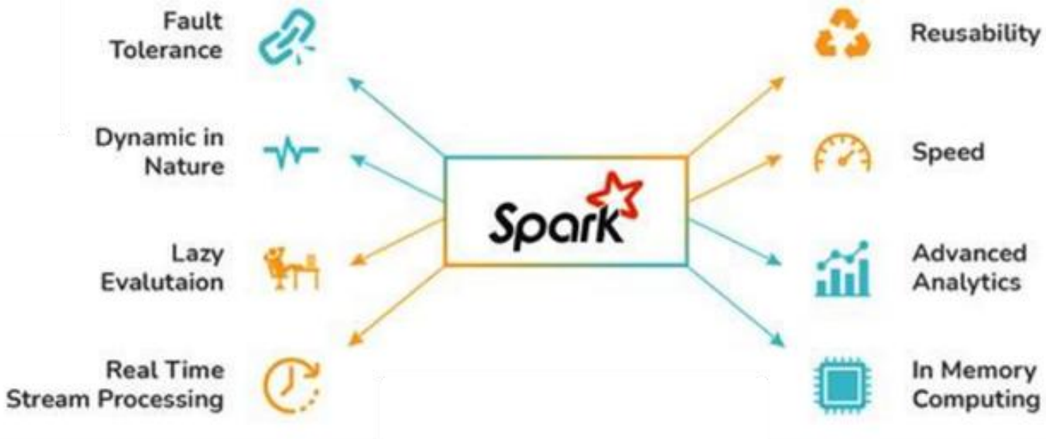
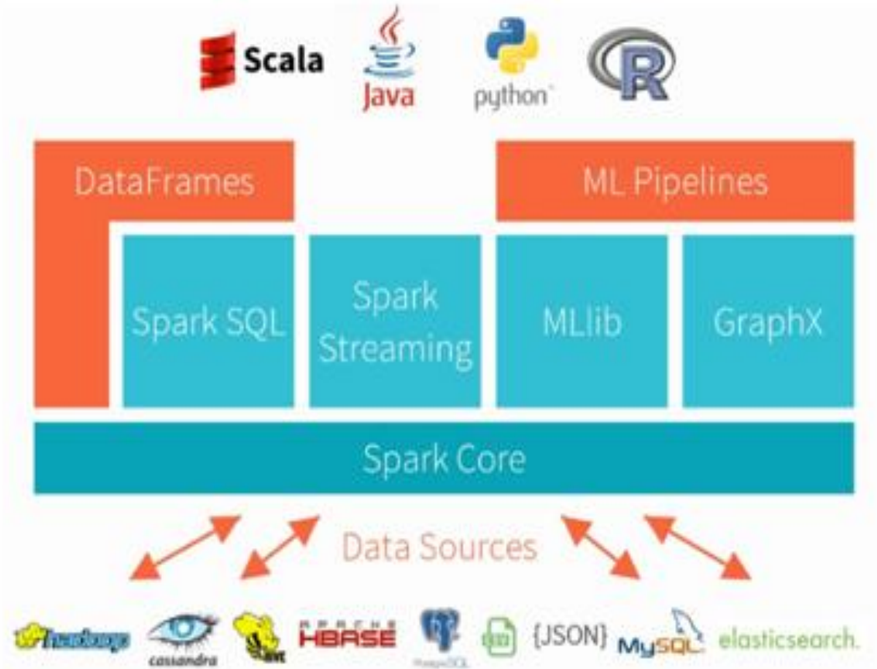
inceptez
Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817

Spark Architecture

About Spark

- ✓ Open source distributed framework to process large volume of data
- ✓ Provides high-level APIs in Scala, Java , Python and R
- ✓ Spark offer different types of data processing and Unified data access
- ✓ Runs on multiple cluster





Spark Installation on Local Machine

Prerequisites for spark:

- Java Version: JDK 17.0.17
- Python Version: 3.10.0
- Spark Version: 3.5.7
- Hadoop Version: 3.3.x

Installation Steps:

Refer below document

Windows:

https://github.com/azurede007/documents-repo/blob/main/Spark_Installation_Guide_Windows.pdf

Mac:

https://github.com/azurede007/documents-repo/blob/main/Spark_Installation_Guide_macOS.pdf

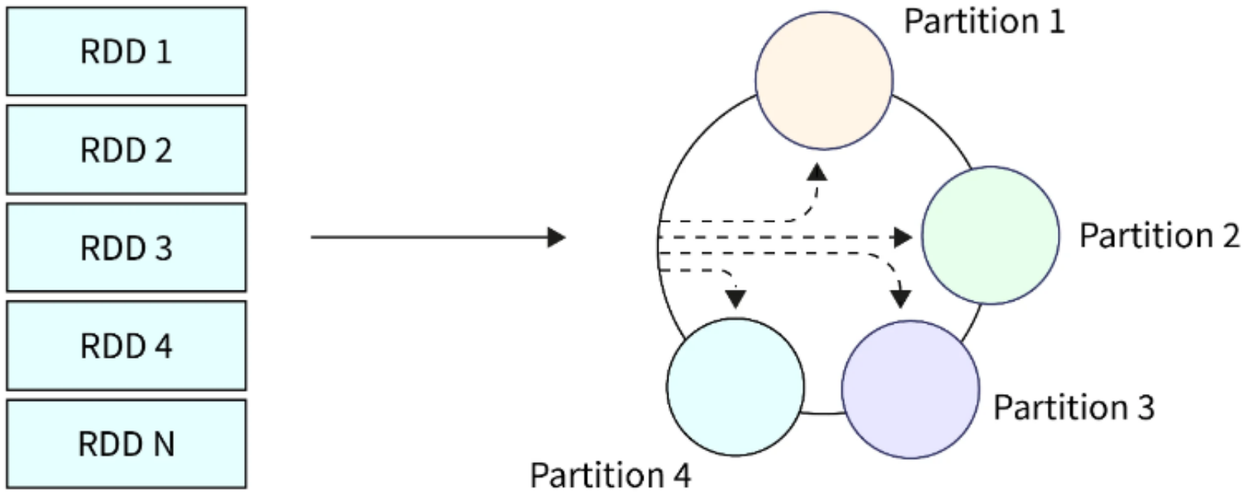
Resilient Distributed Dataset(RDD)

Resilient Distributed Dataset(RDD): Fundamental unit of data in spark

- **Resilient**: Fault tolerant and is capable of rebuilding data on Failure
- **Distributed**: Distributed data among the multiple nodes in a cluster
- **Dataset**: Collection of partitioned data with values

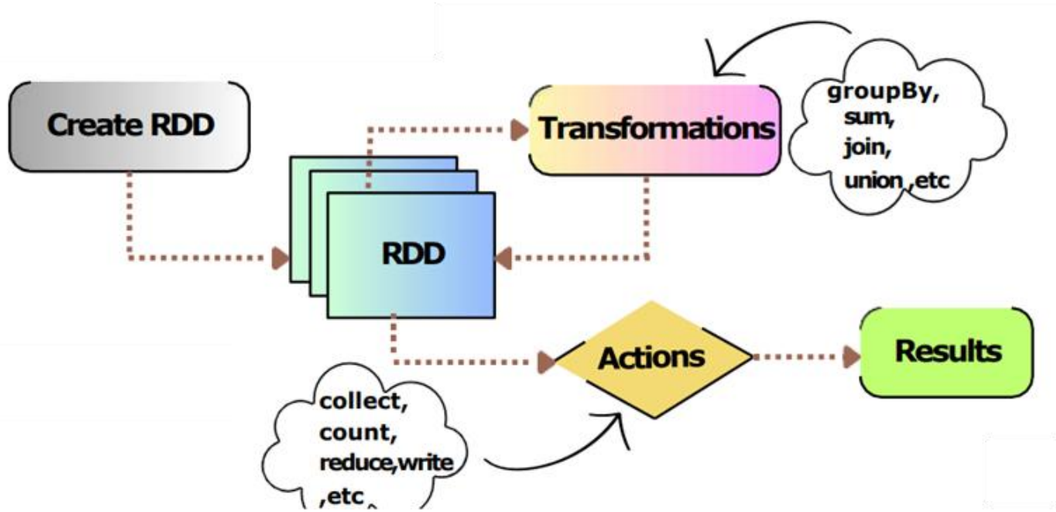
Properties:

- In Memory
- Immutable
- Partitioned
- Parallel



RDD Operations

The RDD provides the two types of operations:



Transformation

- ✓ Transformations create new RDD from existing RDD
- ✓ Transformations are lazily computed
- ✓ Transform in sequence to modify the data in RDD

Action

- ✓ Actions return final results of RDD computations.
- ✓ Actions triggers execution using DAG graph to load the data into original RDD and apply sequence of transformations.
- ✓ Compute a result based on an RDD and either returned or saved to an external storage system (e.g., HDFS).

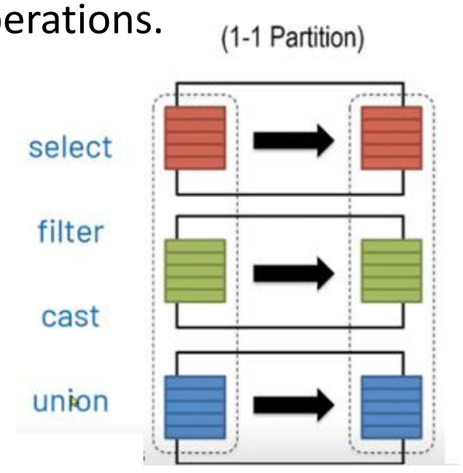
Narrow VS Wide Transformations

Narrow Transformation

- Each input partition contributes to **only one** output partition.
- No data movement (shuffle) happens across the cluster.
- Hence, they are **fast** and **optimized**.

Characteristics:

- No shuffle across executors.
- Operations can be executed **within the same partition**.
- Suitable for **pipeline parallelism**.
- Typically used in **map-side** operations.

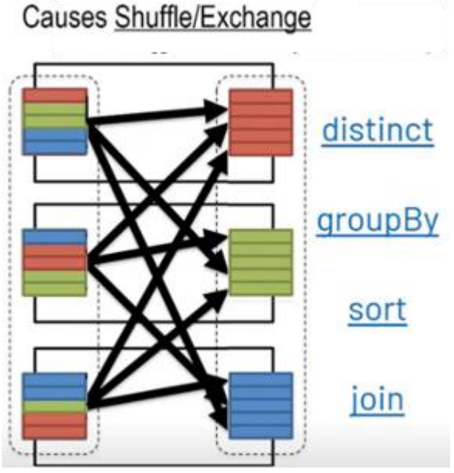


Wide Transformation

- Each input partition contributes to **multiple** output partitions.
- Data is **shuffled** across nodes — involving **network I/O**.
- Hence, **slower** but necessary for grouping, joining, or sorting data.

Characteristics:

- Requires **shuffle** stage.
- Creates **stage boundaries** in DAG.
- Often used in **group-based operations** (aggregation, join, etc.).



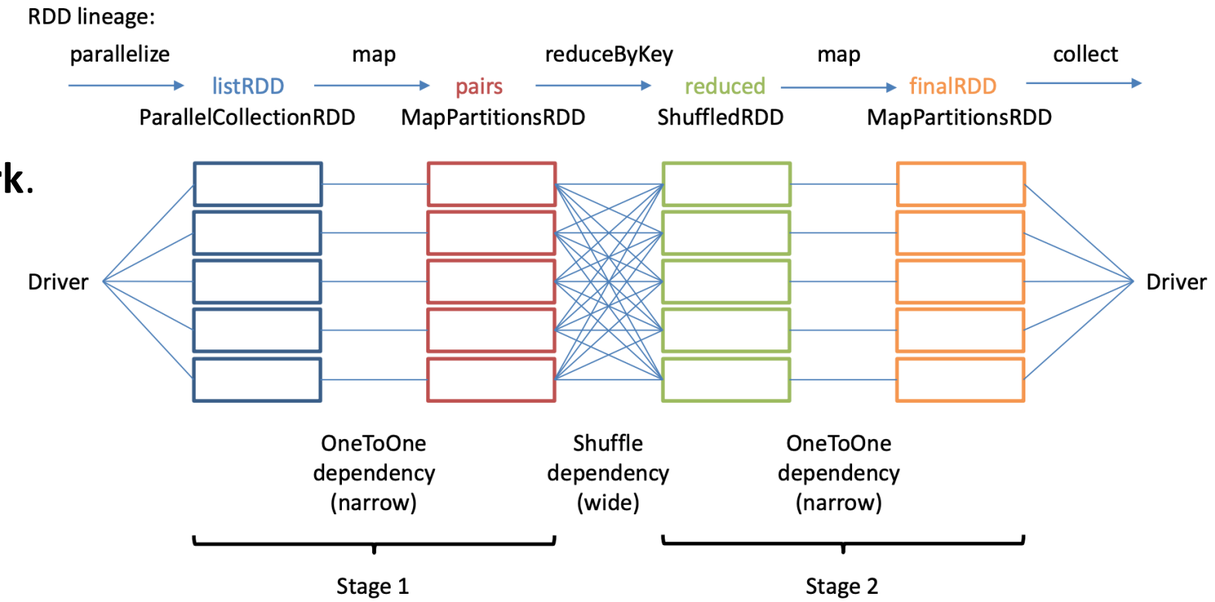
Shuffling in Spark

Shuffling in Spark is the process of redistributing data across partitions, typically caused by wide transformations such as joins and groupBy.

It involves network I/O, disk I/O, and expensive serialization, making it the most costly operation in Spark.

It happens **between stages** and involves:

- Moving data over the network
- Writing intermediate files to disk
- Serialization + deserialization
- Because of these, **shuffling is the most expensive operation in Spark**.



Components in Spark Architecture

Spark Application

- Program built using **Spark APIs** (PySpark, Scala, Java, etc.)
- Runs on a **Spark-compatible cluster**
- Consists of:
  - **Driver** – controls execution
  - **Executors** – perform computation

SparkSession / SparkContext

- Entry point to Spark functionalities
- Connects the application to the Spark cluster
- Provides access to APIs for RDDs, DataFrames, and SQL

Job

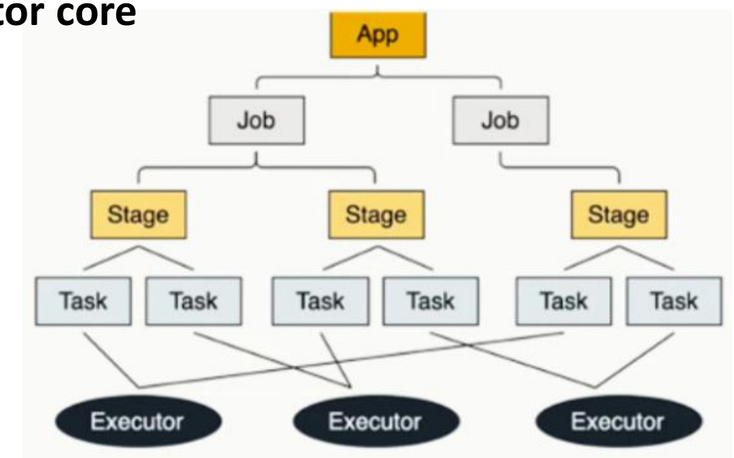
- A **parallel computation** triggered by an **action** (e.g., collect(), save())
- Represents a complete **execution plan**

Stage

- A **subset of a job** that can run in parallel
- Divided based on **narrow** and **wide transformations**
- Executes operations on multiple partitions simultaneously

Task

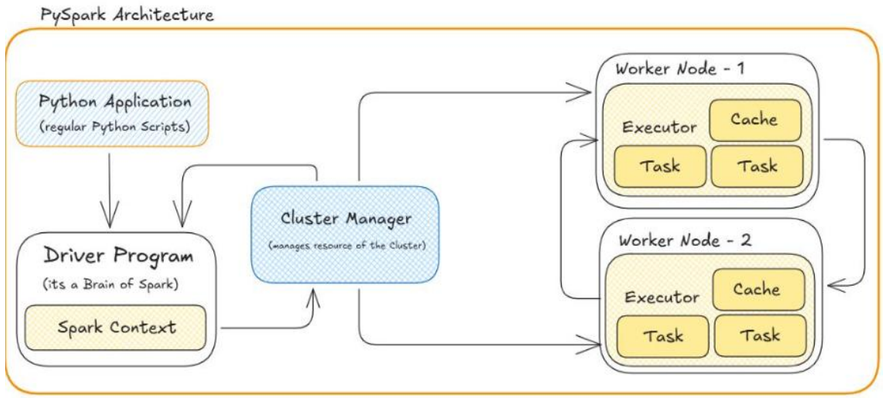
- The **smallest unit of execution** in Spark
- Each task processes **one partition** of data
- Runs on an **executor core**



Driver Component

Overview

- Driver is the central controller of a Spark application that runs main program.
- Creates and manages jobs, stages, tasks, and sends tasks to executors for execution.
- Communicates with the cluster manager and maintains metadata, scheduling, and overall coordination.



Key Responsibilities

Manage SparkContext

- Creates and manages the **SparkContext**, the main entry point to Spark.

Break Down Application

- Splits the application into **tasks** that can run in **parallel** across worker nodes.

Schedule Tasks

- Assigns tasks to executors based on **available resources** and **data locality**.

Monitor Execution

- Tracks task progress and **reschedules failed tasks**
- on different nodes if needed.

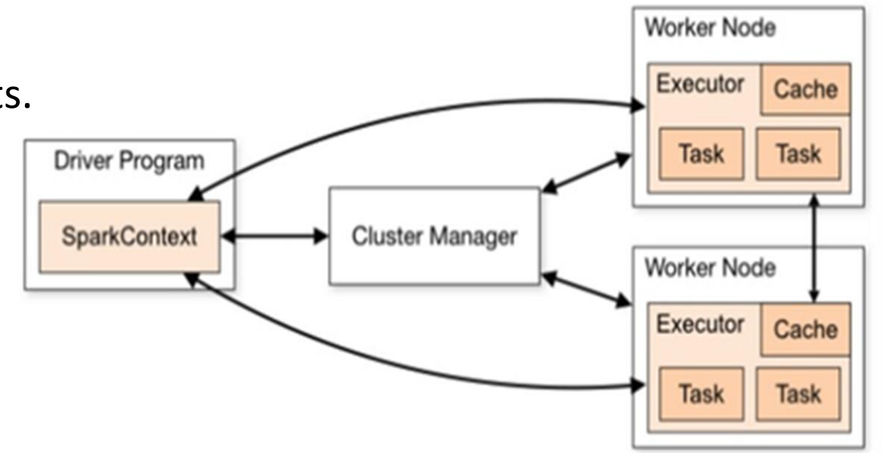
Collect Results

- **Aggregates task outputs** and compiles the **final result** of the application.

Executor Component

Overview

- Executors are worker processes launched on cluster nodes by the Cluster Manager.
- They are responsible for executing tasks assigned by the Driver and returning the results.
- Each Spark application has its own set of executors.



Key Responsibilities

Execute Task

- Perform the actual computations on data partitions.
- Each executor runs multiple tasks in parallel using its cores

Stores Data

- Cache or persist intermediate results in memory or disk for reuse.
- Helps improve performance in iterative and interactive workloads.

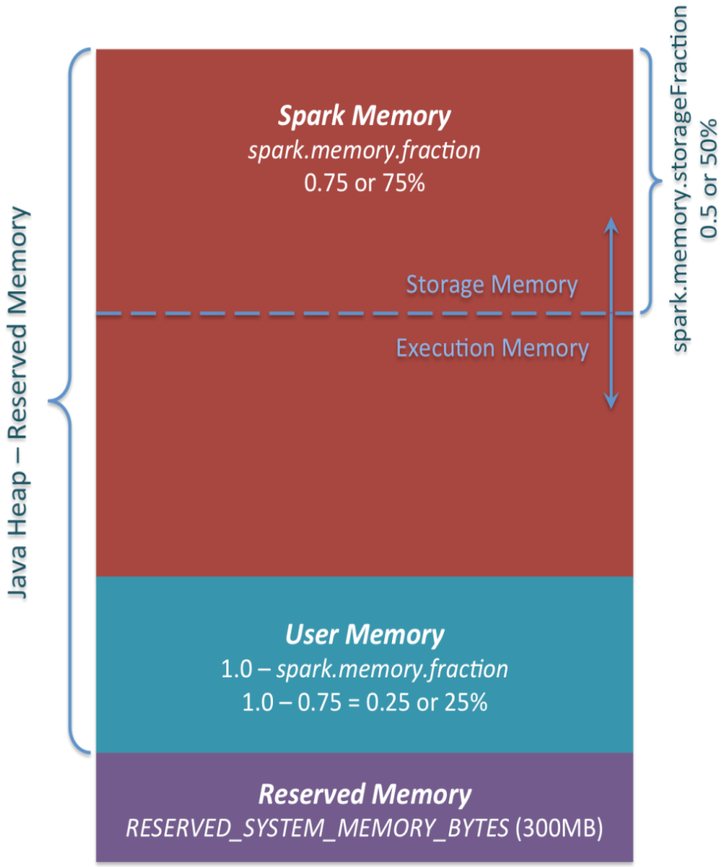
Report Status

- Continuously sends task progress and metrics back to the Driver.
- Helps the Driver track success, failure, and resource usage.

Manage Resources

- Utilizes assigned CPU cores and memory efficiently for task execution.
- Cleans up after tasks complete.

Executor - Memory Management



1. Reserved Memory (~300 MB)

- Fixed memory Spark reserves internally.
- Not available for tasks.
- For running executors

2. Usable Memory = Executor Memory – Reserved Memory

This is then split dynamically between:

- **Execution Memory**

  Used for:

  - Holding **intermediate shuffle data**
  - Sorting, Joins
  - Aggregations

- **Storage Memory**

  Used for:

  - **Caching/Persisting** RDD/DataFrames
  - Broadcast variables
  - Serialized data blocks
  - Execution and Storage memory dynamically borrow from each other based on need.

3. User Memory (~25%)

Used for:

- Python/JVM Objects creation
- Custom data structures
- UDF Operations

Executor - Memory Management

Memory Type	1 GB Executor (1024 MB)	Calculation (1 GB)	10 GB Executor (10240 MB)	Calculation (10 GB)
Reserved Memory	**300 MB**	Fixed value	**300 MB**	Fixed value
Usable Memory	**724 MB**	1024 − 300	**9920 MB**	10240 − 300
User Memory (25%)	**181 MB**	724 × 0.25	**2480 MB (2.48 GB)**	9920 × 0.25
Unified Memory (75%)	**543 MB**	724 × 0.75	**7440 MB (7.44 GB)**	9920 × 0.75
Execution Memory (50% of Unified)	**271 MB**	543 ÷ 2	**3720 MB (3.72 GB)**	7440 ÷ 2
Storage Memory (50% of Unified)	**271 MB**	543 ÷ 2	**3720 MB (3.72 GB)**	7440 ÷ 2

Syntax:

--executor-memory <Size>G

Executor - Cores

- Each **executor** is assigned a certain number of CPU cores.
  These cores determine how many tasks it can run in parallel.
- Each core = 1 parallel task inside the executor.
- If an executor has 5 cores, it can run 5 tasks simultaneously.
- More cores → higher parallelism per executor.

Syntax:

--executor-cores <num_of_cores>

Note: *Total cores available from the cluster determine how many executors you can get.*

Workload	Recommended Executor Cores
General workloads	**4–5 cores**
Heavy shuffles / joins	**3–4 cores**
Streaming workloads	**1–2 cores**
MLlib / CPU-heavy tasks	**4–6 cores**

Example Cluster:

- Total cluster cores: **32**
- Set executor cores = **4**

Number of executors = 32 / 4 = 8 executors

Tasks in parallel = 8 executors × 4 cores = 32 tasks

So the job can run **32 parallel tasks**.

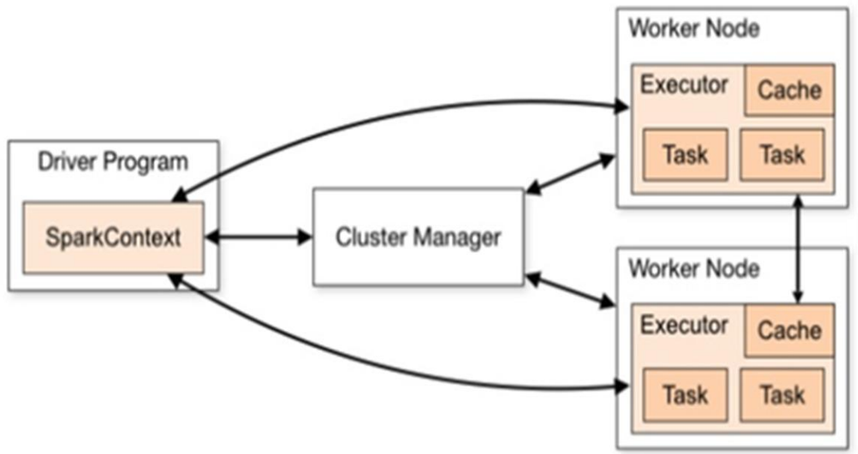
Master Node

The **Master Node** in Apache Spark is the central coordinator of the entire Spark cluster.

It performs *cluster management*, *resource allocation*, and *job scheduling*.

Key Responsibilities:

- Cluster resource management
- Job Scheduling & Coordination
- Resource Allocation
- Worker health monitoring & failure recovery
- Application Lifecycle Management
- Communication between driver and workers

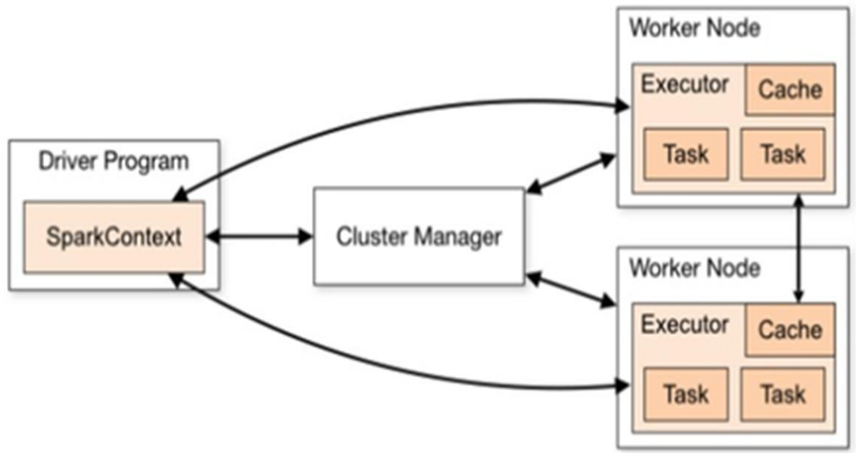


Worker Node

The **Worker Node** in Apache Spark is responsible for **executing tasks**, **hosting executors**, and **providing compute resources** for the cluster.

Key Responsibilities:

- Launches and manages executors (JVM processes running tasks).
- Executes assigned tasks (transformations and actions).
- Caches or persists RDD/DataFrame partitions in memory or disk for reuse.
- Handles shuffle read/write during wide transformations.
- Sends status updates and heartbeat messages to the master node.
- Fault Tolerance Support
- Local File System Management



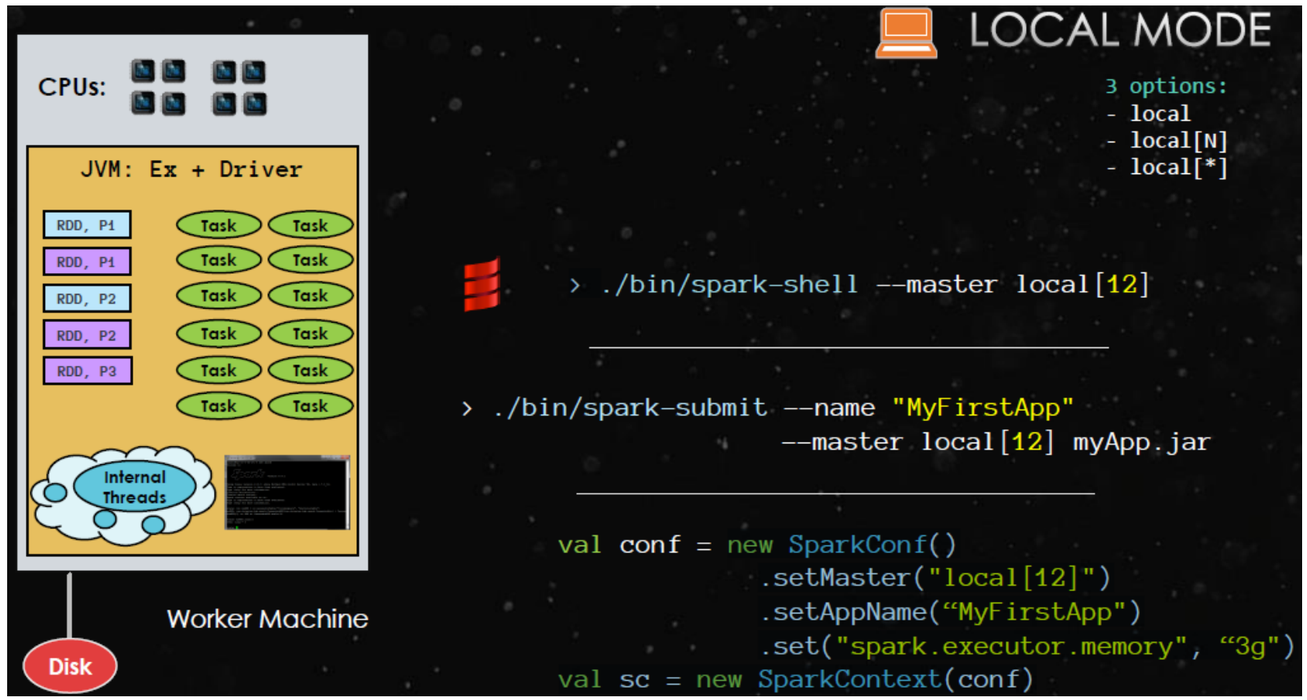
Spark Local Mode

- Default Mode
- Does not require any resource manager
- Partitions are generally equal to number of CPU's
- Used generally for testing

pyspark
pyspark --master local
pyspark --master local[n]
pyspark --master local[*] (Default)

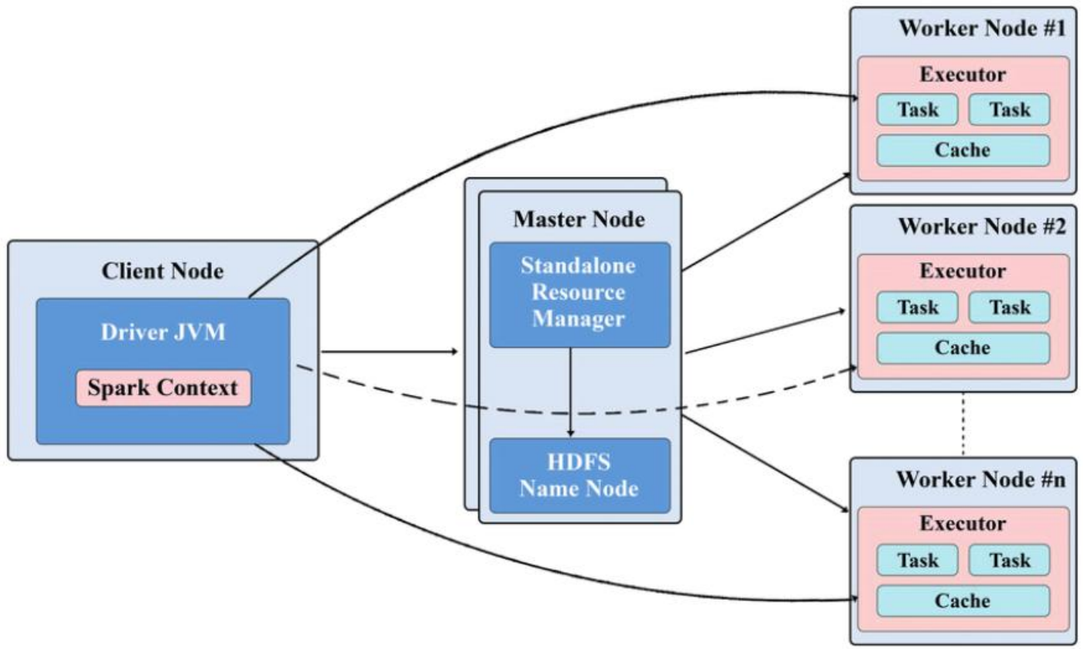
Check spark is running in local:
sc.isLocal or sc.master

sc.defaultParallelism
df.rdd.getNumPartitions()



Spark Standalone Mode

- ✓ Spark distribution comes with its own resource manager
- ✓ There are two daemons (Master and Workers)
- ✓ Spark Master manages the resource



Start Master Node:

cd %SPARK_HOME%\bin

spark-class.cmd org.apache.spark.deploy.master.Master

Start Worker Node:

cd %SPARK_HOME%\bin

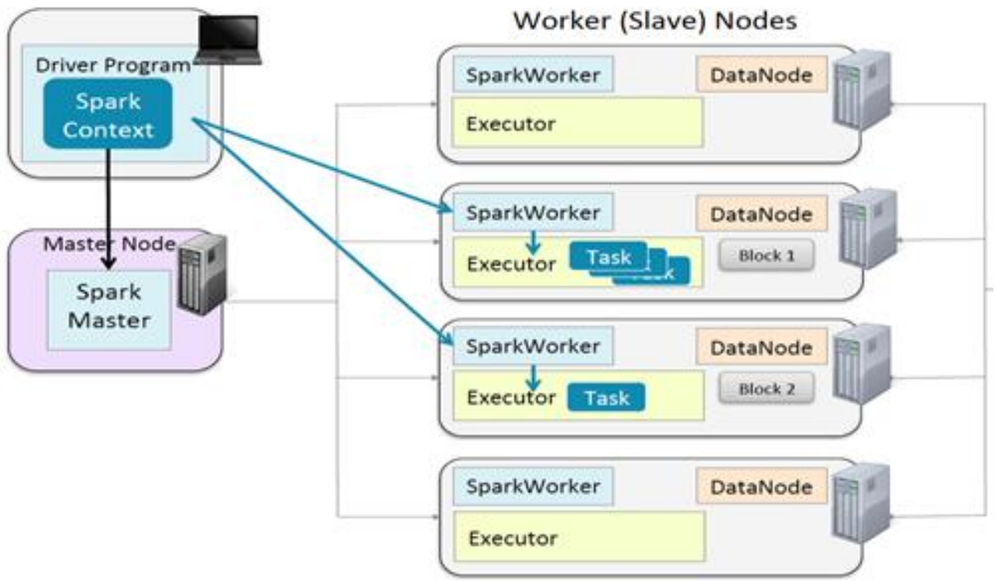
spark-class.cmd org.apache.spark.deploy.worker.Worker spark://localhost :7077

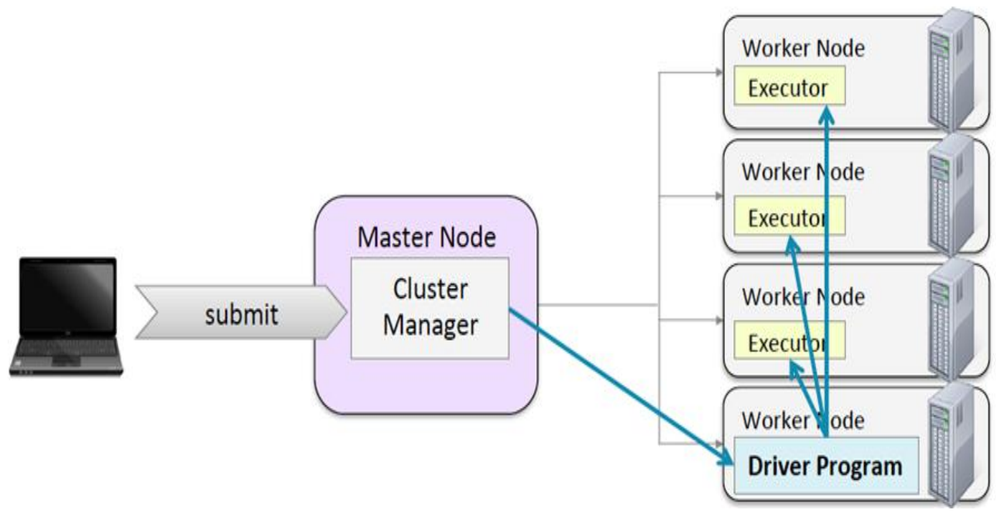
spark-shell –spark://localhost:7077

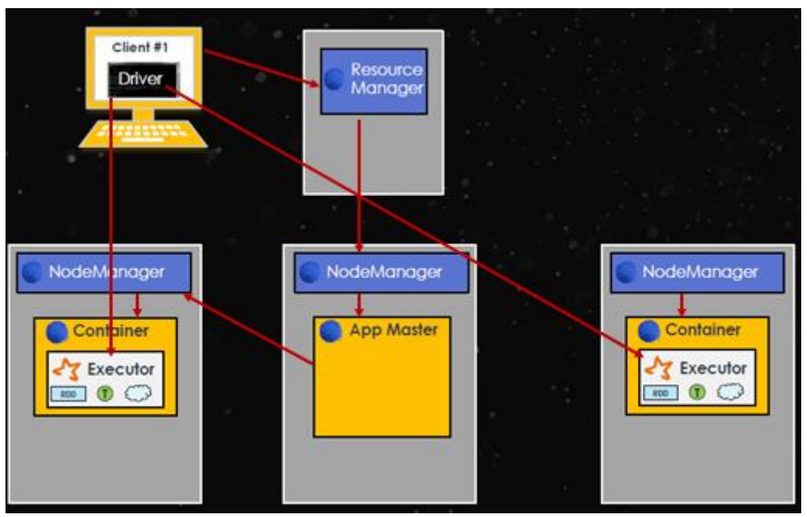
Start UI:

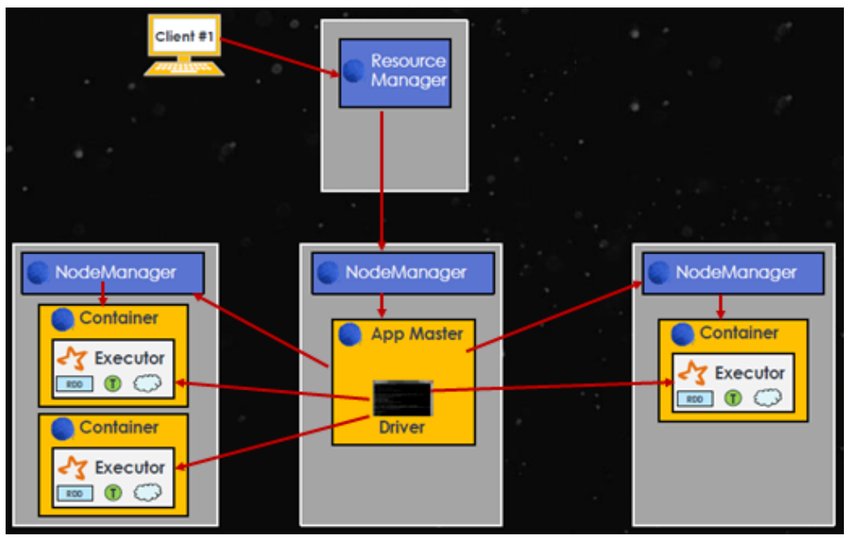
http://localhost:8080/index.html

Client Mode and Cluster Mode









Spark YARN

YARN (Yet Another Resource Negotiator) is the **cluster manager** used in **Hadoop ecosystems**.

When we run **Spark on YARN**, Spark uses YARN to **request resources** (CPU, memory) and **run executors** on Hadoop cluster.

So instead of Spark's standalone master–worker setup, **YARN becomes the cluster manager**.

Spark on YARN – Components

1. Client (Driver Program)

1. Submits the Spark application to YARN.
2. Can run **locally** or **inside YARN** depending on the mode.

2. ResourceManager (RM)

1. Core YARN service.
2. Allocates containers (resources like CPU and memory) to applications.

4. NodeManager (NM)

1. Runs on each node.
2. Launches and monitors containers (executors).

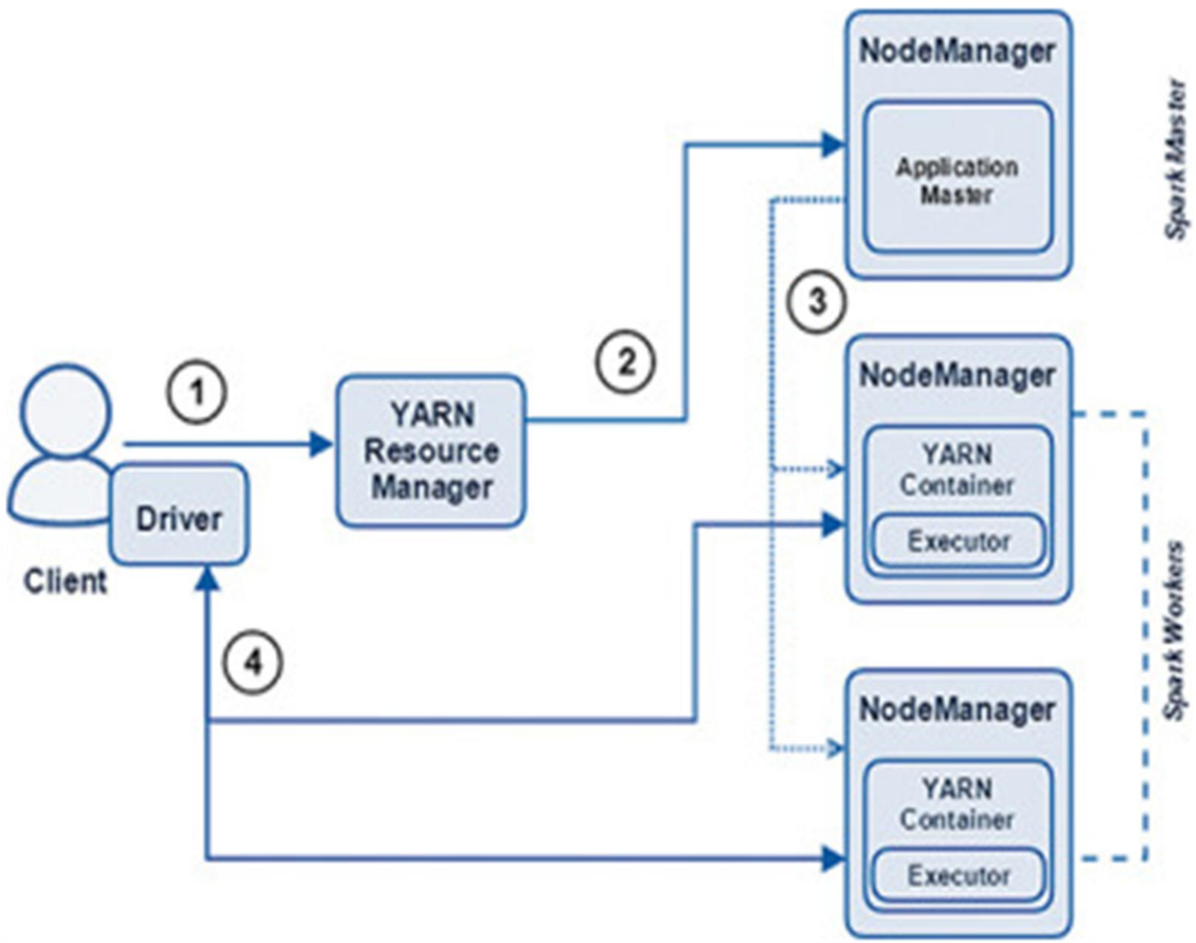
5. ApplicationMaster (AM)

1. YARN component launched per Spark application.
2. Negotiates resources with RM and monitors executors.

6. Executors

1. Run inside YARN containers.
2. Execute Spark tasks and store data in memory.

Spark YARN Client Mode



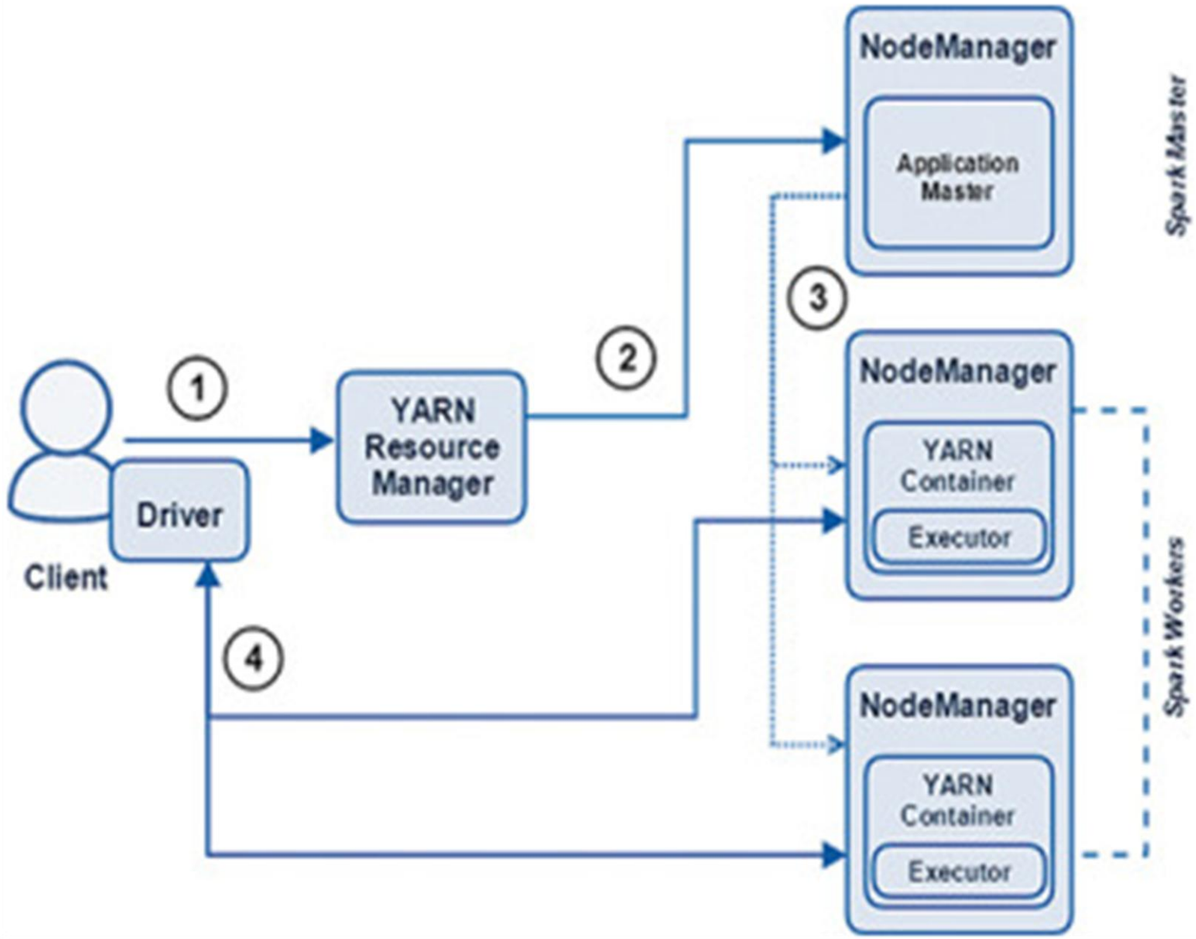
1. The client submits a Spark application to the Cluster Manager (the YARN ResourceManager). The Driver process, SparkSession, and SparkContext are created and run on the client.

2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application.

3. The ApplicationMaster requests containers to be used for Executors from the ResourceManager. With the containers assigned, the Executors spawn.

4. The Driver, located on the client, then communicates with the Executors to marshal processing of tasks and stages of the Spark program. The Driver returns the progress, results, and status to the client.

Spark YARN – Cluster Mode



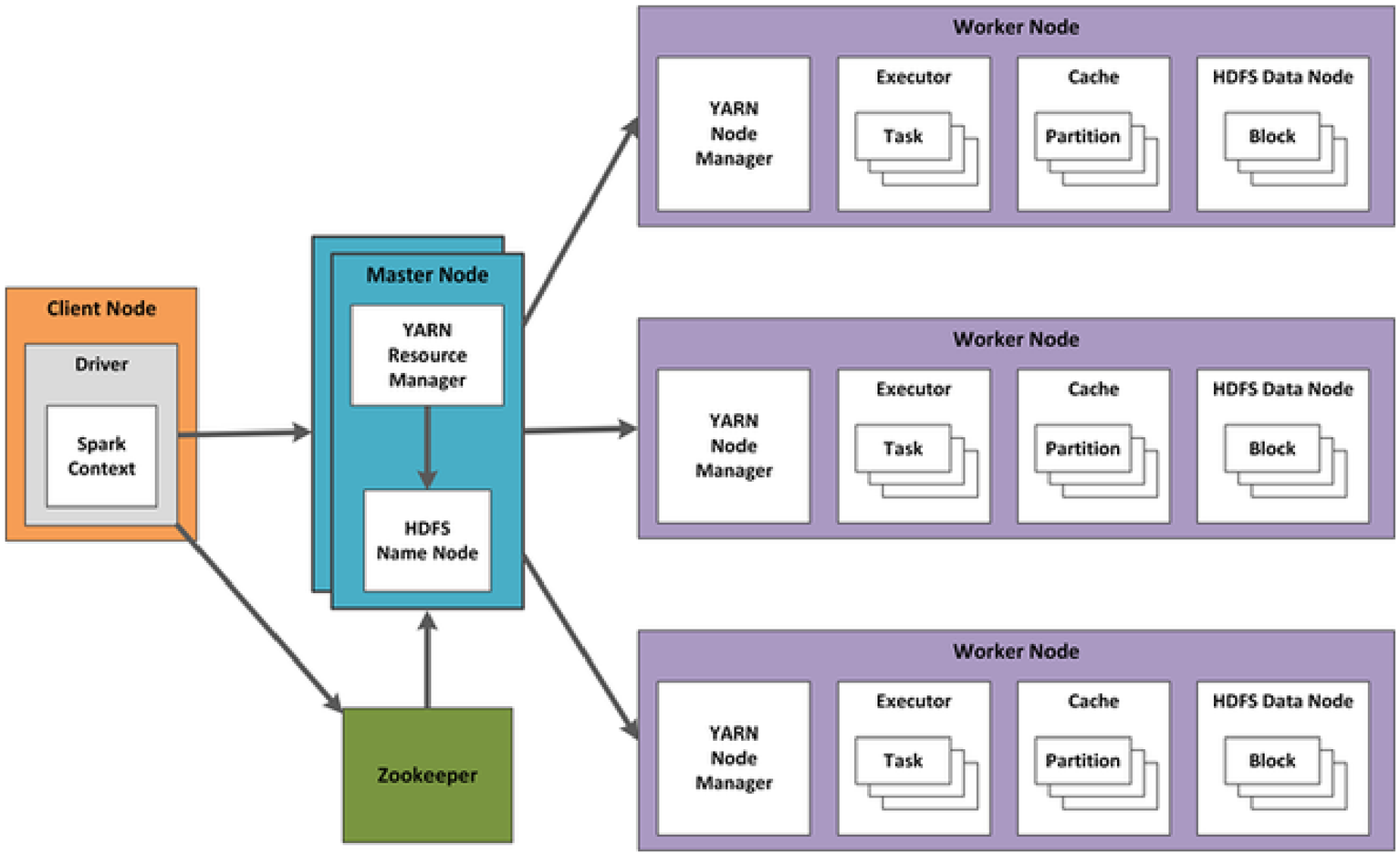
1. The client, a user process that invokes spark-submit, submits a Spark application to the Cluster Manager (the YARN ResourceManager).

2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application. The Driver process is created on the same cluster node.

3. The ApplicationMaster requests containers for Executors from the ResourceManager. Executors are spawned within the containers allocated to the ApplicationMaster by the ResourceManager. The Driver then communicates with the Executors to marshal processing of tasks and stages of the Spark program.

4. The Driver, running on a node in the cluster, returns progress, results, and status to the client.

Spark YARN


Worker Node
YARN Node Manager
Executor
Task
Cache
Partition
HDFS Data Node
Block
Master Node
YARN Resource Manager
HDFS Name Node
Client Node
Driver
Spark Context
Zookeeper
Worker Node
YARN Node Manager
Executor
Task
Cache
Partition
HDFS Data Node
Block
Worker Node
YARN Node Manager
Executor
Task
Cache
Partition
HDFS Data Node
Block

Spark Submit – Different Clusters

```
# Run application locally on 8 cores
./bin/spark-submit --master local[8] /path/to/examples.py  100
```

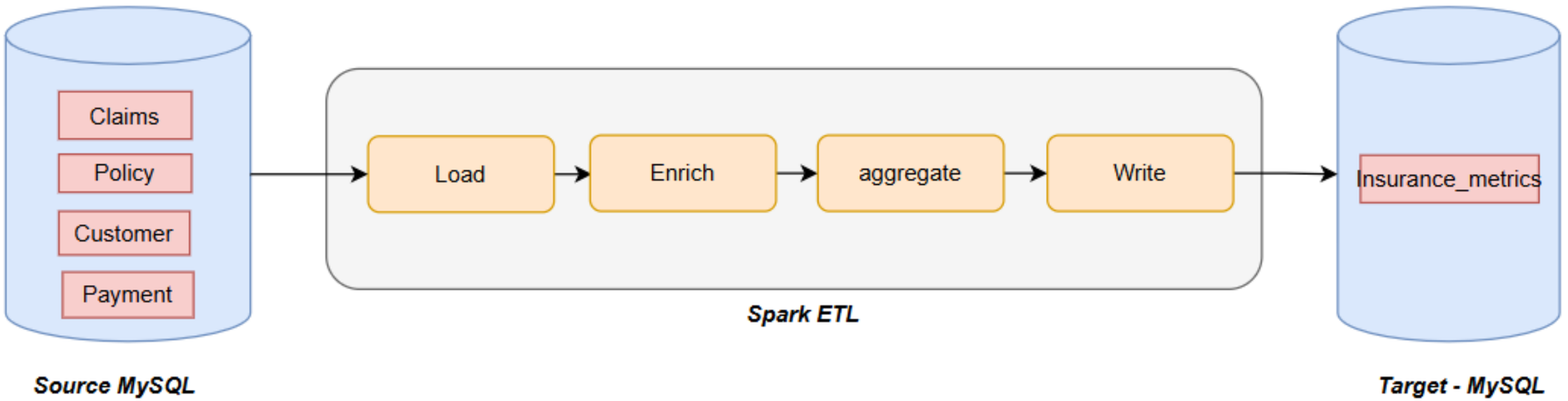
```
# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
--master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.py \
  1000
```

```
# Run on a Spark standalone cluster in cluster deploy
mode with supervise
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.py \
  1000
```

```
# Run on a YARN cluster
./bin/spark-submit \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory  20G \
  --num-executors  50 \
  /path/to/examples.py \
  1000
```

```
# Run a Python application on a Spark standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

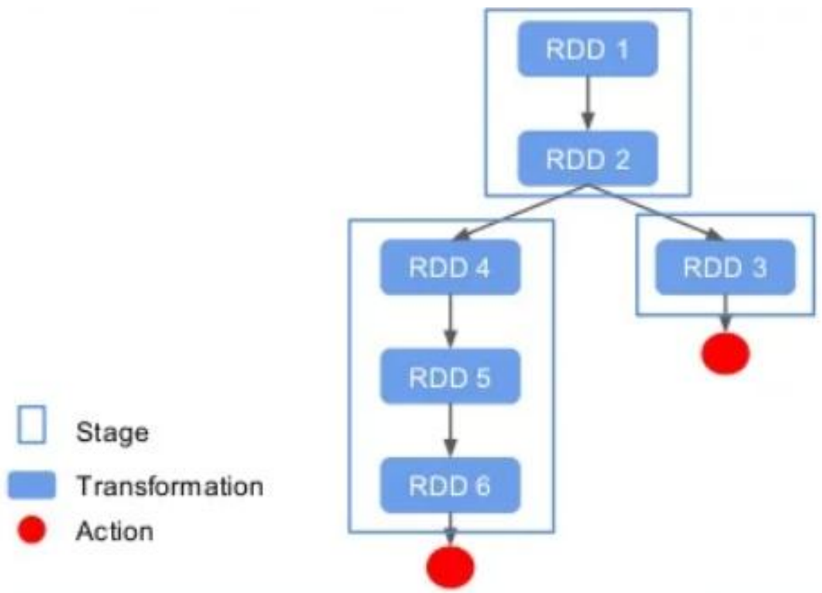
Insurance ETL - Project

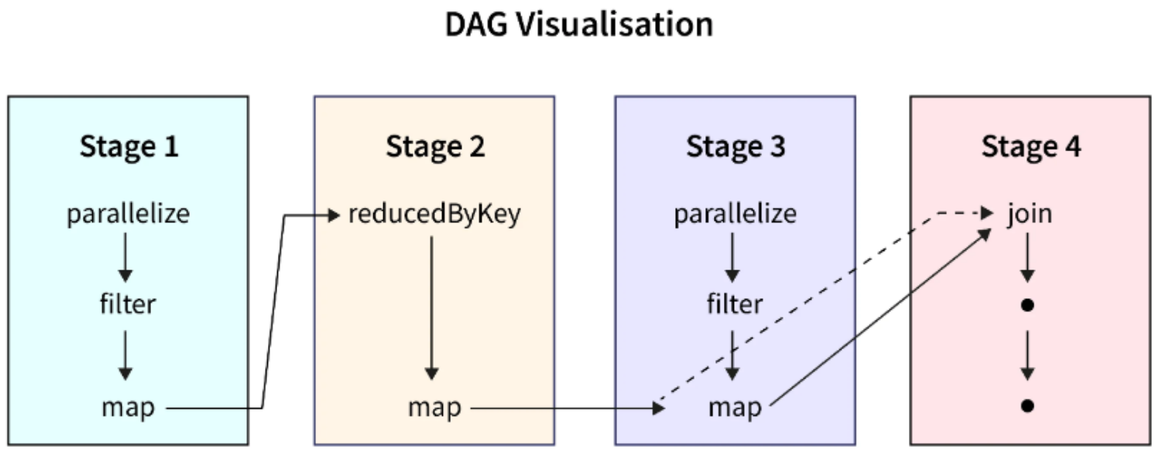


THANK YOU

Directed Acyclic Graph (DAG)

- Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data.
- DAG in Spark supports cyclic data Flow. Every Spark job creates a DAG of task stages that will be executed on the cluster.
- In a Spark DAG, there are consecutive computation stages that optimize the execution plan.
- Achieve Fault-tolerance in Spark with DAG.





Distributed System

Distributed System

- A Distributed System is a network of independent computers (nodes) that work together as a single system to achieve a common goal.
- Each node in the system has its own CPU, memory, and storage, but they communicate and coordinate over a network to perform large-scale computations or services.

Key Characteristics

- **Multiple nodes** work together on a common problem.
- **Resource sharing** – CPU, memory, storage, and data.
- **Fault tolerance** – if one node fails, others continue processing.
- **Scalability** – easily add or remove nodes based on demand.
- **Transparency** – the user sees it as one unified system.

Category	Technology / Platform
Distributed Storage	HDFS (Hadoop Distributed File System) Amazon S3 Google File System (GFS) Azure Data Lake Storage (ADLS) Cassandra / HBase
Distributed Processing	Apache Spark Apache Flink Apache Hadoop (MapReduce) Google BigQuery Databricks / Azure Synapse NoSQL database

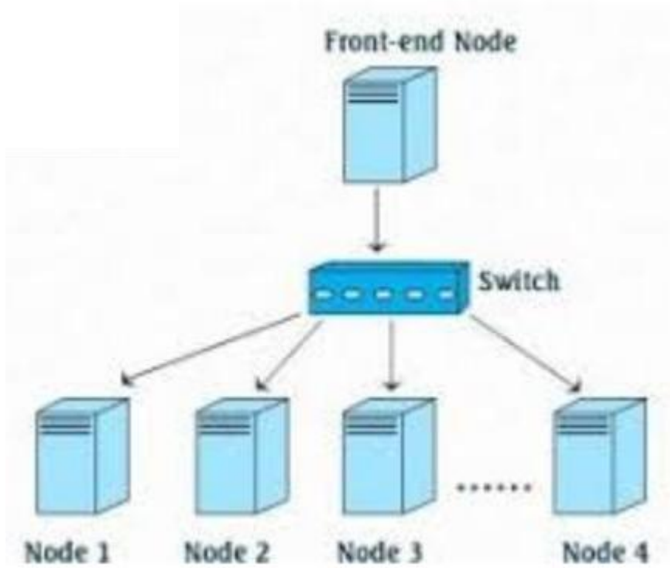
Cluster

Distributed System

- A cluster in Apache Spark is a collection of interconnected machines (nodes) that work together to execute Spark applications in a distributed and parallel manner.
- It provides the computing power, memory, and storage required to process large-scale data efficiently.

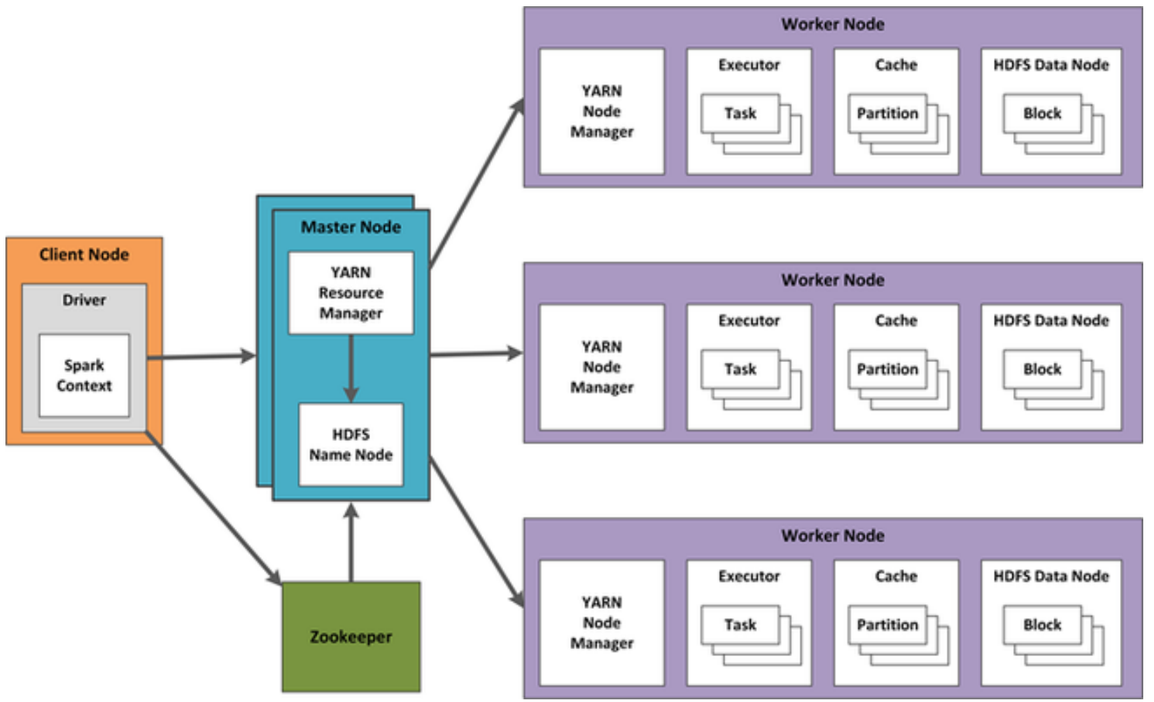
Key Characteristics

- **Multiple nodes** work together on a common problem.
- **Resource sharing** – CPU, memory, storage, and data.
- **Fault tolerance** – if one node fails, others continue processing.
- **Scalability** – easily add or remove nodes based on demand.
- **Transparency** – the user sees it as one unified system.

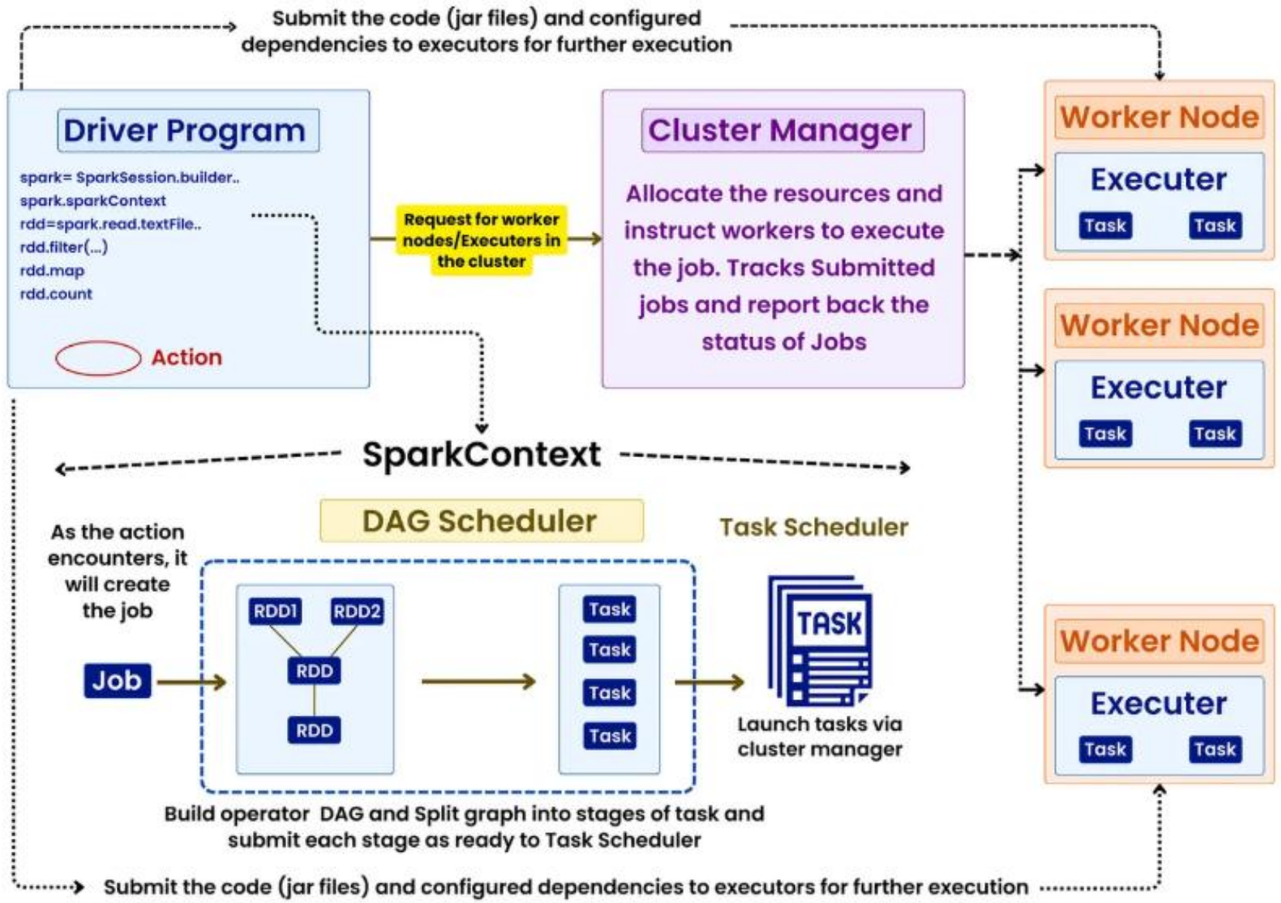


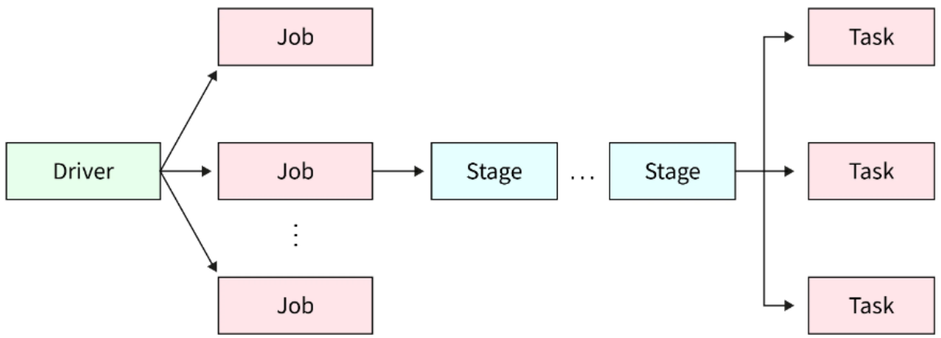
YARN

- ✓ **YARN (Yet Another Resource Negotiator)** is a **cluster manager** used by Spark to request resources (CPU, memory), launch executors, manage jobs, and scale applications across a Hadoop cluster.
- ✓ YARN provides:
  - **Resource allocation** (memory, cores for executors)
  - **Scheduling** (which job runs first)
  - **Container management** (launch/restart executors)
  - **Monitoring** (resource usage, logs, failures)
  - **Multi-user cluster sharing**



Driver Component





Spark Sample Program

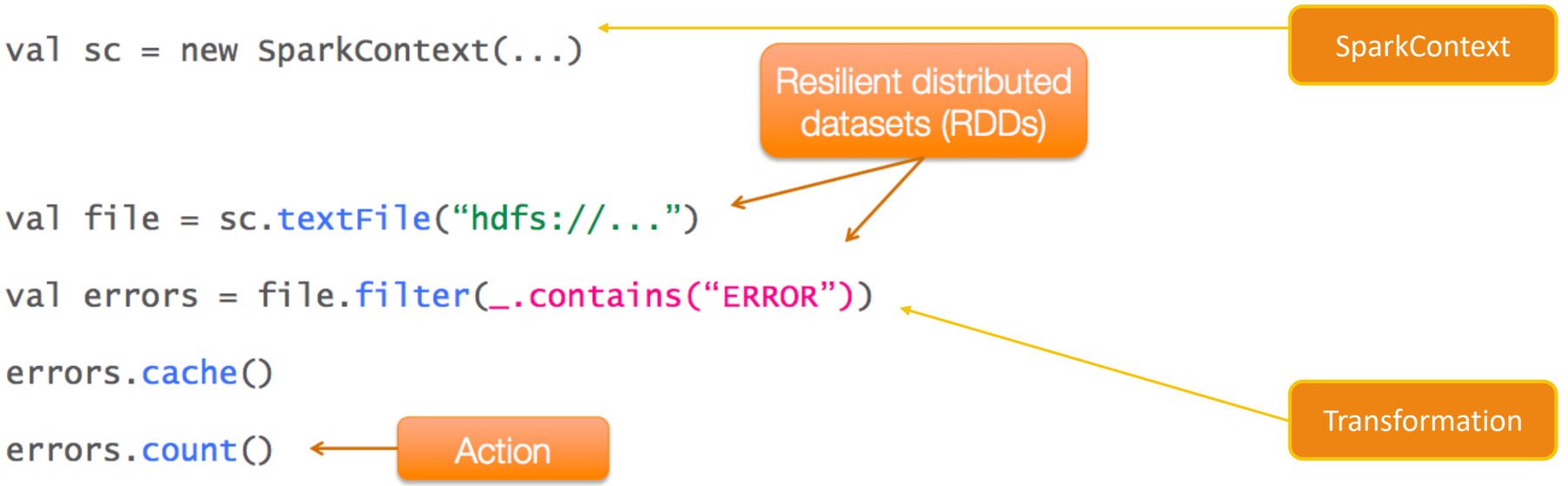
```
val sc = new SparkContext(...)

val file = sc.textFile("hdfs://...")

val errors = file.filter(_.contains("ERROR"))

errors.cache()

errors.count()
```



Spark Terminologies

Driver Program: The process to start the execution (main() function)

Spark Context: Constructor created that tells how to access the cluster.

Application : User Program and its dependencies bundled as jar

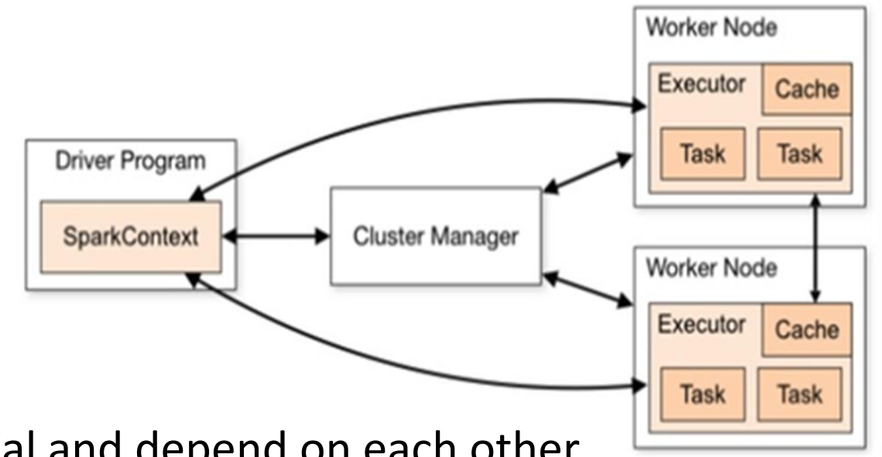
Job: Consists multiple tasks, Created based on a Action

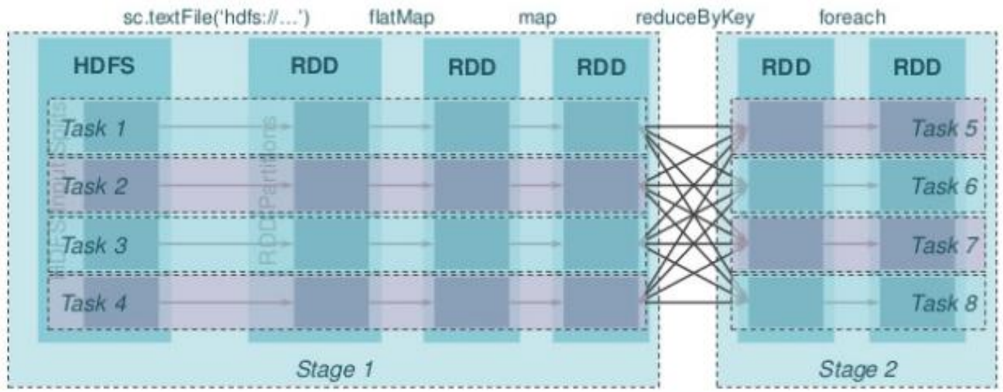
Stage : Each Job is divided into a smaller set of tasks called Stages that is sequential and depend on each other

Task: A unit of work that will be sent to executor.

Partitions: Data unit that will be handled parallel, same as blocks in HDFS

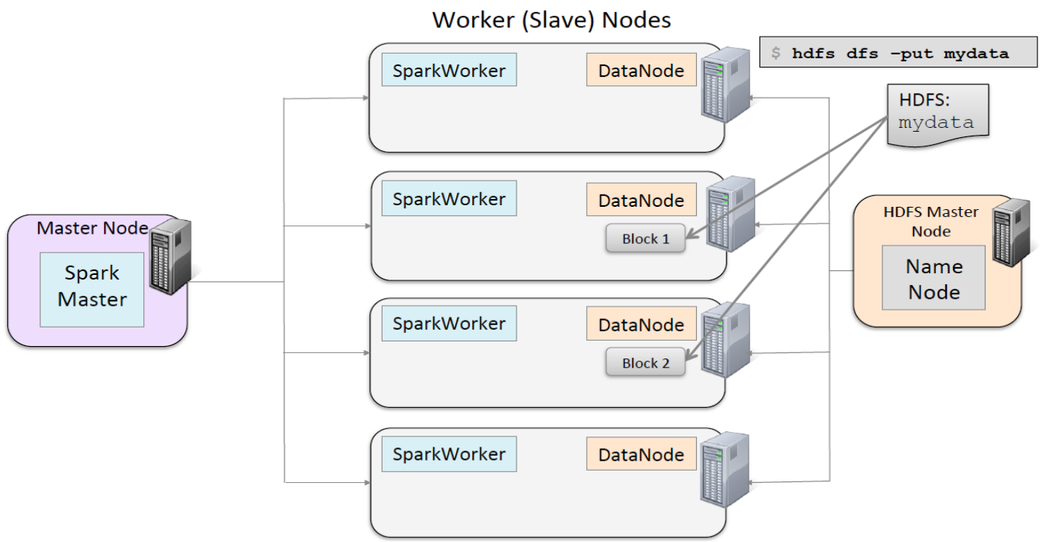
Executor : Process launched on a worker node,
that runs the Tasks Keep data in memory or disk storage



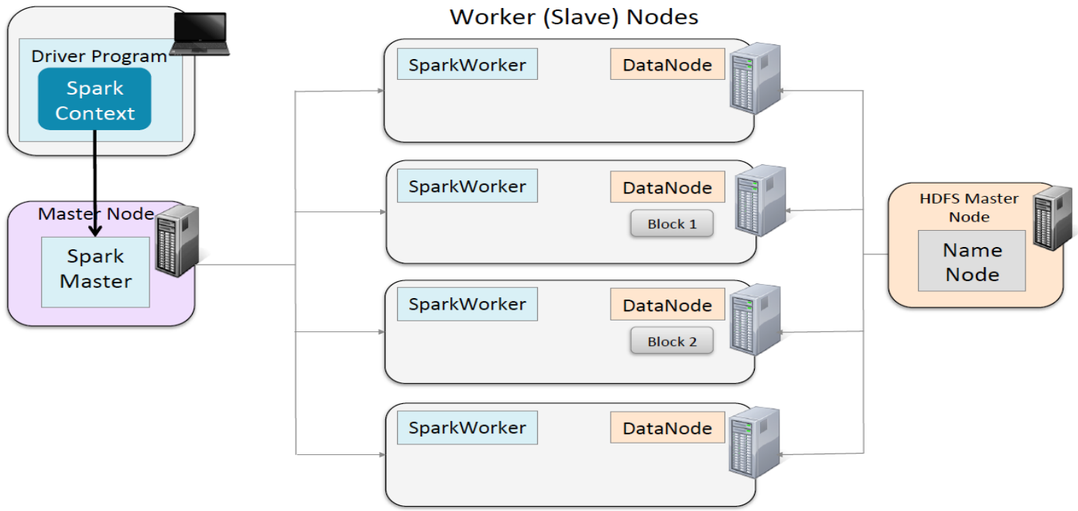


Spark Standalone Mode

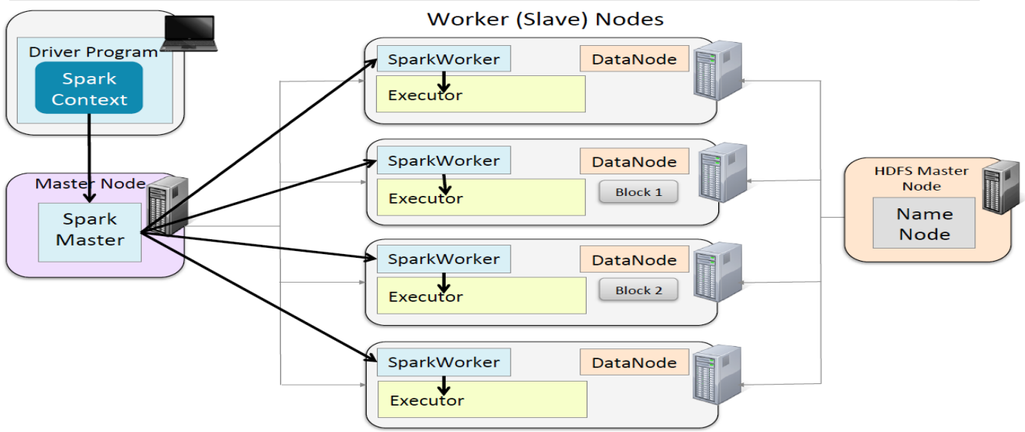
Running Spark on a Standalone Cluster (2)



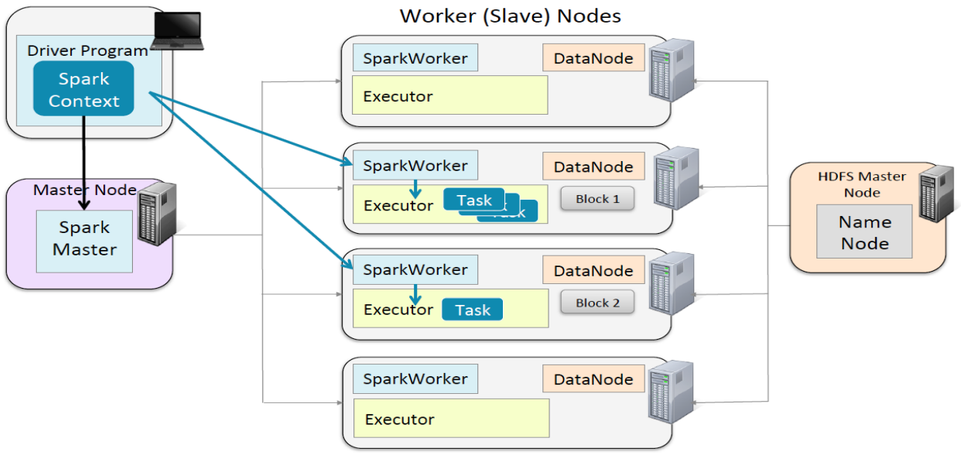
Running Spark on a Standalone Cluster (3)



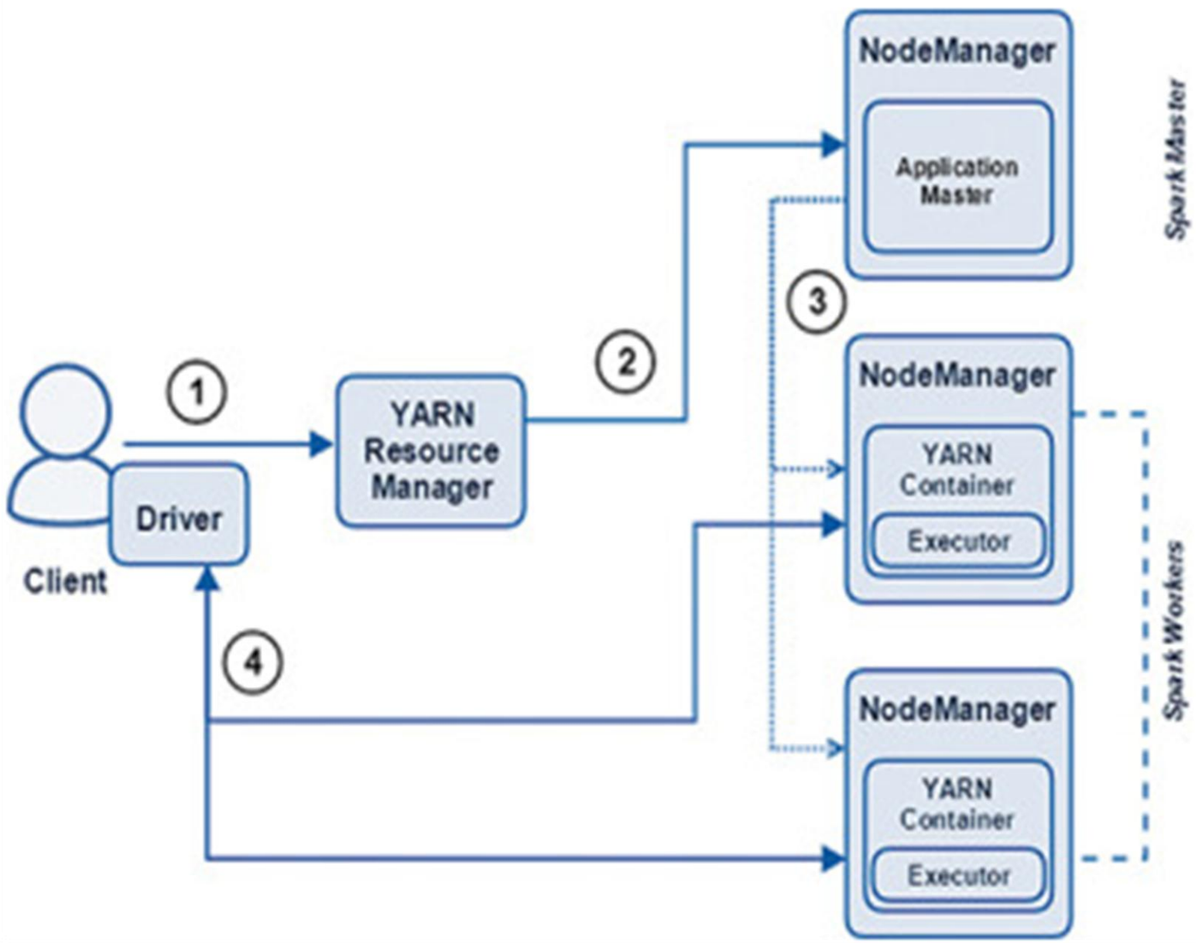
Running Spark on a Standalone Cluster (4)



Running Spark on a Standalone Cluster (5)



Spark YARN Client Mode



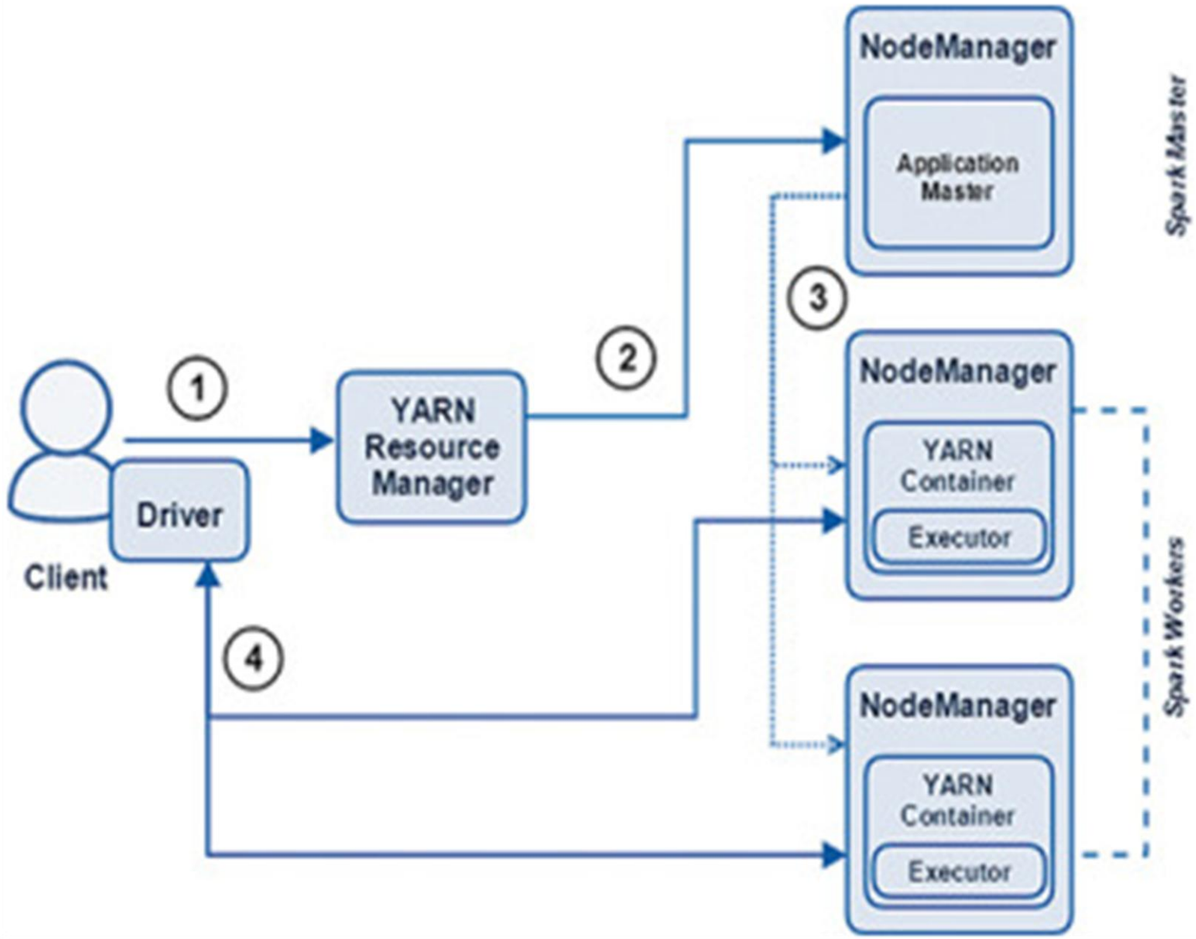
1. The client submits a Spark application to the Cluster Manager (the YARN ResourceManager). The Driver process, SparkSession, and SparkContext are created and run on the client.

2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application.

3. The ApplicationMaster requests containers to be used for Executors from the ResourceManager. With the containers assigned, the Executors spawn.

4. The Driver, located on the client, then communicates with the Executors to marshal processing of tasks and stages of the Spark program. The Driver returns the progress, results, and status to the client.

Spark YARN – Cluster Mode



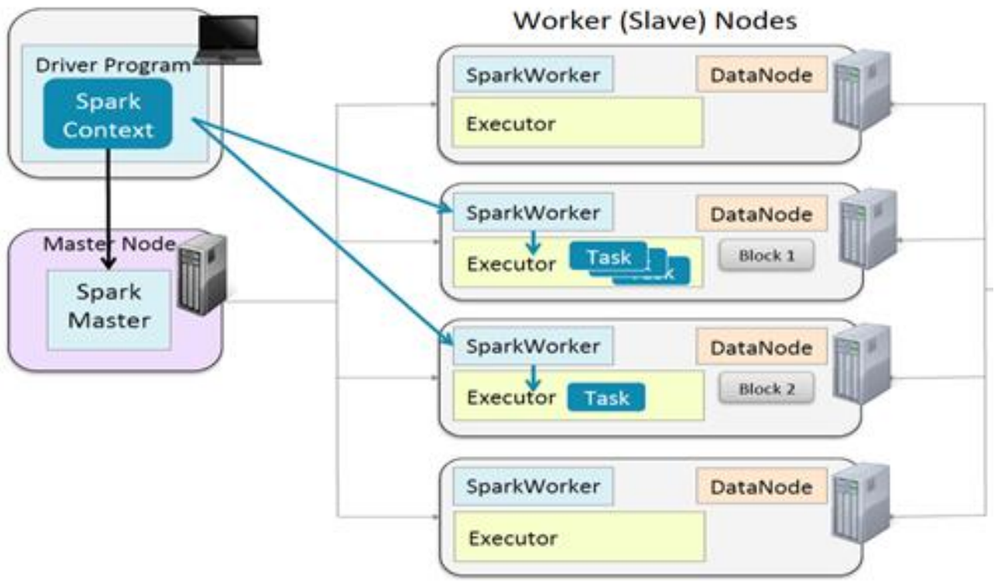
1. The client, a user process that invokes spark-submit, submits a Spark application to the Cluster Manager (the YARN ResourceManager).

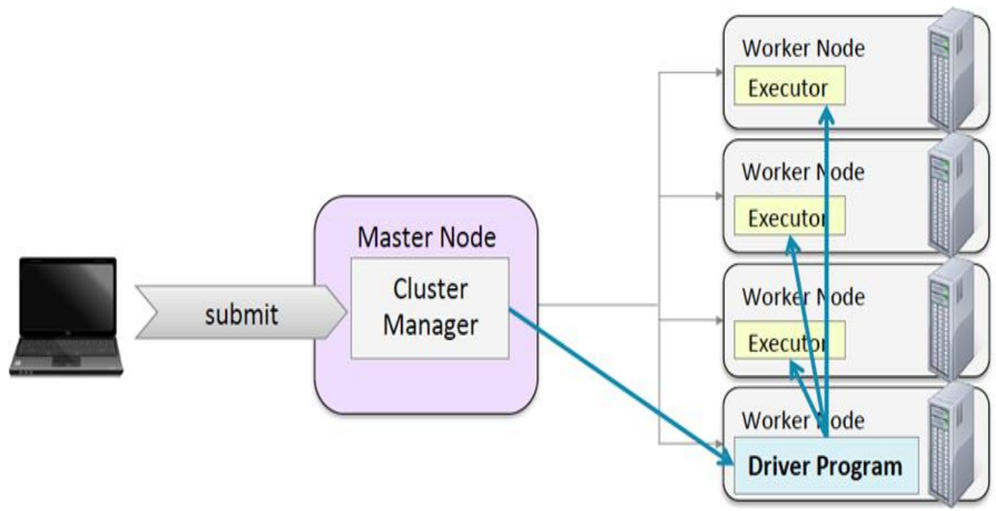
2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application. The Driver process is created on the same cluster node.

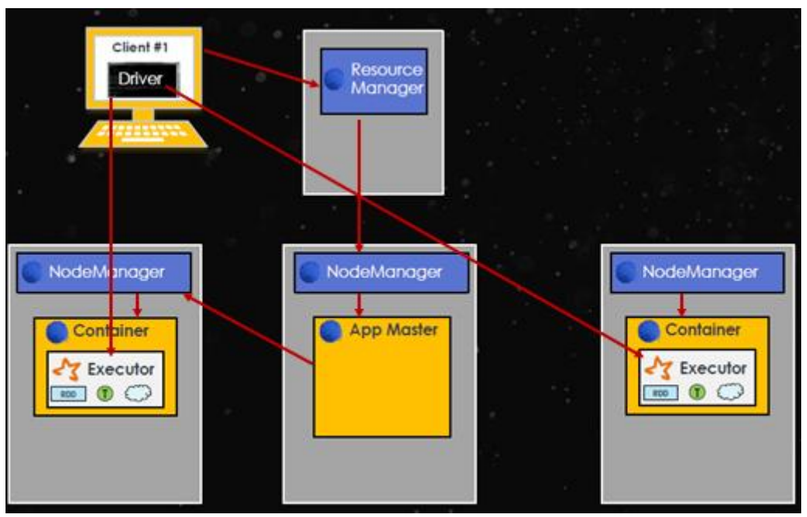
3. The ApplicationMaster requests containers for Executors from the ResourceManager. Executors are spawned within the containers allocated to the ApplicationMaster by the ResourceManager. The Driver then communicates with the Executors to marshal processing of tasks and stages of the Spark program.

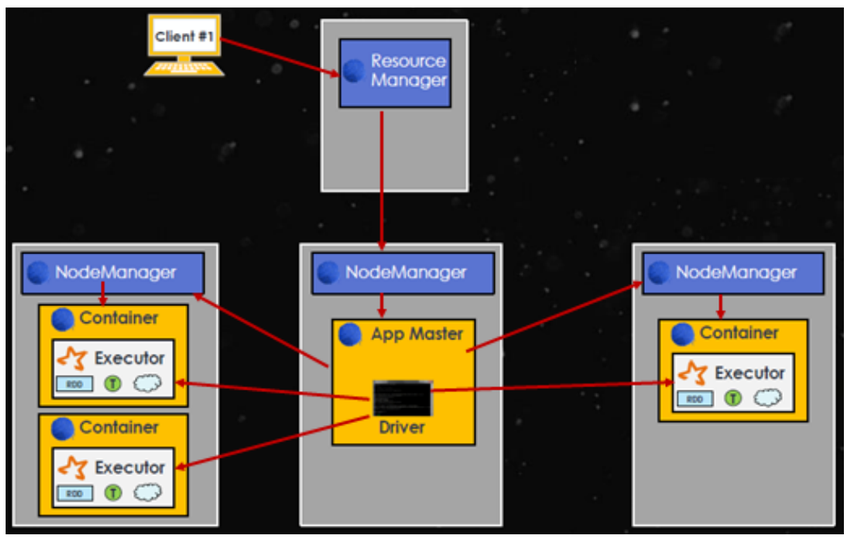
4. The Driver, running on a node in the cluster, returns progress, results, and status to the client.

Client Mode and Cluster Mode



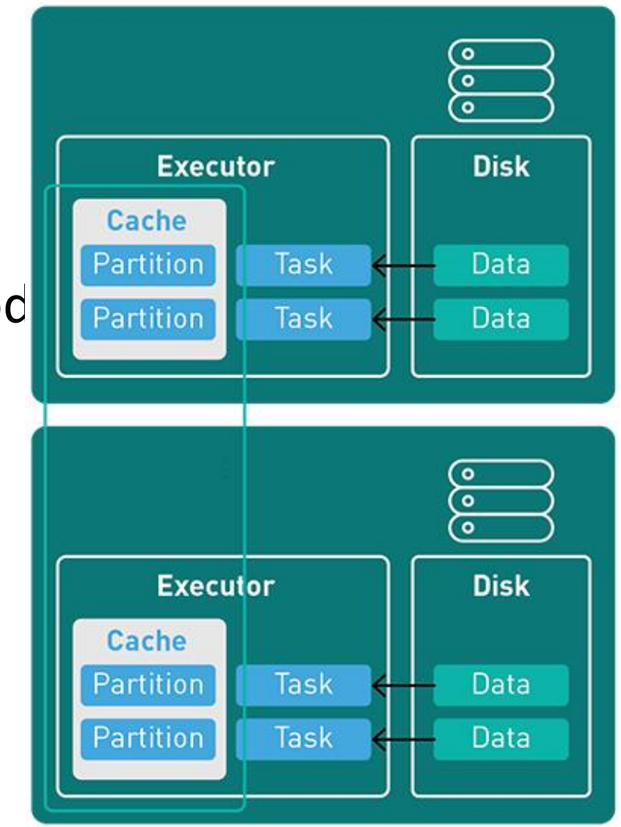






Spark Cache

- ✓one of the strongest features of spark is the ability to cache to RDD.
- ✓Spark can cache the items of the RDD in memory or an disk
- ✓Cache can avoid expensive recalculation this way
- ✓The cache() and persist() store the content in memory
- ✓Spark can provide a different storage level by supplying it to the persist method



Transformation	Description
MEMORY_ONLY	RDD is stored as deserialized Java objects in the JVM. If the RDD does not fit into the memory, it will be only partially cached and missing partitions will be recomputed when they are required. This is the default mode.
MEMORY_AND_DISK	The RDD is stored as deserialized Java objects in the JVM. If the RDD does not fit into memory, it will be partially accessed from disk.
MEMORY_ONLY_SER	The RDD is stored as serialized Java objects (one by array for a partition). This approach is more byte-efficient than the former methods; however, requires more CPU to read. Recomputes missing partitions when required.
MEMORY_AND_DISK_SER	Similar to the above case, but uses disk instead of recomputing the missing partitions.
DISK_ONLY	Store the RDD only to disk.
MEMORY_ONLY_2...	Same as above, but with replication to two clusters.

Shared Variables

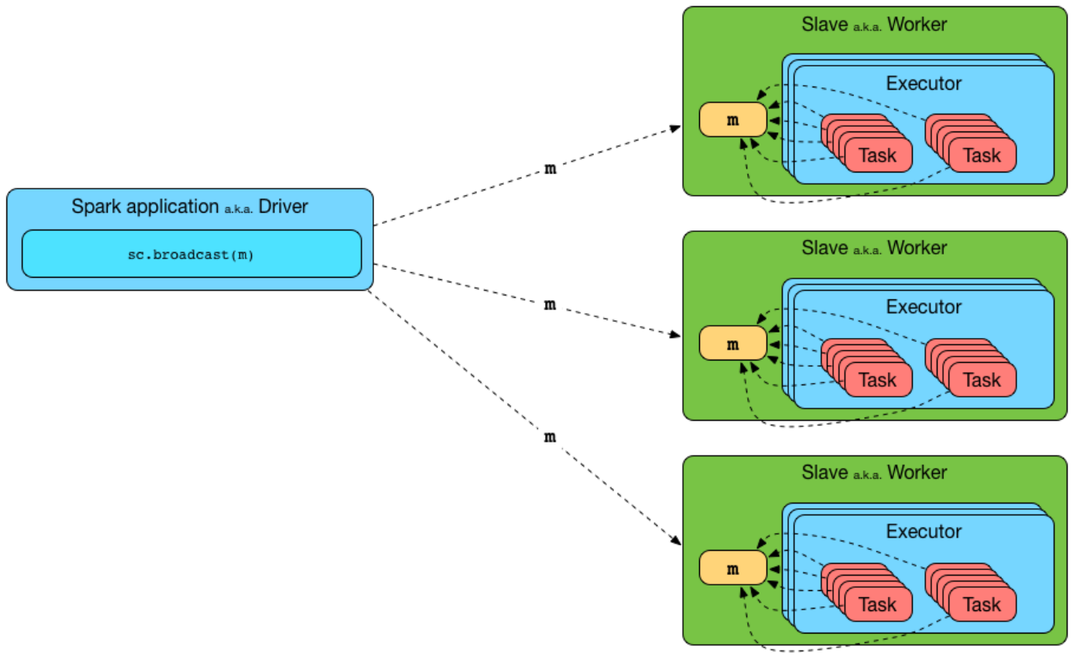
- Normally when functions are executed on a remote node it works on immutable copies
- However, Sparks does provide two types of shared variables for two usages:
  - Broadcast variables
  - Accumulators

Broadcast Variable

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] =
    Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```



Accumulator Variable

- Accumulators are variables that are only “added” to through an associative operation and can therefore be efficiently supported in parallel
- Spark natively supports accumulators of numeric types, and programmers can add support for new types

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: spark.Accumulator[Int] = 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
scala> accum.value
res7: Int = 10
```