Understanding Apache Spark Architecture and PySpark Job Execution

8 min read · May 6, 2025

DataWithSantosh Follow

What is Apache Spark?

Apache Spark is an open-source, distributed computing framework designed for **big data processing.** It's fast, scalable, and handles large datasets across multiple computers (a cluster). Think of Spark as a super-efficient chef who can cook massive meals by coordinating many kitchen assistants (computers) to work together.

Spark is used for:

— **Batch processing** (e.g., processing historical sales data).

— **Stream processing** (e.g., analyzing live social media feeds).

— **Machine learning** (e.g., training models on large datasets).

— **SQL queries** (e.g., querying big data like a database).

Spark's key advantage is its **in-memory processing**, which makes it much faster than older systems like Hadoop MapReduce, as it keeps data in memory rather than constantly reading from disk.

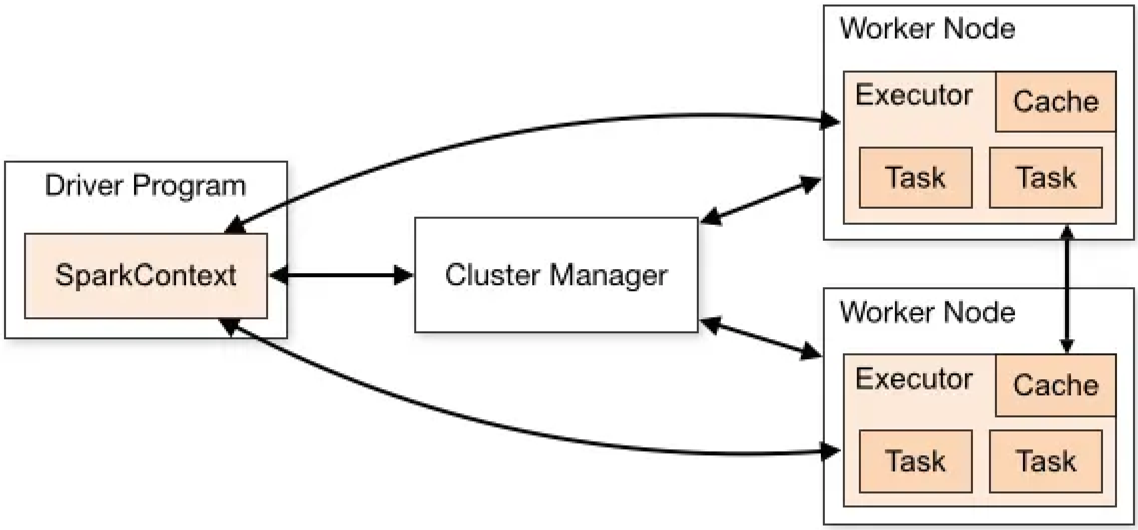

image Source: https://spark.apache.org/

Apache Spark Architecture: The Big Picture

Spark's architecture is like a well-organized factory:

— There's a **manager** (Spark Driver) who plans the work.

— A **supervisor** (Cluster Manager) assigns tasks to workers.

— **Workers** (Executors) do the actual work on different machines.

Here's a visual representation of Spark's architecture:

```
[ Spark Application ]
       |
       v
[ Driver Program ]
   - SparkContext
   - Creates DAG (logical plan)
   - Coordinates with Cluster Manager
       |
       v
[ Cluster Manager ]
   - Allocates resources
   - Assigns tasks to Executors
       |
       v
[ Executors (on Worker Nodes) ]
   - Run tasks
   - Store data in memory/disk
   - Report results to Driver
```

Let’s break down each component and how they work together.

Components of Apache Spark Architecture

1. Spark Driver

- **What is it?** The Driver is the brain of a Spark application. It runs the main program and coordinates the entire job.
- **Key Responsibilities:**
- Creates the **SparkContext**, which is like the entry point to Spark. It connects your program to the Spark cluster.
- Converts your code (e.g., Python, Scala) into a **Directed Acyclic Graph (DAG)**, a logical plan of tasks.
- Splits the DAG into **stages** and **tasks** and sends tasks to Executors.
- Tracks the progress of tasks and handles failures.
- **Analogy**: Think of the Driver as a project manager who creates a to-do list, assigns tasks to team members, and checks if everything is done.

- **Example:** If you write a PySpark program to count words in a large text file, the Driver translates your code into a plan (e.g., "read file, split words, count them") and assigns tasks to workers.

2. Cluster Manager

- **What is it?** The Cluster Manager is the resource allocator. It manages the cluster's resources (CPU, memory) and assigns tasks to worker nodes.
- **Types of Cluster Managers**:
- **Standalone**: Spark's built-in manager, simple for small clusters.
- **YARN**: Used in Hadoop ecosystems, good for shared clusters.
- **Mesos**: A general-purpose cluster manager.
- **Kubernetes**: Modern option for containerized environments (e.g., on GCP).
- **Key Responsibilities**:
- Allocates resources (e.g., CPU cores, memory) to your Spark application.
- Assigns Executors to worker nodes.
- Monitors resource usage and handles node failures.
- **Analogy**: Think of the Cluster Manager as a factory supervisor who decides which machines (worker nodes) will do the work and ensures they have enough tools (resources).
- **Example**: On Google Cloud's Dataproc (a Spark cluster service), the Cluster Manager (YARN) assigns Executors to virtual machines in your cluster.

3. Executors

- **What are they?** Executors are worker processes running on worker nodes (computers in the cluster). Each Executor handles a portion of the data and tasks.
- **Key Responsibilities**:
- Execute tasks assigned by the Driver.
- Store data in memory (or disk if memory is full) for fast processing.

- Send results back to the Driver.
- **Analogy**: Executors are like factory workers who follow the manager's instructions, process raw materials (data), and produce results.
- **Example**: If your job is to filter a 1TB dataset, each Executor processes a chunk of the data (e.g., 100GB) in parallel.

4. Worker Nodes

- **What are they?** Physical or virtual machines in the cluster that host Executors.
- **Role**: Provide the computing power (CPU, memory, disk) for Executors to run tasks.
- **Example**: In a GCP Dataproc cluster, worker nodes are Google Compute Engine VMs.

5. SparkContext

- **What is it?** The main entry point for interacting with Spark, created by the Driver.
- **Role**: Manages the connection to the cluster, tracks resources, and coordinates job execution.
- **Example**: In PySpark, you create a `SparkContext` (or `SparkSession` in newer versions) to start your application:

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

How These Components Work Together

Here's how a Spark job flows through the architecture: 1. You submit a Spark application (e.g., a PySpark script).

2. The **Driver** starts, creates a **SparkContext**, and builds a **DAG** (logical plan) from your code.

3. The Driver communicates with the **Cluster Manager** to request resources.

4. The Cluster Manager allocates **Executors** on **Worker Nodes**.

5. The Driver breaks the DAG into **stages** (groups of tasks) and sends **tasks** to Executors.

6. Executors process tasks in parallel, storing data in memory for speed.

7. Executors send results back to the Driver.

8. The Driver collects results and either outputs them (e.g., to a file) or continues with the next stage.

Key Concepts in Spark Architecture

1. DAG (Directed Acyclic Graph)

- Spark creates a DAG to represent your job as a series of operations (e.g., read, filter, group).
- The DAG is split into **stages**, where each stage contains tasks that can run in parallel.
- **Example**: If you filter a dataset and then group it, Spark creates a DAG with two stages: one for filtering, one for grouping.

2. Stages and Tasks

- A **stage** is a group of tasks that can run without shuffling data across nodes (e.g., filtering rows).
- A **task** is the smallest unit of work, executed by an Executor on a data partition.
- **Example**: If you have 10GB of data split into 10 partitions, Spark creates 10 tasks, each processing 1GB.

3. Data Partitions

- Spark splits large datasets into smaller chunks called **partitions**, processed in parallel by Executors.
- **Example**: A 1TB file might be split into 1000 partitions, each 1GB, processed by multiple Executors.

4. In-Memory Processing

- Spark stores data in memory (RAM) to avoid slow disk reads, making it much faster than Hadoop.

- If memory is full, Spark spills data to disk but optimizes to minimize this.

5. Fault Tolerance

- Spark achieves fault tolerance through **lineage** (the DAG tracks how data was created).
- If an Executor fails, Spark re-runs tasks on another node using the lineage, ensuring no data is lost.
- **Example:** If a node crashes while filtering data, Spark re-executes the filtering task on another node.

. . .

Visualizing Spark's Workflow

Here's a simplified diagram of a Spark job processing a dataset:

Here's a simplified diagram of a Spark job processing a dataset:

```
[Your Code: Read CSV -> Filter -> GroupBy -> Count]
       |
       v
[Driver: Creates DAG]
       |
       v
[Cluster Manager: Allocates 3 Executors]
       |        |        |
       v        v        v
[Executor 1] [Executor 2] [Executor 3]
   (Process   (Process   (Process
    Partition  Partition  Partition
      1)        2)         3)
       |        |        |
       v        v        v
[Results sent back to Driver]
       |
       v
[Output: Save or Display]
```

Practical Example: Word Count in Spark

Let's see how the architecture works with a classic word count example: — **Goal:** Count the frequency of words in a large text file (e.g., 100GB). — **Code** (in PySpark):

```
from pyspark.sql import SparkSession



spark = SparkSession.builder.appName("WordCount").getOrCreate()
  text = spark.read.text("hdfs://largetextfile.txt")
  words = text.rdd.flatMap(lambda line: line[0].split(" "))
  wordcounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
  wordcounts.saveAsTextFile("hdfs://output")
  spark.stop()
```

How It Works in the Architecture:

1. **Driver:**

- Creates a `SparkSession` (includes SparkContext).
- Builds a DAG: read file → split into words → count words → save output.
- Splits the DAG into stages (e.g., read + split, count, save).

2. Cluster Manager:

- Allocates, say, 10 Executors on 10 worker nodes.

3. Executors:

- Each Executor processes a partition of the file (e.g., 10GB each).
- Tasks: Split lines into words, count occurrences, aggregate counts.
- Store intermediate data (word counts) in memory.

4. Data Flow:

- Executors send partial counts (e.g., `("hello", 100)`) to the Driver.
- The Driver combines results and saves them to HDFS.

5. Fault Tolerance:

- If an Executor fails, the Driver uses the DAG to re-run tasks on another node.

How a PySpark Job Works When Submitted

Now, let's walk through the end-to-end process of submitting a PySpark job, assuming you're running it on a GCP Dataproc cluster (a common setup for a GCP Data Engineer role).

Step 1: Write the PySpark Code

You write a Python script (`word_count.py`) like the word count example above.

Step 2: Submit the Job

You submit the job to the cluster using the `spark-submit` command:

```
spark-submit --master yarn word_count.py
```

`--master yarn` tells Spark to use YARN as the Cluster Manager (common on Dataproc).

- The script is sent to the Driver node.

Step 3: Driver Initialization

- The **Driver** starts on the master node (or a designated node in the cluster).
- It creates a **SparkSession** (or SparkContext) to connect to the cluster.
- The Driver parses your code and builds a **DAG** of operations (e.g., read, transform, save).

Step 4: Resource Allocation

- The Driver contacts the **Cluster Manager** (YARN on Dataproc).
- YARN allocates resources (e.g., 10 Executors with 4GB memory each) across worker nodes.
- Each Executor is a JVM process running on a worker node.

Step 5: DAG and Task Creation

- The Driver divides the DAG into **stages** based on operations that require data shuffling (e.g., `reduceByKey` in word count requires shuffling).
- Each stage is broken into **tasks** (one task per partition).

- The Driver sends tasks to Executors via the Cluster Manager.

Step 6: Task Execution

- Each **Executor**:
- Receives tasks from the Driver.
- Processes its assigned data partition (e.g., a chunk of the input file).
- Stores intermediate results in memory (e.g., partial word counts).
- Performs transformations (e.g., `flatMap`, `map`) and actions (e.g., `reduceByKey`).
- If a shuffle is needed (e.g., grouping words), Executors exchange data across nodes.

Step 7: Fault Tolerance and Recovery

- If an Executor fails (e.g., node crashes), the Driver detects it via heartbeats.
- The Driver uses the DAG's lineage to re-run failed tasks on another Executor.
- YARN may allocate a new Executor if needed.

Step 8: Result Collection

- Executors send results (e.g., final word counts) back to the Driver.
- The Driver combines results and performs any final actions (e.g., saving to Cloud Storage).

Step 9: Job Completion

- The Driver saves the output (e.g., to `gs://output` on GCP).
- The SparkSession is closed, and resources are released by the Cluster Manager.
- You see the job's output or logs in the Dataproc UI or command line.

Example on GCP Dataproc

- You create a Dataproc cluster with 1 master node and 5 worker nodes.
- Submit the job: `gcloud dataproc jobs submit pyspark --cluster=my-cluster word_count.py`.
- YARN allocates Executors to worker nodes.

- The job processes a 100GB file stored in Google Cloud Storage (`gs://input`).
- Output is saved to `gs://output`.

Summary

- **Spark Architecture:**
- **Driver:** Plans and coordinates the job, creates DAG.
- **Cluster Manager:** Allocates resources (e.g., YARN on Dataproc).
- **Executors:** Process data in parallel on worker nodes.
- **SparkContext:** Connects your program to the cluster.
- **PySpark Job:**
- Submit code → Driver builds DAG → Cluster Manager allocates Executors → Tasks run in parallel → Results collected → Output saved.
- **Key Features:** In-memory processing, fault tolerance via lineage, scalability for big data.

Note

If this article helped you gain some knowledge, please clap and comment. Don't forget to follow me on Medium . Your support helps me create more content like this and keeps us connected in the data engineering community. Thank you!

Pyspark Spark Architecture Apache Spark Data Engineering

Written by DataWithSantosh

A GCP Data Engineer sharing cutting-edge data insights.For the latest in data engineering!