Apache Spark Fundamentals

Architecture, DataFrames, Lazy Evaluation & Practical Tasks

Yash Kavaiya

14-Day AI Challenge - Day 2

Agenda

- **Introduction to Apache Spark**
- **Spark Architecture Deep Dive**
  - Driver, Executors, Cluster Manager
  - DAG (Directed Acyclic Graph)
- **DataFrames vs RDDs**
  - Performance Comparison
  - When to Use What
- **Lazy Evaluation**
  - Transformations vs Actions
  - Benefits & Optimization
- **Notebook Magic Commands**
- **Practical E-Commerce Analysis**
  - Select, Filter, GroupBy, OrderBy

What is Apache Spark?

Definition

An **open-source, distributed computing system** designed for fast, general-purpose cluster computing. Developed at UC Berkeley's AMPLab in 2009.

MapReduce Problems:

- Disk I/O bottleneck
- Not suitable for iterative algorithms
- Complex programming model

Spark Solutions:

- **In-memory computing** (100x faster)
- Unified engine for all workloads
- Rich APIs (Python, Scala, Java, R)

Spark Ecosystem Components

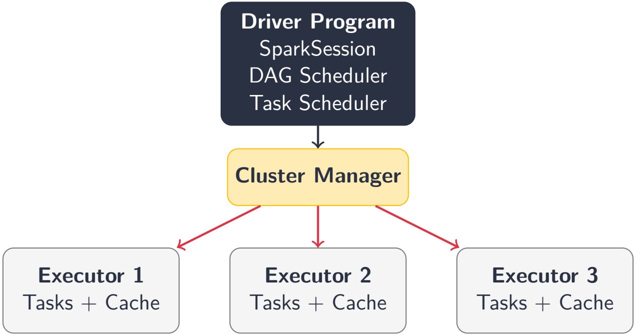
Spark SQL	**Streaming**	**MLlib**	**GraphX**
Structured Data	Real-time	Machine Learn	Graph Processing

Spark Core - Task Scheduling, Memory Management, Fault Recovery

Key Insight

Spark provides a **unified platform** for batch processing, streaming, SQL, ML, and graph analytics.

Spark Architecture Overview



Architecture Pattern

Master-Slave model: Driver (brain) $\rightarrow$ Cluster Manager (resources) $\rightarrow$ Executors (workers)

Driver Program - The Brain

Key Responsibilities:

- Creates **SparkSession/SparkContext**
- Converts user code to tasks
- Schedules tasks on executors
- Maintains metadata about RDDs

Memory Considerations:

- Holds application metadata
- Collects results with `collect()`
- Broadcasts variables to executors

Warning

`collect()` on large data $\Rightarrow$ **OutOfMemoryError!**

Executors - The Workers

Definition

Worker processes that execute tasks and store data. Each executor runs on a worker node in the cluster.

Characteristics:

- Launched at app start, run for lifetime
- Execute tasks, return results
- Store cached RDDs/DataFrames
- Multiple cores for parallel tasks

Memory Division:

- **Storage Memory:** Caching
- **Execution Memory:** Shuffles, joins
- **User Memory:** Data structures
- **Reserved:** 300MB fixed

Cluster Managers

Purpose

External service that manages resources across the cluster and allocates resources to applications.

Type	Description	Use Case
Standalone	Spark's built-in manager	Development, learning
YARN	Hadoop's resource manager	Production Hadoop
Mesos	Apache Mesos	Multi-framework
Kubernetes	Container orchestration	Cloud-native
Local	Single JVM	Testing

DAG - Directed Acyclic Graph

Spark's Secret Weapon for Optimization

When you write Spark code, it builds a DAG of operations before executing.

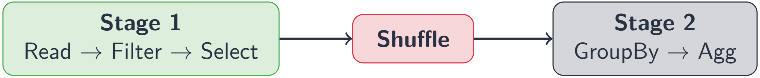
DAG Properties:

- **Directed:** Operations flow input → output
- **Acyclic:** No circular dependencies
- **Graph:** Visual computation representation

Execution Flow:

- Build Logical Plan
- Optimize the Plan
- Create Physical Plan
- Divide into Stages
- Create Tasks

Stages and Shuffles



Narrow Transformations:

- One input → one output partition
- `map`, `filter`, `select`
- **No shuffle needed**

Wide Transformations:

- Input → multiple output partitions
- `groupBy`, `reduceByKey`, `join`
- **Requires shuffle (expensive!)**

What is RDD?

Resilient Distributed Dataset

The fundamental data structure of Spark - an immutable, distributed collection of objects.

Key Properties:

- **Resilient:** Fault-tolerant via lineage
- **Distributed:** Data across nodes
- **Dataset:** Partitioned collection

Creating RDDs:

- From collection: `parallelize()`
- From file: `textFile()`
- From another RDD: `map()`

RDD Lineage

RDDs track their transformation history, enabling automatic recomputation of lost partitions.

What is DataFrame?

Definition

A distributed collection of data organized into named columns, similar to a database table or pandas DataFrame.

Key Features:

- Has defined **schema**
- **Catalyst** optimizer
- **Tungsten** execution engine
- SQL + DSL APIs

Creating DataFrames:

- `spark.read.csv()`
- `spark.read.json()`
- `spark.read.parquet()`
- `spark.createDataFrame()`

RDD vs DataFrame Comparison

Aspect	RDD	DataFrame
Abstraction	Low-level	High-level
Schema	No schema	Has schema
Optimization	No automatic	Catalyst + Tungsten
Performance	Slower	**Faster**
Memory Usage	Higher (Java objects)	Lower (off-heap)
APIs	Functional transformations	SQL + DSL
Use Case	Unstructured data	Structured data

Recommendation

Use DataFrames for most use cases. Use RDDs only when you need fine-grained control over physical execution or work with unstructured data.

Why DataFrames are Faster

Catalyst Optimizer:

- Analyzes query plans
- Applies optimizations:
  - Predicate pushdown
  - Column pruning
  - Constant folding
- Generates optimized execution

Tungsten Engine:

- Off-heap memory management
- Avoids JVM garbage collection
- Cache-aware computation
- Runtime code generation
- Columnar storage format

Performance Gain

DataFrame operations can be **10-100x faster** than equivalent RDD operations!

Lazy Evaluation

Definition

Transformations are not executed immediately. Spark records them in a DAG and executes only when an action is called.

Think of it like a recipe book:
Transformations = Writing recipe steps (no cooking)
Actions = Actually cooking the dish (execution)

Example

`df.filter(...).select(...).groupBy(...)` → **Not executed yet!**
`df.count()` → **NOW everything executes!**

Transformations vs Actions

Transformations (Lazy):

- `select()`
- `filter()` / `where()`
- `map()` / `flatMap()`
- `groupBy()`
- `orderBy()` / `sort()`
- `join()` / `union()`
- `withColumn()` / `drop()`

Actions (Trigger Execution):

- `show()`
- `count()`
- `collect()`
- `first()` / `take(n)`
- `reduce()`
- `foreach()`
- `write.save()`

Benefits of Lazy Evaluation

Query Optimization:

- Predicate pushdown
- Column pruning
- Combined operations

Fault Tolerance:

- Lineage enables recomputation
- Lost partitions recovered automatically

Memory Efficiency:

- No intermediate materialization
- Stream processing through pipeline

Pipelining:

- Narrow transformations combined
- Single pass through data

Common Pitfall

Multiple actions re-execute entire DAG! Use `.cache()` for reused DataFrames.

Notebook Magic Commands

Command	Description	Example
`%python`	Execute Python code	Default in notebooks
`%sql`	Execute SQL query	`%sql SELECT * FROM table`
`%scala`	Execute Scala code	Performance-critical code
`%r`	Execute R code	Statistical analysis
`%fs`	File system operations	`%fs ls /data/`
`%sh`	Shell commands	`%sh pip install pandas`
`%md`	Markdown rendering	Documentation
`%run`	Run another notebook	`%run ./utilities`

Key Feature

Seamlessly switch between languages in the same notebook for maximum flexibility!

Practical Tasks: E-Commerce Analysis

Objective

Apply Spark DataFrame operations to analyze e-commerce sales data.

1. **Task 1:** Upload Sample E-Commerce CSV
2. **Task 2:** Read Data into DataFrame
3. **Task 3:** Basic DataFrame Operations
   - SELECT - Choosing columns
   - FILTER - Filtering rows
   - GROUPBY - Aggregating data
   - ORDERBY - Sorting results
4. **Task 4:** Export Results

SELECT - Choosing Columns

Purpose

Extract specific columns from a DataFrame.

Methods:

- String names: `df.select("col1", "col2")`
- Using `col()`: `df.select(col("col1"))`
- DataFrame ref: `df.select(df.col1)`

With Transformations:

- Alias: `col("a").alias("new_name")`
- Expression: `selectExpr("a * 2")`

Example

```
df.select("order_id", "product_name", "total_amount")
```

FILTER / WHERE - Filtering Rows

Purpose

Keep only rows that satisfy certain conditions.

Operation	**Syntax**	**Example**
Equals	`==`	`col("city") == "NYC"`
Not equals	`!=`	`col("city") != "NYC"`
Greater/Less	>, <, >=, <=	`col("amount") > 100`
AND	`&`	`(cond1) & (cond2)`
OR	`|`	`(cond1) | (cond2)`
IN list	`.isin()`	`col("city").isin(["A","B"])`
LIKE	`.like()`	`col("name").like("%Pro%")`

GROUPBY - Aggregating Data

Purpose

Group rows by column values and perform aggregate calculations.

Aggregation Functions:

- `count()` - Count rows
- `sum()` - Sum values
- `avg()` / `mean()` - Average
- `min()` / `max()` - Min/Max
- `countDistinct()` - Unique count

Example:

- `df.groupBy("category")`
- `.agg(`
- `count("*").alias("orders"),`
- `sum("amount").alias("total")`
- `)`

ORDERBY / SORT - Sorting Data

Purpose

Sort rows by one or more columns.

Ascending (Default):

- `df.orderBy("column")`
- `df.orderBy(col("a").asc())`

Descending:

- `df.orderBy(col("a").desc())`
- `df.orderBy(desc("a"))`

Multiple Columns

```
df.orderBy(col("category").asc(), col("amount").desc())
```

Exporting Results

Format	Command
CSV	`df.write.mode("overwrite").csv("/output")`
Single CSV	`df.coalesce(1).write.csv("/output")`
Parquet	`df.write.parquet("/output")`
JSON	`df.write.json("/output")`
Table	`df.write.saveAsTable("db.table")`
Partitioned	`df.write.partitionBy("col").parquet()`

Write Modes

overwrite - Replace • **append** - Add • **ignore** - Skip if exists • **error** - Fail if exists

Recommendation

Use **Parquet** for big data (columnar, efficient). Use CSV for interoperability.

Key Takeaways

Architecture

- Driver = Brain (scheduling)
- Executors = Workers (execution)
- DAG = Optimization graph
- Avoid shuffles when possible

Best Practices

- Prefer DataFrames over RDDs
- Cache reused DataFrames
- Use Parquet for storage
- Watch out for `collect()`

Spark = In-memory + DAG optimization + Unified platform

Thank You!

Questions?

Connect with me:

linkedin.com/in/yashkavaiya

Gen AI Guru