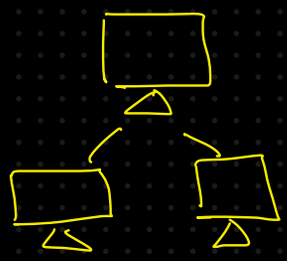
Spark architecture & Components

Apache Spark is able to work on large datasets quickly by distributing the work load across multiple machines in a cluster.

How Spark can run?

Standalone mode → uses its own cluster manager
good for small to medium tasks
simple to configure



Some cluster manager → YARN

Loosely coupled

Spark can integrate easily with large scale cluster managers

Large scale production env.

Distributed nature of Spark

data → partition → distributed processing

100000 papers → 1 person

→ 1000 → 100 people
100 partitions

Massive datasets

MPP → massive parallel processing →

In-memory processing

Unlike tradition systems, Spark does the comp. in memory

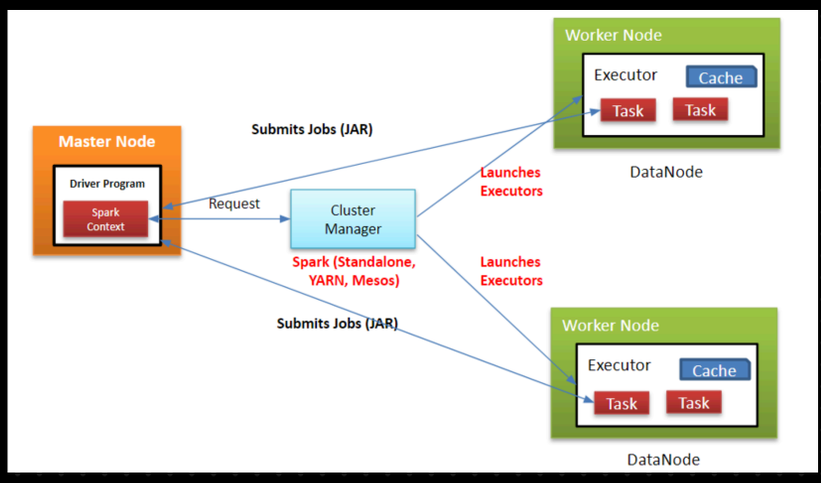
2^{10}

2^1 2^2 2^3 2^4 2^5 ... 2^{10}

→ 2 4 8 16 32 1024

1024 ←

Spark Architecture and its Component



Each component in spark architecture plays a specific role and work together with others to ensure efficient data processing.

Driver Program

entry point to our spark application

main()

Responsibilities

- User defined → DAG
- scheduling task and managing the execution plan
- Collecting results from executors

Runs on the master node

Spark context is created here

2. Cluster Manager

manages and allocate resources

→ Standalone
→ YARN
→ mesos
→ Kubernetes

3. Worker nodes

responsibilities

→ Execute task assigned by the driver
→ Perform data transform. and action
→ Store intermediate results in memory and disk

Executors work & report back to driver

Tasks

Smallest unit of work

Created by splitting data → partition

Each task processes 1 partition

1 Core → 1 task
↓
1 partition

DAG Scheduler

high level transformation (map, filter)
↓
series of stages

Each stage consist of tasks which can parallel

SQL catalyst Optimizer

Optimization techniques to our SQL Queries

efficient execution plan for Query processing

Step by step Execution Process

1. Job Submission
2. DAG creation
3. Task division
4. Resource Allocation
5. Task execution
6. Intermediate Storage
7. Result Collection

Each component interact with another

- **Driver** communicates with the **cluster manager** to request resources.
- **Cluster manager** assigns resources (executors) to the driver.
- **Executors** perform the actual computation on partitions of data.
- **Tasks** are executed by executors, ensuring parallel processing.
- Intermediate results are stored in the **storage layer** if memory is insufficient.

Spark Context

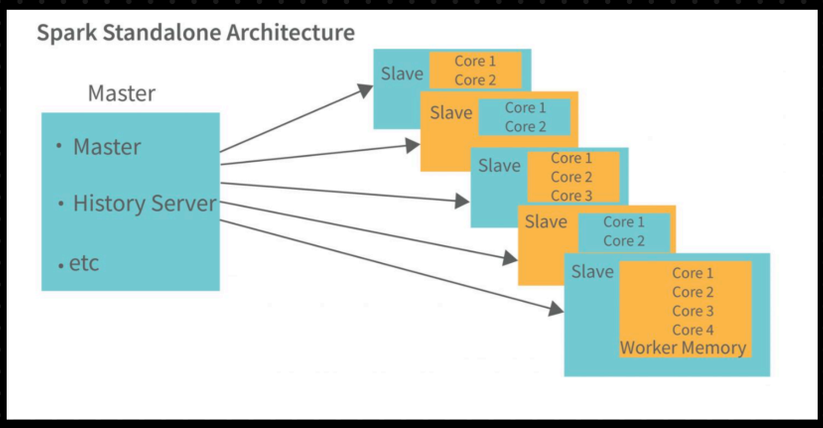
Executors

Task : Smallest unit of work

RDD

DAG

Spark Standalone architecture



Master

lead of spark standalone cluster

Central point and entry to spark cluster

→ managing and distributing tasks to the workers

→ Checks periodically if workers are alive.

→

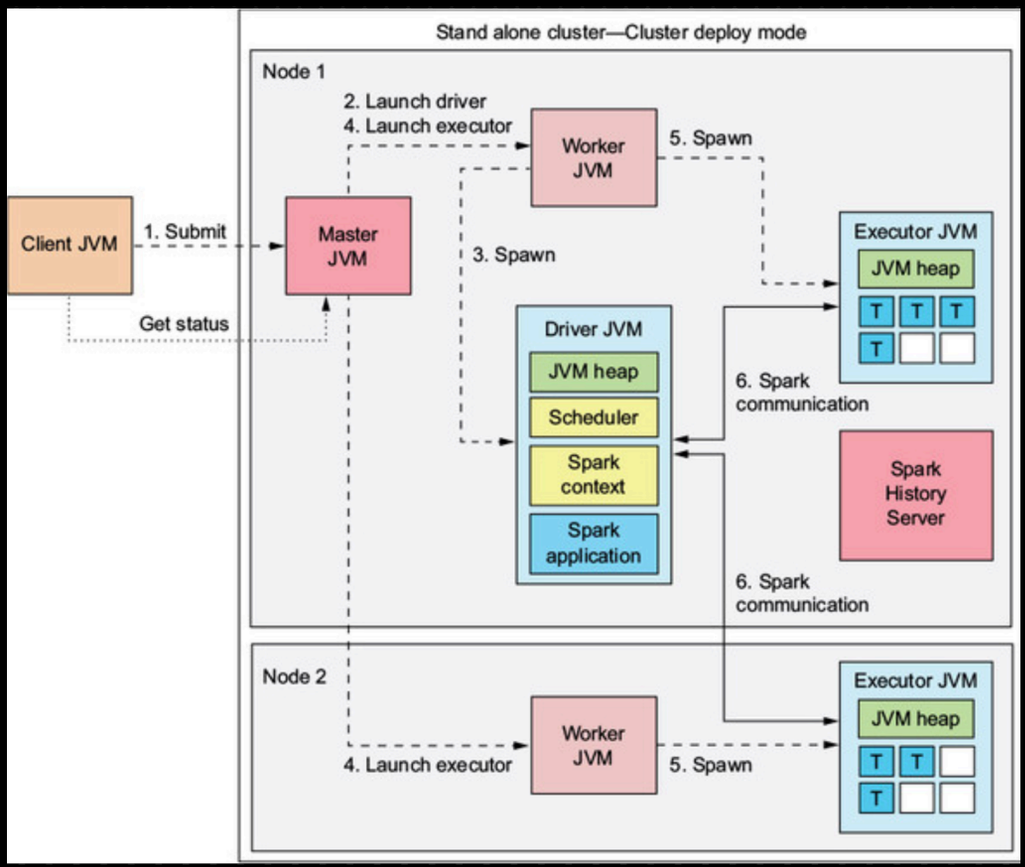
Worker

recieve the task from master and execute

each worker ⟶ executors JVMs ⟶ Communicate with master & executors for Job execution

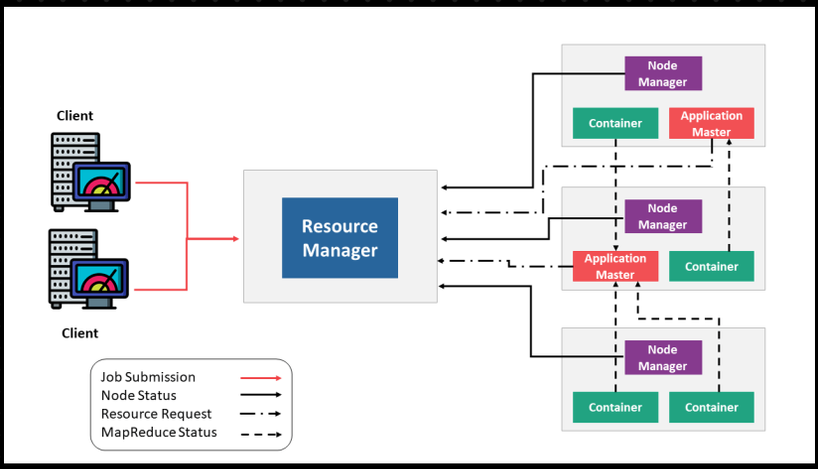
↓

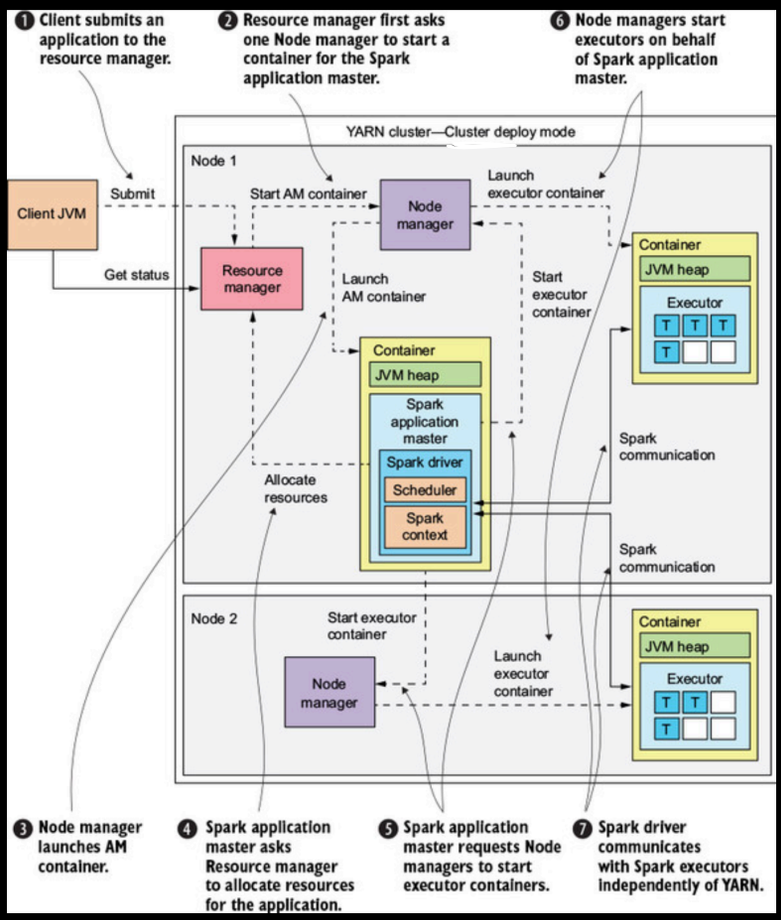
managing resources



1. A Spark application is submitted to the Master.
2. The Master allocates resources on the Worker nodes.
3. The Master tells the Workers to launch Executors.
4. The Driver program (in its JVM) divides the application into tasks.
5. The Scheduler assigns tasks to the Executors.
6. The Executors execute the tasks in parallel.
7. The results are communicated back to the Driver.
8. The History Server records the job execution details.

Spark on YARN architecture





Drive Program

→ Client mode

→ Cluster mode

Aspect	Standalone Architecture	YARN Architecture
Cluster Manager	Built-in Spark cluster manager.	YARN (part of Hadoop ecosystem).
Resource Management	Spark directly manages resources.	YARN manages resources and assigns containers.
Scalability	Suitable for small to medium-scale clusters.	Designed for large-scale production clusters.
Driver Location	Runs on the master node of the Spark cluster.	Can run on a client machine (client mode) or YARN (cluster mode).
Fault Tolerance	Limited to Spark's built-in mechanisms.	Relies on YARN's fault tolerance and resource allocation.
Ease of Use	Simple to set up and manage.	Requires integration with Hadoop YARN.
Integration	Primarily used as a standalone framework.	Seamlessly integrates with Hadoop tools like Hive and HBase.
Executor Management	Managed directly by Spark.	Executors run inside YARN-managed containers.
Application Monitoring	Uses Spark UI for job monitoring.	Uses both Spark UI and YARN Web UI for monitoring.

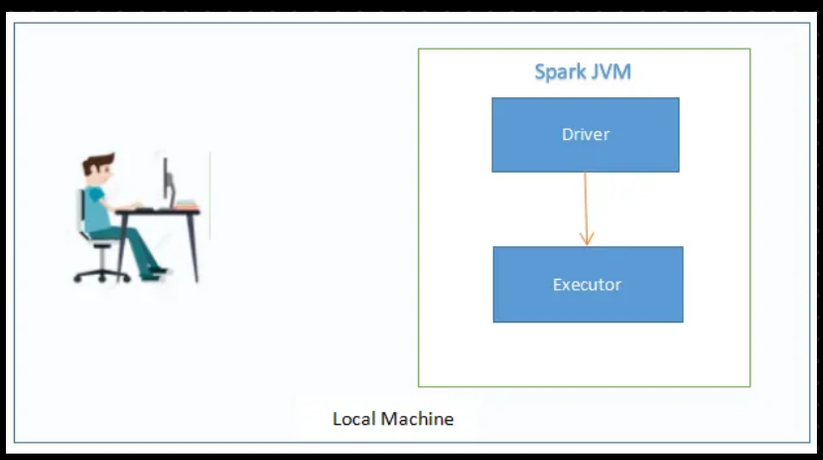
Spark Deployement Modes

multiple deployement mode

→ gives us flexibility in running application

These modes dictate where the driver program runs and how executors interact with driver.

1. Local Mode

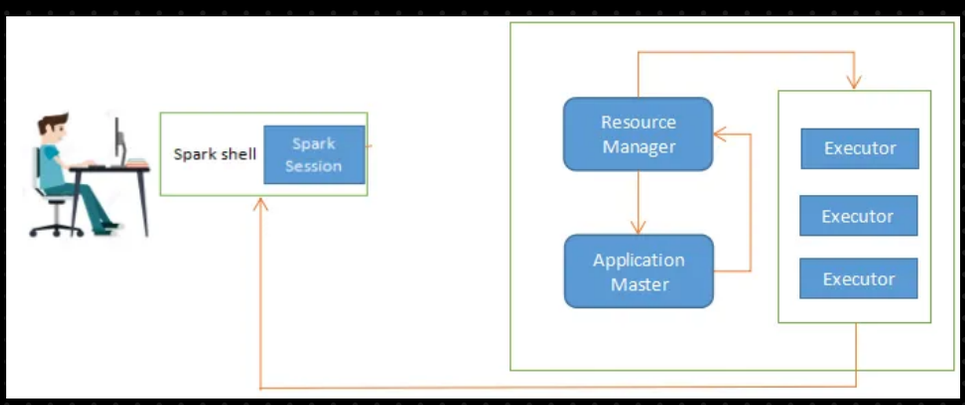


Uses one or multiple threads for parallelism

→ Dev & testing

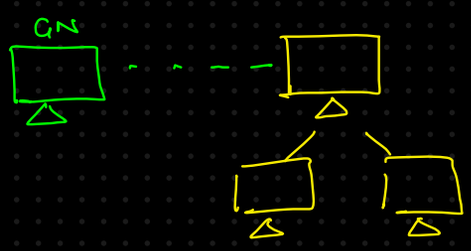
→ small scale

2. Client Mode



drivers program runs on client machine

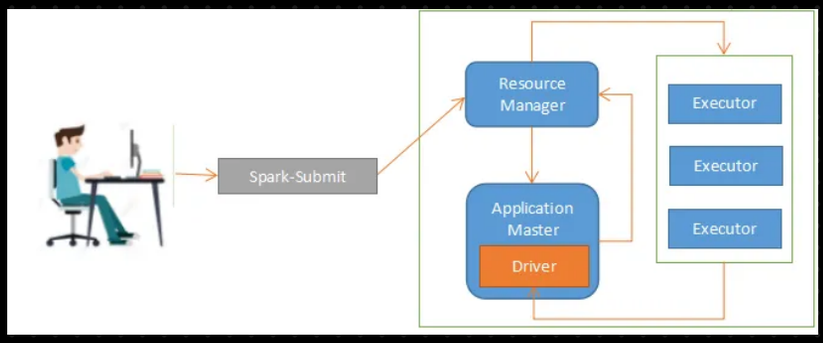
Live by live run debugging learning



Cluster Mode

Good for Production runs

Jobs with high availability



Aspect	Local Mode	Client Mode	Cluster Mode
Driver Location	Same machine as the application	On the client machine (job submission node)	On one of the cluster nodes
When to Use	Development/testing	Interactive applications or monitoring jobs	Production-grade, automated jobs
Resource Management	Limited to local machine	Executors on cluster, driver on client	Both driver and executors on the cluster
Fault Tolerance	Minimal	Depends on cluster manager (executor fault-tolerant)	High (driver is fault-tolerant in cluster)
Communication Latency	Low	Higher due to driver-cluster communication	Minimal latency
Cloud Examples	Local VM for development	Jupyter on Dataproc or EMR	Spark-submit jobs on AWS EMR/GCP Dataproc

Spark session

Unified entry platform to interact with Spark cluster

Spark 2.0

Spark context
SQL context
Hive context
} Single abstraction

Unified Context
Cluster Connectivity
Configuration options

Spark context
Spark conf ⟸

Property/Method	Description	Example
appName(name)	Sets the name of the Spark application.	spark = SparkSession.builder.appName("MySparkApp").getOrCreate()
master(url)	Specifies the cluster manager for the Spark application. Options: local, yarn, or mesos.	spark = SparkSession.builder.master("yarn").getOrCreate()
config(key, value)	Sets Spark configurations. Commonly used for specifying parameters like memory, shuffle partitions, and warehouse directory.	spark = SparkSession.builder.config("spark.executor.memory", "4g").config("spark.sql.shuffle.partitions", "100").getOrCreate()
enableHiveSupport()	Enables Hive support for accessing Hive metastore and querying Hive tables.	spark = SparkSession.builder.enableHiveSupport().getOrCreate()
getOrCreate()	Returns an existing Spark Session if available; otherwise, it creates a new one. Ensures there's only one Spark context in the application.	spark = SparkSession.builder.appName("MySparkApp").getOrCreate()
sparkContext	Returns the SparkContext associated with the Spark Session.	sc = spark.sparkContext
sql("query")	Executes a SQL query and returns the result as a DataFrame.	df = spark.sql("SELECT * FROM customers")
stop()	Stops the current Spark Session and releases cluster resources.	spark.stop()
spark.sql.warehouse.dir	Specifies the default directory for Spark SQL-managed tables (managed tables store both metadata and data here).	config("spark.sql.warehouse.dir", "/user/hive/warehouse")
spark.executor.memory	Allocates memory to executors (worker processes).	config("spark.executor.memory", "4g")
spark.sql.shuffle.partitions	Defines the number of partitions created during shuffle operations (e.g., joins, groupBy).	config("spark.sql.shuffle.partitions", "100")
spark.driver.memory	Sets memory allocation for the driver program.	config("spark.driver.memory", "2g")
spark.ui.port	Sets the port for Spark's Web UI for monitoring.	config("spark.ui.port", "4040")
spark.executor.cores	Sets the number of cores for each executor.	config("spark.executor.cores", "4")
spark.sql.inMemoryColumnarStorage.compressed	Enables in-memory compression for DataFrame columns to optimize memory usage.	config("spark.sql.inMemoryColumnarStorage.compressed", "true")