Comprehensive Lecture Notes: Apache Spark

Based on Lecture by Dr. Tariq Mahmood
Fall 2025

November 17, 2025

Contents

1 Introduction: The Need for Apache Spark

1.1 Limitations of Hadoop MapReduce

While Hadoop MapReduce revolutionized big data processing by enabling distributed computing on commodity hardware, it faces significant limitations for certain classes of applications:

- **Disk I/O Bottlenecks:** MapReduce is inherently disk-based. Between the Map and Reduce phases, intermediate data is written to the hard disk to ensure fault tolerance. For iterative algorithms (like K-Means clustering or Logistic Regression) that pass over the same data multiple times, this constant reading and writing to disk creates massive latency.
- **Lack of Efficient Data Sharing:** In MapReduce, data sharing between distinct jobs requires writing to a distributed file system (like HDFS). This adds serialization and replication overhead.
- **Latency:** It is optimized for batch processing and is generally unsuitable for interactive queries or real-time stream processing.

1.2 The Spark Solution

Apache Spark was developed at the UC Berkeley AMPLab (2009) to address these inefficiencies. It is a unified analytics engine for large-scale data processing.

- **In-Memory Computing:** Spark's primary advantage is its ability to process data in-memory (RAM). By avoiding intermediate disk I/O, Spark can be up to **100x faster** than MapReduce for certain applications.
- **Versatility:** Unlike MapReduce, which is strictly for batch processing, Spark supports streaming, SQL queries, machine learning, and graph processing within a single engine.
- **Sort Competition Record (2014):** Spark famously sorted 100 TB of data in 23 minutes using 206 nodes, whereas Hadoop MapReduce required 72 minutes using 2100 nodes. This demonstrated Spark's efficiency (3x faster with 1/10th the resources).

2 Spark Architecture

Spark employs a master-slave architecture that decouples the compute engine from the resource manager.

2.1 Core Components

1. **Driver Program:** The "brain" of the application.
   - Runs the `main()` function and creates the `SparkContext` (or `SparkSession`).
   - Converts the user's code into a Directed Acyclic Graph (DAG) of execution.
   - Schedules tasks and coordinates with the Cluster Manager.
2. **Cluster Manager:** The resource allocator.
   - Responsible for allocating resources (CPU, RAM) across the cluster.
   - Types: Standalone (built-in), Apache YARN (Hadoop), Apache Mesos, and Kubernetes.

3. **Worker Nodes:** The machines where computation happens.
4. **Executors:** JVM processes running on Worker Nodes.
   - They execute the tasks assigned by the Driver.
   - They store computation results in memory (caching).
   - If an executor fails, the tasks are reassigned to others.
5. **Tasks:** The smallest unit of work. A stage is divided into tasks, where each task processes one partition of data.

2.2 Execution Hierarchy

The execution flow follows a strict hierarchy:

$$\text{Application} \rightarrow \text{Jobs} \rightarrow \text{Stages} \rightarrow \text{Tasks}$$

- **Job:** Triggered by an *Action* (e.g., `count()`, `collect()`).
- **Stage:** Divided based on shuffle boundaries. Operations that do not require moving data (like `map`) are pipelined into a single stage. Operations requiring data movement (like `reduceByKey`) start new stages.
- **Task:** A unit of work sent to one executor.

3 Resilient Distributed Datasets (RDDs)

3.1 Definition

The RDD is the primary data abstraction in Spark (introduced in Spark 1.0). It represents a collection of elements that is:

- **Resilient:** Fault-tolerant. If a node fails, the data can be reconstructed.
- **Distributed:** Data is partitioned across multiple nodes in the cluster.
- **Dataset:** A collection of partitioned data.

3.2 Key Characteristics

- **Immutability:** Once created, an RDD cannot be modified. To change data, a new RDD must be created via transformation. This simplifies consistency in parallel programming.
- **Lazy Evaluation:** Transformations are not executed immediately. Spark records the "recipe" (DAG) and executes it only when an *Action* triggers it. This allows for optimization (e.g., pipelining operations).
- **Partitioning:** A partition is the atomic unit of parallelism.

  $$\text{Number of Tasks} = \text{Number of Partitions}$$

  If an RDD has 10 partitions, Spark will spawn 10 tasks to process it in parallel.

3.3 Operations: Transformations vs. Actions

Spark operations are categorized into two types:

3.3.1 1. Transformations (Lazy)

Create a new RDD from an existing one. They build the lineage graph.

- **Narrow Transformations:** No data shuffling required. Data is processed within the same partition.
  - Examples: `map()`, `filter()`, `flatMap()`.
- **Wide Transformations:** Requires data shuffling across the network. Data from all partitions may be needed to compute the result.
  - Examples: `reduceByKey()`, `groupByKey()`, `join()`, `distinct()`.

3.3.2 2. Actions (Eager)

Trigger the actual computation and return a result to the Driver or write to storage.

- Examples: `count()`, `collect()`, `take(n)`, `saveAsTextFile()`.
- *Warning:* `collect()` brings all data to the Driver's memory. On large datasets, this causes OutOfMemory (OOM) errors.

4 Optimization Concepts

4.1 The DAG (Directed Acyclic Graph)

The DAG represents the logical execution plan.

- **Lineage:** RDDs do not store data physically on disk (unless cached/checkpointed). Instead, they store *lineage*—the set of steps required to compute the RDD from the source.
- **Fault Tolerance:** If a partition is lost due to node failure, Spark looks at the lineage graph and re-computes *only* the missing partition, rather than replicating data (like Hadoop) or restarting the whole job.

4.2 Shuffling

Shuffling is the process of redistributing data across partitions (and physical nodes) so that data with the same key is grouped together.

- It is expensive because it involves Disk I/O, Network I/O, and Serialization.
- Triggered by operations like `repartition`, `join`, and `ByKey` operations.

4.3 Optimization: ReduceByKey vs. GroupByKey

This is a critical optimization pattern in Spark.

- **groupByKey():** Shuffles *all* data across the network to group values. This causes high network traffic and potential OOM errors if a single key has many values.
- **reduceByKey():** Performs a **local aggregation** (map-side combine) on each mapper before shuffling. This significantly reduces the amount of data sent over the network.
- *Rule:* Always prefer `reduceByKey` over `groupByKey` for aggregations.

4.4 Persistence and Caching

By default, RDDs are recomputed every time an action runs on them. For iterative algorithms (like Machine Learning), this is inefficient.

- **cache():** Stores the RDD in memory (RAM). Short for `persist(MEMORY_ONLY)`.
- **persist(level):** Allows specifying storage levels:
  - `MEMORY_ONLY`: Fast, recomputes if RAM fills up.
  - `MEMORY_AND_DISK`: Spills to disk if RAM fills up (slower but safer).
  - `DISK_ONLY`: Useful for massive datasets.

5 Spark Ecosystem and APIs

5.1 Evolution of APIs

1. **RDD (Spark 1.0):** Low-level, functional programming style. No schema awareness. Optimization is limited to the user's coding skill.
2. **DataFrames (Spark 1.3):** Organized into named columns (like a relational database table).
   - *Catalyst Optimizer:* Spark can optimize queries (e.g., filter pushdown) because it understands the structure of the data.
   - Faster than RDDs for Python/R due to optimizations.
3. **Datasets (Spark 1.6):** Provides type safety (compile-time checks) and object-oriented programming interface. Available in Scala and Java (not Python/R).

5.2 Spark with Hadoop

Spark is strictly a compute engine; it does not have its own storage system. It integrates heavily with the Hadoop ecosystem:

- **Storage:** Reads from HDFS, S3, HBase.
- **Resource Management:** Runs on YARN.
- **Metadata:** Integrates with Hive Metastore to read table schemas.

Deployment Modes on YARN:

- **Client Mode:** The Driver runs on the client machine (e.g., your laptop or a gateway node). Good for interactive debugging but bad for production (network latency).
- **Cluster Mode:** The Driver runs inside a container (ApplicationMaster) on the cluster. Best for production jobs.

6 Real-World Use Cases

- **Uber:** Uses Kafka, Spark Streaming, and HDFS to build continuous ETL pipelines. Converts unstructured event data into structured data for analytics and operations (e.g., surge pricing calculations).

- **Netflix:** Uses Spark for recommendation engines. It analyzes user viewing habits to personalize movie suggestions and inform content creation strategies.
- **Pinterest:** Uses Spark for ETL and streaming analytics to understand user engagement with "Pins" in real-time.
- **Capital One:** Uses Spark for fraud detection. Analyzing transaction patterns to identify probability of fraud.

7 Code Examples

7.1 Word Count (Python)

```
# Initialize RDD from text file
text_file = sc.textFile("hdfs://path/to/book.txt")

# Transformation Pipeline
counts = text_file.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)

# Action: Save to disk
counts.saveAsTextFile("hdfs://path/to/output")
```

7.2 Estimating Pi (Python)

```
import random

def sample(p):
    x, y = random.random(), random.random()
    # Check if point is inside unit circle
    return 1 if x*x + y*y < 1 else 0

NUM_SAMPLES = 1000000
# Parallelize creation of list, map sample function, reduce sum
count = spark.sparkContext.parallelize(range(0, NUM_SAMPLES)) \
             .map(sample) \
             .reduce(lambda a, b: a + b)

print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))
```