusters;
- job scheduling through the DAG Scheduler and Task Scheduler;
- implements in-memory computation via unified memory management (execution + storage);
- coordinates with the Cluster Manager to acquire executors and distribute tasks;
- exposes fault-tolerant primitives such as lineage-based recomputation and shuffle management.

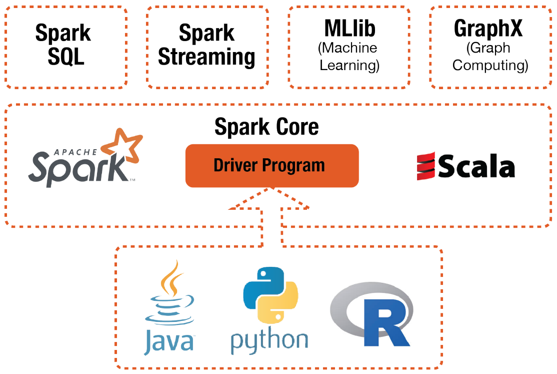

Fig. 3. Spark Core

B. Resilient Distributed Datasets (RDDs)

RDDs are a low-level, fault-tolerant and immutable distributed collections of objects. They are the foundational abstraction used internally by higher-level libraries provided by Spark:
- immutable distributed partitions processed in parallel;
- support narrow and wide transformations with lazy evaluation;
- track lineage graphs enabling deterministic recomputation on failures;
- allow explicit control over partitioning strategies and persistence levels;
- designed for high performance when fine-grained data and compute control is required;

This collection supports 2 types of operations: lazy transformations and immediate actions. The below figure 4 offers visualization for understanding this collection:

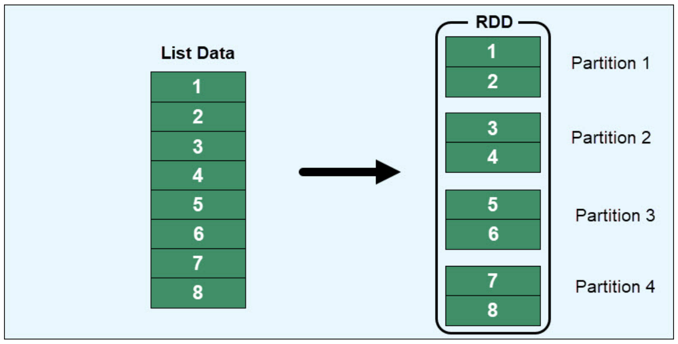

Fig. 4. RDD Visual Structure

C. Dataframes and Datasets

DataFrames and Datasets provide a higher-level structured API built on top of Spark SQL. They use the Catalyst optimizer and an execution engine (called Tungsten engine). They represent distributed tabular data with named columns (schema-aware).

While both APIs share the same high-performance execution engine, they differ primarily in type safety and language support:
- DataFrames (Untyped API): used across all languages (Scala, Java, Python, R, SQL). They treat data as a generic collection of Row objects and offer runtime type checking, so errors related to schema or type mismatches are caught only when the job is executed, similar to running a SQL query;
- Datasets (Typed API): only available in Scala and Java (JVM languages). They are a collection of strongly typed JVM objects, offering compile-time type safety, so the compiler can catch type errors before the application even runs, offering greater robustness for complex applications;

D. Spark SQL

Spark SQL is the module providing relational processing, SQL queries, and a unified optimizer for structured data. It supports ANSI SQL, DataFrame API, Dataset API, and UDFs (User Defined Functions). They also use Catalyst for logical/-physical plan optimization and rule-based transformations and Tungsten to manage off-heap memory, code generation, and binary execution formats.

This library enables interoperability with other data processing formats such as Parquet, ORC, Hive, JDBC and cloud storage.

E. Structured Streaming

Structured Streaming provides a high-level streaming engine using incremental execution over DataFrame / Dataset APIs.

This API treats streaming data as an infinite table with continuous updates and supports micro-batch and continuous processing modes, ensuring also exactly-once semantics using checkpointing and write-ahead logs (WAL) and integration with Kafka, Socket, FileStream, Kinesis, and Delta.

F. MLlib

MLlib is a distributed ML library built on top of DataFrames, with scalable implementations of common ML algorithms (classification, clustering, regression, recommendation). It provides more ML specific features such as:
- DataFrames + Pipelines + Transformers/Estimators API;
- feature extraction, hyperparameter tuning, and model persistence;
- distributed linear algebra through optimized block matrices;

G. GraphX

GraphX enables graph-parallel computation embedded in Spark, by representing graphs using RDD-based abstraction;

It supports operations like joinVertices, aggregateMessages, and subgraph extraction and enables iterative graph algorithms (PageRank, connected components, SSSP);

V. Practical Concepts and Examples

This chapter explores the concrete aspects of working with Spark in real distributed environments. It focuses on how Spark is installed and prepared for execution, how configurations influence performance and resource allocation, how Java applications interact with the Spark runtime and how the core Spark abstractions such as RDDs, DataFrames and SQL queries are used in practice. The chapter concludes with a workflow overview, showing how these elements fit together in a typical distributed data processing pipeline.

A. Setup and Prerequisites

The setup includes installing the necessary software components, configuring the execution environment, preparing the application code and ensuring that the cluster can access all relevant data sources.

Spark requires a compatible JVM (JDK 8 or JDK 11 in most deployments), Apache Spark binaries and optionally Hadoop components when running on HDFS or YARN. On-premise clusters typically require installing Hadoop Common, HDFS, YARN ResourceManager and NodeManagers, while Kubernetes-based deployments only require the Spark distribution and container images.

Prerequisites:
- Software prerequisites: Java JDK (8 or 11 in most deployments), Apache Spark binaries, Hadoop client libraries (if using HDFS) etc;
- Cluster prerequisites: a running cluster manager (Standalone Master, YARN ResourceManager or Kubernetes), worker nodes, properly configured network and SSH access;
- Storage prerequisites: data must already exist in HDFS, S3, local FS, Kafka topics, JDBC databases;
- Configuration prerequisites: Hadoop's core-site.xml, hdfs-site.xml, Spark's spark-defaults.conf, credentials, security settings;
- Hardware prerequisites: CPU cores, RAM amounts, disk space on each node to meet the application's requirements;
- Build prerequisites: Maven or Gradle installed, correct version compatibility (to avoid runtime incompatibilities);

Once the application and environment are prepared, Spark jobs can be executed using the ‘spark-submit‘ utility, which sends the compiled Java application to the cluster along with configuration parameters such as executor memory, the number of executors, and optional JVM settings.

B. Configuration

A Java Spark application typically uses Maven to manage dependencies. The pom.xml file must include the Spark modules required by the application. A project leveraging both the fundamental execution engine and the structured data API requires at least the spark-core and spark-sql dependencies, but in our below example we added spark-stream (for real time data ingestion and processing apps) and spark-mllib (for Spark's ML API) too:

```
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.12</artifactId>
        <version>3.5.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.12</artifactId>
        <version>3.5.0</version>
    </dependency>
    \dependency{
        \groupId{org.apache.spark}
        \artifactId{spark-mllib\_2.12}
        \version{3.5.0}
    \enddependency}
    \dependency{
        \groupId{org.apache.spark}
        \artifactId{spark-streaming\_2.12}
        \version{3.5.0}
    \enddependency}
</dependencies>
```

The next step is to create an executable JAR, that should include all dependencies, usually done using Maven Shade Plusgin, so pom.xml should also contain the following piece of xml tags that will finally produce a complex JAR, that Spark can run on any cluster node:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.2.4</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <createDependencyReducedPom>false</
                createDependencyReducedPom>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

After building: ”mvn clean package”, the resulting JAR file (inside target folder) can be sumitted to any Spark cluster

Since Spark relies heavily on external data sources, these must be placed in locations accessible by all cluster nodes:
- HDFS: Common choice for on-prem distributed clusters. Data is uploaded using hdfs dfs -put file data/;
- Cloud storage (S3, Azure Blob, GCS): Requires credentials configured in Spark and Hadoop's core-site.xml;
- Local filesystem: Only suitable for local mode, not for distributed clusters;
- Kafka streams: Requires topic configuration and network access;
- Relational databases: JDBC drivers must be provided inside the JAR or on the classpath;

To ensure consistent access, all nodes in the cluster must reference the same paths, for example reading a CSV in Java as a Dataset:

```
Dataset<Row> df = spark.read()
    .option("header", "true")
    .csv("hdfs:///datasets/users.csv");
```

To ensure that a Spark cluster operates correctly, the worker nodes must be able to resolve to the master node's hostname, required ports must be opened (7077 - standalone master, 8088 - YARN), all nodes need access to the same storage layer, the same spark version should be installed across nodes to avoid incompatibilities.

The 'spark-submit' command launches the driver program on the target cluster, deploying the app from the client machine to an edge node or a gateway VM, then specifies parameters such as the master URL (local, standalone, YARN, or Kubernetes), the JAR to execute, the number of executors, executor memory, driver memory and any additional JVM flags. At this stage, the environment is fully prepared to run distributed Spark applications in a reliable and scalable manner.

```
spark-submit \
  --class com.example.MySparkApp \
  --master yarn \
  --deploy-mode client \
  --executor-memory 2g \
  --num-executors 4 \
  --executor-cores 2 \
  --conf spark.executor.extraJavaOptions
      "-Dsome.property=value" \
  --conf spark.dynamicAllocation.enabled=true \
  my-spark-app.jar
```

C. Integration With Java

While Spark is implemented in Scala, the Java API offers full access to all major features, including RDDs, DataFrames, Datasets and the SQL engine (full API details are available in the official Spark documentation [5]). However, the Java interface relies heavily on explicit typing, encoders and JavaBeans.

1) SparkSession: The entry point for any Spark application is the SparkSession, as discussed in the previous chapter. After the environment has been configured, the Java program initializes this session, which internally creates the underlying SparkContext and SQL components:

```
SparkSession spark = SparkSession.builder()
    .appName("My Java Spark App")
    .master("yarn")
    .getOrCreate();
```

From this point, the Java application can interact with distributed datasets in several ways.

2) RDDs: For low-level operations, Java developers typically use JavaRDD objects, which provide functional-style transformations and actions.

RDDs are immutable collections of data distirbuted across a cluster. They enable parallel execution on multiple nodes, fault tolerance, lazy evaluation and efficient caching. You can create an RDD from a file using the sparkContext() provided by the Spark session, as below:

```
JavaRDD<String> lines = spark.sparkContext()
    .textFile("hdfs:///datasets/logs.txt", 1)
    .toJavaRDD();

// filter lines containing "ERROR"
JavaRDD<String> errorLines = lines.filter(line ->
    line.contains("ERROR"));

// count the number of error lines
long count = errorLines.count();
```

Transformations such as filter are lazily evaluated, meaning Spark builds a computation DAG rather than executing immediately. Actions like count() trigger the job execution, aggregating results across partitions. Using JavaRDD ensures type safety and compatibility with Java functional interfaces.

RDD transformations are lazily evaluated, allowing Spark to build an optimized DAG of operations. Some transformations provided by JavaRDD:
- map(Function) – applies a function to each element and returns a new RDD;
- flatMap(Function) – similar to map, but each input element can return multiple output elements;
- distinct() – removes duplicate elements;
- union(RDD) – merges two RDDs;
- groupBy(Function) – groups elements based on a key function;

RDD actions trigger computation and produce results. Some actions (trigger computation) provided by JavaRDD:
- collect() – retrieves the entire RDD to the driver (careful for large datasets);
- take(n) – returns the first n elements;
- reduce(Function) – aggregates elements using a binary function;
- saveAsTextFile(path) – writes RDD contents to distributed storage;

3) Dataframe / Dataset Operations: For structured workloads, the preferred approach is to use Dataset¡Row¿ (DataFrames) or strongly typed Dataset¡T¿ objects combined with Java encoders.

```
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoders;

// read CSV into DataFrame
Dataset<Row> df = spark.read()
    .option("header", "true")
    .csv("hdfs:///datasets/users.csv");

// show first 5 rows
df.show(5);

// select columns and filter
Dataset<Row> adults = df.select("name", "age")
                       .filter(df.col("age").gt(18));

// dataset with type-safe JavaBeans
public class User implements Serializable {
    private String name;
    private int age;
    // getters / setters
}

Encoder<User> userEncoder =
    Encoders.bean(User.class); // serialize users
Dataset<User> users = df.as(userEncoder); //
    serialized dataset

// apply transformation / filter
Dataset<User> adultUsers = users.filter(user ->
    user.getAge() > 18);
adultUsers.show();
```

JavaBeans allows Spark to infer schemas and apply type-safe transformations. When mapping structured data into typed datasets, Java developers must define encoder instances explicitly, as seen above in the user object's serialization example. Due to the distributed nature of Spark, classes used inside transformations must be serializable. In Java, this often requires implementing "java.io.Serializable" or enabling more efficient serialization mechanisms such as Kryo through the Spark configuration.

Some frequently used operations applied on Dataset / Dataframe are:
- select(String, ...) – choose specific columns;
- filter(Column) – filter rows based on conditions;
- groupBy(String, ...) – group data for aggregation;
- agg(Map¡String, String¿) – apply aggregate functions (sum, avg, max);
- join(Dataset, Column) – join two Datasets/DataFrames on a key;
- withColumn(String, Column) – create or modify a column;
- drop(String, ...) – remove column / columns;
- orderBy(String, ...) – sort rows;

4) SQL: Spark SQL enables running familiar SQL queries on distributed datasets. Queries can perform operations such as filter, aggregate, join, sort.

```
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Dataset;

// create temporary view from DataFrame
df.createOrReplaceTempView("users");

// execute query
Dataset<Row> sqlResult = spark
    .sql("SELECT name, age FROM users WHERE age >
        18");

// show query result
sqlResult.show();
```

Temporary views created from DataFrames or Datasets allow developers to write declarative queries while Spark handles distributed execution behind the scenes. Some operations used include:
- filtering: SELECT * FROM table WHERE column ¿ 10;
- aggregation: SELECT key, COUNT(*) FROM table GROUP BY key;
- joins: SELECT a.*, b.* FROM tableA a JOIN tableB b ON a.id = b.id;
- ordering: SELECT * FROM table ORDER BY column DESC;

Some advanced SQL features may differ or have limitations, for instance window functions (ROW_NUMBER(), RANK(), LEAD(), LAG()) are supported, while some vendor specific functions (like Oracle or PostgreSQL extensions) are not available by default. However, Spark has many builtin functions (date_format, concat_ws, substring, round etc), to replace the unsupported vendor specific functions.

D. Workflow

The below code snippet illustrates a typical hybrid Spark workflow that uses both the low-level RDD API and the higher-level DataFrame/Spark SQL API to process log data and generate a summary. This is a five-step process that executes on the cluster:

```
JavaRDD<String> rawLogs = spark.sparkContext()
    .textFile("hdfs:///datasets/logs.txt", 1)
    .toJavaRDD();
JavaRDD<String> errorLogs = rawLogs.filter(line ->
    line.contains("ERROR"));

// convert RDD to Dataset<Row> (or Dataframe)
Dataset<Row> errorDF =
    spark.createDataset(errorLogs.rdd(),
    Encoders.STRING())
                            .toDF("logLine");

// apply transformations and SQL
errorDF.createOrReplaceTempView("error_logs");
Dataset<Row> result = spark.sql("SELECT COUNT(*)
    AS error_count FROM error_logs");

// write result to HDFS
result.write()
      .mode("overwrite")
      .csv("hdfs:///datasets/error_summary");

// print result locally
result.show();
```

The process begins by reading a text file from the distributed filesystem (HDFS) and creating a RDD (data ingestion). The filter transformation creates a new RDD, errorLogs, which contains only the log lines that include the substring ”ERROR” (Spark does not execute this yet, it only adds the filter operation to the DAG). Then the workflow transitions from the untyped RDD API to the highly optimized, structured DataFrame API. The errorLogs RDD is converted into a Dataset¡Row¿ (a DataFrame) and from this point, the data is managed in Spark’s optimized, columnar (Tungsten) format, enabling Catalyst optimization.

High-level transformations and actions are performed using the familiar SQL interface, the DataFrame is registered as a temporary, in-memory table named error_logs.

The spark.sql method executes an SQL query that calculates the total number of rows (number of error logs) from the temporary view. This is where the Catalyst Optimizer analyzes and optimizes the physical execution plan (combining the filtering and counting operations).

The final step triggers the actual execution of the entire DAG and writes the results to storage and the console. These are Actions that force the entire Spark job to execute: write() saves the 1-row result DataFrame as a CSV file in HDFS, overwriting the destination if it exists, while show() prints the result to driver console.

VI. USE CASES

There are multiple use cases for Apache Spark across various industries, driven by its unified engine for batch, streaming and advanced analytics. It is mostly used for data engineering, data science and ML workloads.

A. Log Processing

This process involves ingesting, cleaning, parsing and analyzing the massive volumes of unstructured log data generated by servers and applications. Due to the massive volumes of log data,

Spark fits this workload using its distributed capabilities, of processing data concurrently. Spark SQL and DataFrames allow developers to quickly impose structure (schema) onto logs, at a very large scale. The Spark Session will be able to provide an optimized way to rapidly calculate metrics (ex: error rates, unique users) from Tb of daily logs.

B. ETL Pipelines

This is a process specific to data engineering tasks, extracting data from one or more sources, applying business logic and transformations (cleaning, aggregating, changing format of fields, etc) and loading the result to a destination (data warehouse or data lake).

Spark is the industry standard for large-scale ETL workflows due to its in-memory processing and the performance of Catalyst. It efficiently handles complex multi-stage pipelines involving large joins and data shuffling.

C. Distributed ML

Training large-scale ML models on datasets too large to fit into the memory of a single server needs high-performance, distributed algorithms.

Spark's core strength lies in handling iterative workloads rapidly, as model training often requires the data to be scanned repeatedly, which benefits massively from the in-memory caching layer.

D. Streaming Pipelines (Kafka + Spark)

Continuously ingesting real time data streams from sources like IoT devices or media platforms, often using Kafka as a message buffer, leas to large chunks of dat aot be processed concurrently.

Spark Structured Streaming provides an unified API that treats a continuous stream of data as an infinitely growing table. This allows developers to apply the same efficient SQL and DataFrame logic used for batch data, with low-latency and fault-tolerant processing.

E. Graph Processing

Newtrok data needs supervisation, so analyzing relationships and connections within massive, interconnected network data is important. Network data is often modeled as nodes and edges (connections). Common examples include social networks or road maps.

The specialized GraphX component uses Spark's core engine to run graph algorithms across the distributed cluster.

VII. Advantages, Limitations, Comparison With Other Distributed Processing Frameworks

A. Advantages

Most of the Spark's advantages stem from its architectural design and unified programming model:
- In-Memory Processing: drastically reducing I/O latency;
- DAG Scheduler: allows Spark to optimize the execution plan, combine multiple operations into a single stage and execute the necessary sequence of tasks without unnecessary disk operations;
- Unified API: a single set of APIs (Core, SQL, Streaming, MLlib, GraphX), simplifying development, deployment and maintenance compared to using separate systems for each task;

B. Limitations

- High Memory Demand: Spark demands a large amount of memory per core to process massive volume datasets effectively. The memory pressure coming from big data processing leads to high GC overhead (more frequent STW pauses), which introduces unpredictable latency;
- Setup Complexity: configuring Spark for large, production clusters requires expertise in tuning parameters for the specific workload, cluster manager (YARN, Kubernetes) and JVM settings;
- Cost: due to its reliance on RAM, running it can require more expensive, heavy memory resources compared to disk-based systems like Hadoop;

C. Comparison

1) Hadoop MapReduce: Back when it was developed, the fundamental goal behind developing the framework was to overcome the inefficiencies of MapReduce. Even though MapReduce was a huge success and had wide acceptance, it could not be applied to a wide range of problems. MapReduce is not efficient for multi-pass applications that require low-latency data sharing across multiple parallel operations. Since for some data analytics applications, data has to be read from disk storage sources and then written back to the disk as distinct jobs, MapReduce couldn't fit for tasks such as iterative algorithms, used in ML and graph processing, interactive business intelligence and data mining, where data from different sources are loaded in memory and queried repeatedly and streaming applications that keep updating the existing data and need to maintain the current state based on the latest data ([1]).

2) Apache Flink: Apache Flink is another leading distributed processing engine, but more specialized on stream processing. While Spark Structured Streaming handles continuous data streams using a series of micro-batches, Flink uses an architecture built for stream processing. This makes Flink often the preferred choice for applications requiring very low latency and handling stateful stream processing with high consistency guarantees. Spark, in contrast, offers better generality across batch and stream processing with a single programming model.

3) Comparison Table: To summarize the architectural differences and strengths of the leading distributed processing frameworks, The below table provides a comparison based on key operational features [6].

TABLE I
Comparison Summary of the Distributed Processing Frameworks Discussed Above

Feature	**Apache Spark**	**Hadoop MapReduce**	**Apache Flink**
Primary Model	Unified (Batch + Stream)	Batch	Stream-First
Data Storage	In-Memory (RAM)	Disk (HDFS)	In-Memory / State Backend
Latency	Low (Seconds)	High (Minutes)	Very Low (Milliseconds)
Execution	DAG Scheduler	Two-Stage Map/Reduce	Continuous Streams
Performance	Excellent for Iteration	Poor for Iteration	Excellent for Stateful Streams

VIII. Conclusion

Apache Spark is a highly optimized distributed data processing engine that unifies batch, streaming, ML and graph computations under a single model. The master-worker architecture containing the driver program, cluster manager and executor nodes enable efficient task scheduling, dynamic resource allocation and fault tolerant execution.

The main abstraction of RDDs, DataFrames and Datasets, allows for efficiently processing large volumes of data in parallel while maintaining strong type safety. The DAG model and in-memory computation reduce disk IO overhead and enables fast iterative algorithms, which are essential for current ML and graph computation workloads.

The fault tolerance mechanisms including lineage recomputation, heartbeat monitoring and shuffle resilience ensure reliable processing in the presence of node crashes, JVM failures or data skew. Additionally, its integration with various cluster managers (YARN, Kubernetes, Standalone) and support for multiple storage backends (HDFS, S3, JDBC, Kafka) make it highly adaptable to different environments.

To conclude, Spark's architecture and execution model provide a scalable, high-performance platform for low-latency processing in distributed systems. With robust handling of massive data, flexible libraries, Spark is a superior choice for large-scale data processing from a technical point of view.

References

[1] The Apache Software Foundation. (2024) Evolution of apache spark: Apache spark history. Apache Spark. Accessed: November 23, 2025. [Online]. Available: https://spark.apache.org/history/

[2] I. Borodii, I. Fedorovych, H. Osukhivska, D. Velychko, and R. Butsii, "Comparative analysis of large data processing in apache spark using java, python and scala," in *3rd International Workshop on Computer Information Technologies in Industry 4.0 (CITI)*, ser. CEUR Workshop Proceedings, O. P. Zherlitsyn *et al.*, Eds., vol. 4057, Ternopil, Ukraine, June 2025, pp. 1–10, arXiv:2510.19012.

[3] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[4] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Data Analytics*. O'Reilly Media, 2017. [Online]. Available: https://cs.famaf.unc.edu.ar/ damian/tmp/bib/Learning$_Spark_Lightning-Fast_Big_Data_Analysis.pdf$

[5] The Apache Software Foundation. (2025) Apache spark documentation (latest). Apache Spark. Accessed: November 23, 2025. [Online]. Available: https://spark.apache.org/docs/latest

[6] R. M. A. El-Ghafar, A. El-Bastawissy, E. Nasr, and M. H. Gheith, "Comparison between hadoop mapreduce, apache spark, and apache flink," *IOP Conference Series: Materials Science and Engineering*, vol. 976, no. 1, p. 012015, 2020.