Day 3 – Hadoop & Spark Core with Scala

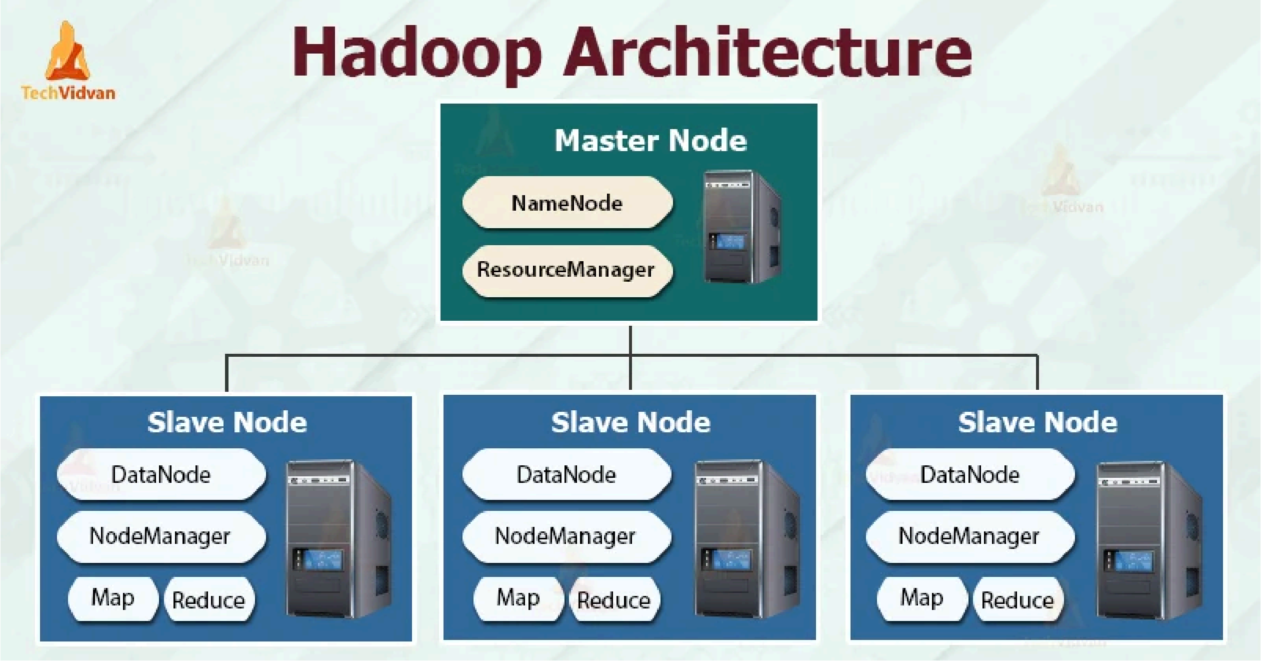
By Dhandapani Yedappalli Krishnamurthi Oct 7, 2025

1. Big Data Ecosystem Overview

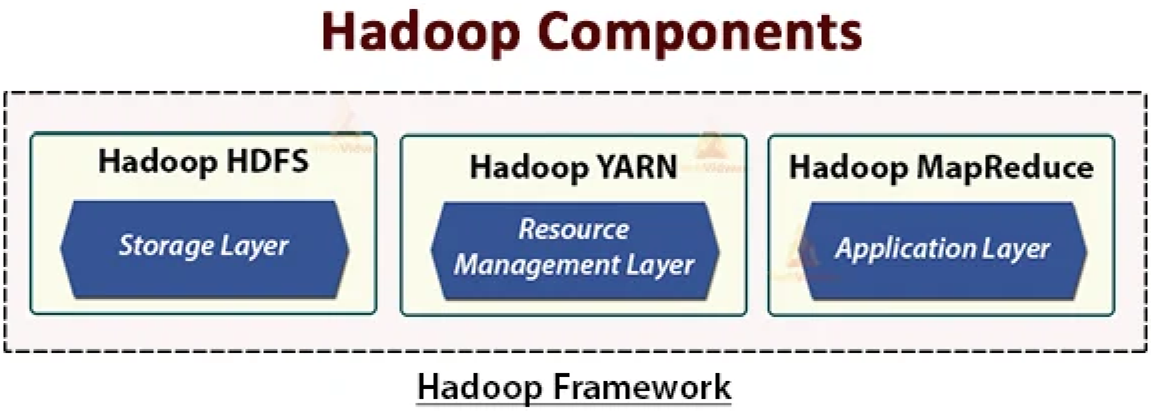
Tool	Purpose
Hadoop	Distributed storage (HDFS) and processing (MapReduce/YARN)
Hive	SQL layer on top of Hadoop for querying
Spark	In-memory distributed computation engine
Kafka	Distributed message broker for streaming data

Illustration:

```
[Kafka Stream] → [Spark Streaming] → [HDFS / Hive Storage]
                   ↑
Real-time producers (apps, sensors)
```



✅ Hadoop = Storage
✅ Spark = Compute
✅ Hive = SQL Layer
✅ Kafka = Streaming



---

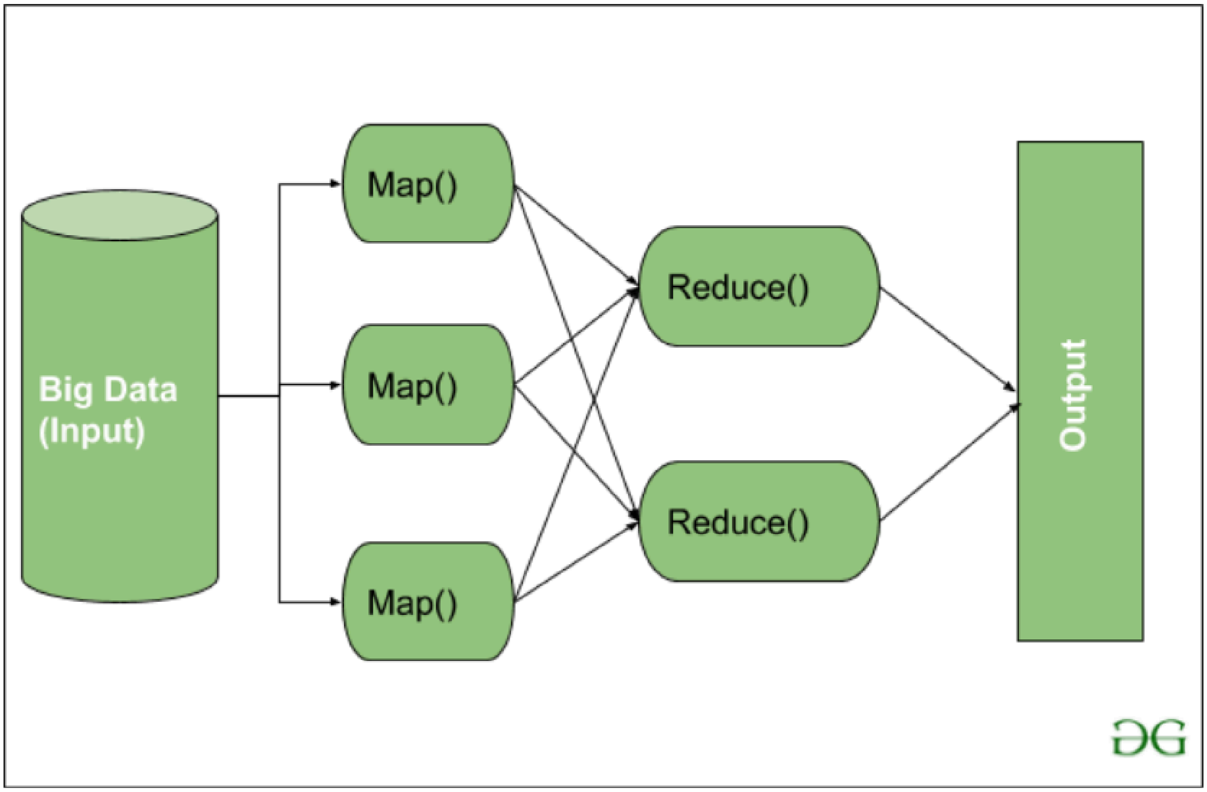
2. Hadoop Architecture

Components:

1. **HDFS (Hadoop Distributed File System)**
   - Stores large files across multiple machines.
   - Has **NameNode (metadata)** and **DataNodes (storage)**.
2. **YARN (Yet Another Resource Negotiator)**
   - Manages cluster resources and job scheduling.
3. **ResourceManager**
   - Allocates compute containers for Spark/Hive/MapReduce jobs.

⚙️ **Illustration:**

```
Client → ResourceManager → NodeManagers
       ↘
        ↘  HDFS: NameNode + DataNodes
```



---

3. Spark Architecture

🧩 **Components:**

Component	Description
Driver	Controls the Spark application (runs main function).
Executor	Worker processes executing tasks.
Cluster Manager	Allocates resources (YARN, Standalone, Mesos, Kubernetes).

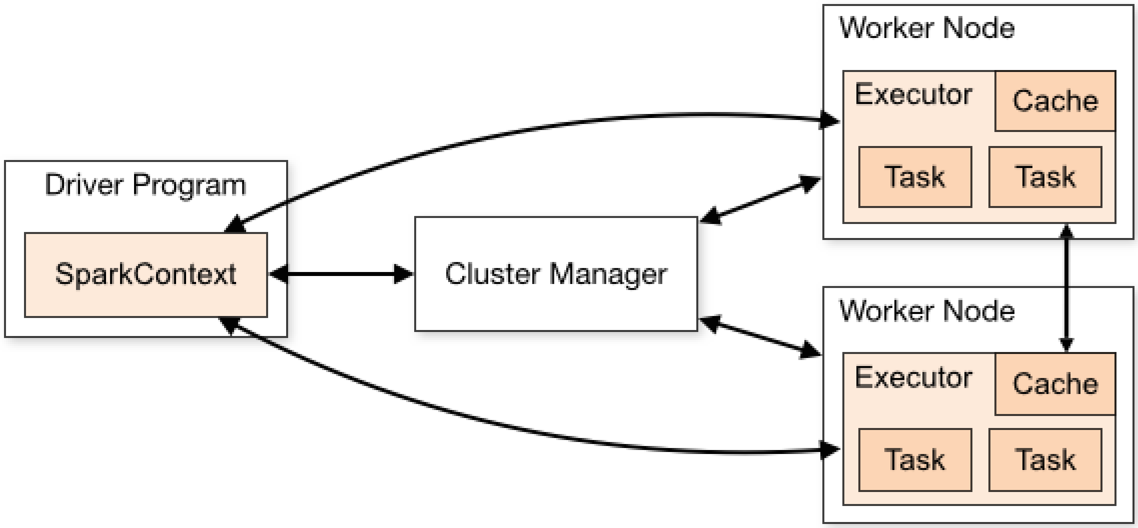


Illustration:

```
Driver Program
               |
               |--> Cluster Manager (YARN)
                       |
                       |--> Executors (on worker nodes)
```

Spark job → split into **Stages → Tasks → Executors**

🔹 4. RDD Programming in Scala

RDD (Resilient Distributed Dataset)

- Immutable distributed collection of objects.
- Built from files, HDFS, or other data sources.
- Two operations: **Transformations** and **Actions**.

✅ **Example: Creating RDDs**

```
import org.apache.spark.sql.SparkSession
```

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

val sc = spark.sparkContext

// Create RDD from collection
val data = Seq(1,2,3,4,5)
val rdd1 = sc.parallelize(data)

// Create RDD from text file
val rdd2 = sc.textFile("hdfs:///data/sales.txt")
```

🔹 5. Transformations and Actions

Type	Operation	Example
Transformation	map, filter, flatMap, reduceByKey	Lazy (not executed immediately)
Action	count, collect, saveAsTextFile	Triggers execution

✅ **Example:**

```
val numbers = sc.parallelize(1 to 10)
val evens = numbers.filter(_ % 2 == 0)
val squares = evens.map(x => x * x)

println("Sum of squares: " + squares.reduce(_ + _))
```

---

6. Pair RDDs and Key-Value Operations

```
val sales = sc.parallelize(Seq(
  ("Electronics", 2000),
  ("Clothing", 1000),
  ("Electronics", 3000)
))
```

```
// reduceByKey aggregates values by key
val totals = sales.reduceByKey(_ + _)
totals.collect().foreach(println)
```

Output:

```
(Electronics,5000)
(Clothing,1000)
```

---

7. Caching and Persistence

Caching stores RDDs in memory for reuse in iterative computations.

```
val data = sc.textFile("hdfs:///data/transactions.txt")
val cleanData = data.filter(!_.contains("NULL")).cache()

println(cleanData.count())
println(cleanData.take(5).mkString("\n"))
```

---

8. Spark DataFrame API Basics

DataFrame = distributed table with named columns (similar to SQL).

```
import spark.implicits._

val df = spark.read.option("header",
"true").csv("hdfs:///data/sales.csv")

df.printSchema()
df.show(5)
```

---

9. Schema Inference

Spark can **infer schema** automatically from CSV/JSON files.

```
val df = spark.read
  .option("header", "true")
  .option("inferSchema", "true")
  .csv("hdfs:///data/customers.csv")

df.printSchema()
```

---

10. DSL Queries (select, filter, groupBy, agg, join)

```
import org.apache.spark.sql.functions._

// Select and filter
val filtered = df.select("region",
"amount").filter($"amount" > 1000)

// GroupBy and aggregate
val totalSales =
df.groupBy("region").agg(sum("amount").alias("total_sales"))

// Join
val customers = spark.read.option("header",
"true").csv("hdfs:///data/customers.csv")
val joined = df.join(customers, "cust_id")

joined.show()
```

---

11. UDFs (User Defined Functions) in Scala

You can define custom functions to use in Spark SQL transformations.

```
import org.apache.spark.sql.functions.udf

val addTax = udf((amount: Double) => amount * 1.18)
val taxedDF = df.withColumn("taxed_amount",
addTax($"amount"))

taxedDF.show()
```

---

12. Integrating Spark with Hadoop (HDFS I/O)

✅ Reading from HDFS

```
val hdfsData = spark.read.text("hdfs:///data/input.txt")
```

✅ Writing to HDFS

```
hdfsData.write.mode("overwrite").text("hdfs:///data/output/"
)
```

---

13. Hands-on Lab 1: Read → Transform → Write (RDDs)

```
val inputRDD = sc.textFile("hdfs:///etl/input/sales.txt")

val cleanRDD = inputRDD
  .filter(line => !line.contains("NULL"))
  .map(line => line.split(","))
  .map(arr => (arr(0), arr(1).toDouble))
  .reduceByKey(_ + _)
```

```
cleanRDD.saveAsTextFile("hdfs:///etl/output/sales_summary")
```

---

14. Hands-on Lab 2: Simple ETL using DataFrame API

```
val df = spark.read.option("header",
"true").csv("hdfs:///etl/input/sales.csv")

val transformed = df
  .filter($"amount" > 1000)
  .withColumn("amount_with_tax", $"amount" * 1.18)
  .groupBy("region")
  .agg(sum("amount_with_tax").alias("region_total"))

transformed.write.mode("overwrite").parquet("hdfs:///etl/out
put/region_summary/")
```

🎯 Outcome:

By the end of Day 3, you can:
✅ Build Spark ETL jobs in Scala
✅ Read and write to HDFS
✅ Perform transformations (RDD & DataFrame APIs)
✅ Define UDFs
✅ Optimize using caching