Grow Data Skills

APACHE Spark™

Problems with Hadoop Map-Reduce?

1. ***Batch Processing:*** Hadoop and MapReduce are designed for batch processing, making them unfit for real-time or near real-time processing such as streaming data.

2. ***Complexity:*** Hadoop has a steep learning curve and its setup, configuration, and maintenance can be complex and time-consuming.

3. ***Data Movement:*** Hadoop's architecture can lead to inefficiencies and network congestion when dealing with smaller data sets.

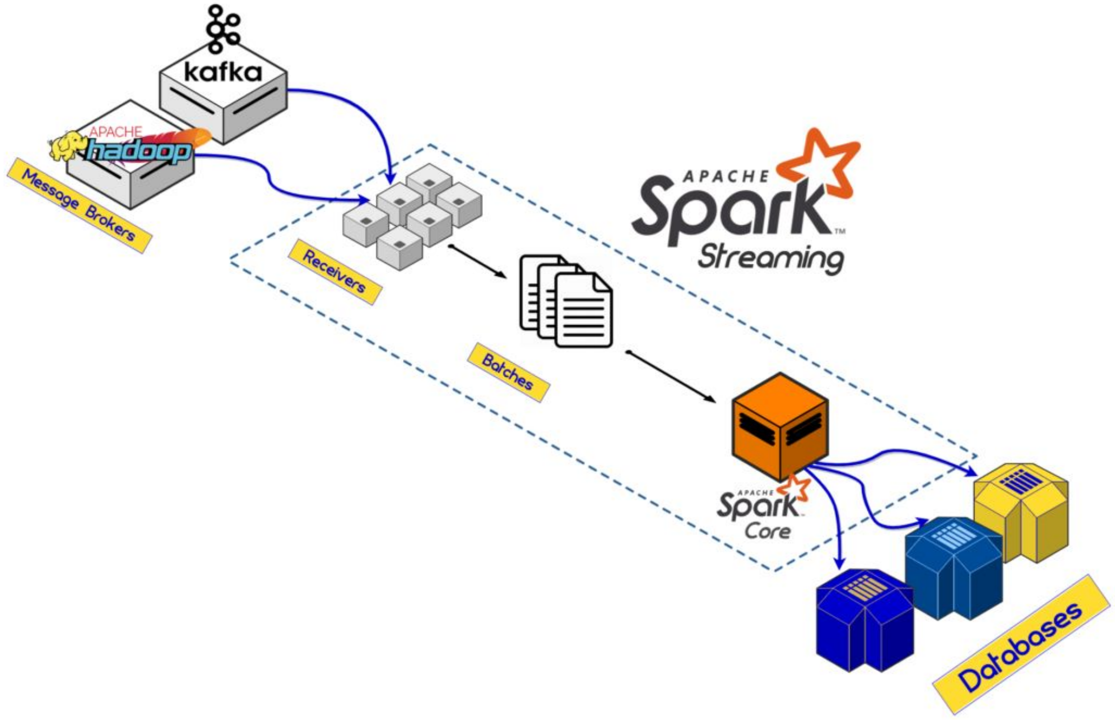
4. ***Fault Tolerance:*** While Hadoop has data replication for fault tolerance, it can lead to inefficient storage use and doesn't cover application-level failures.

5. ***No Support for Interactive Processing:*** MapReduce doesn't support interactive processing, making it unsuitable for tasks needing back-and-forth communication.

6. ***Not Optimal for Small Files:*** Hadoop is less effective with many small files, as it's designed to handle large data files.

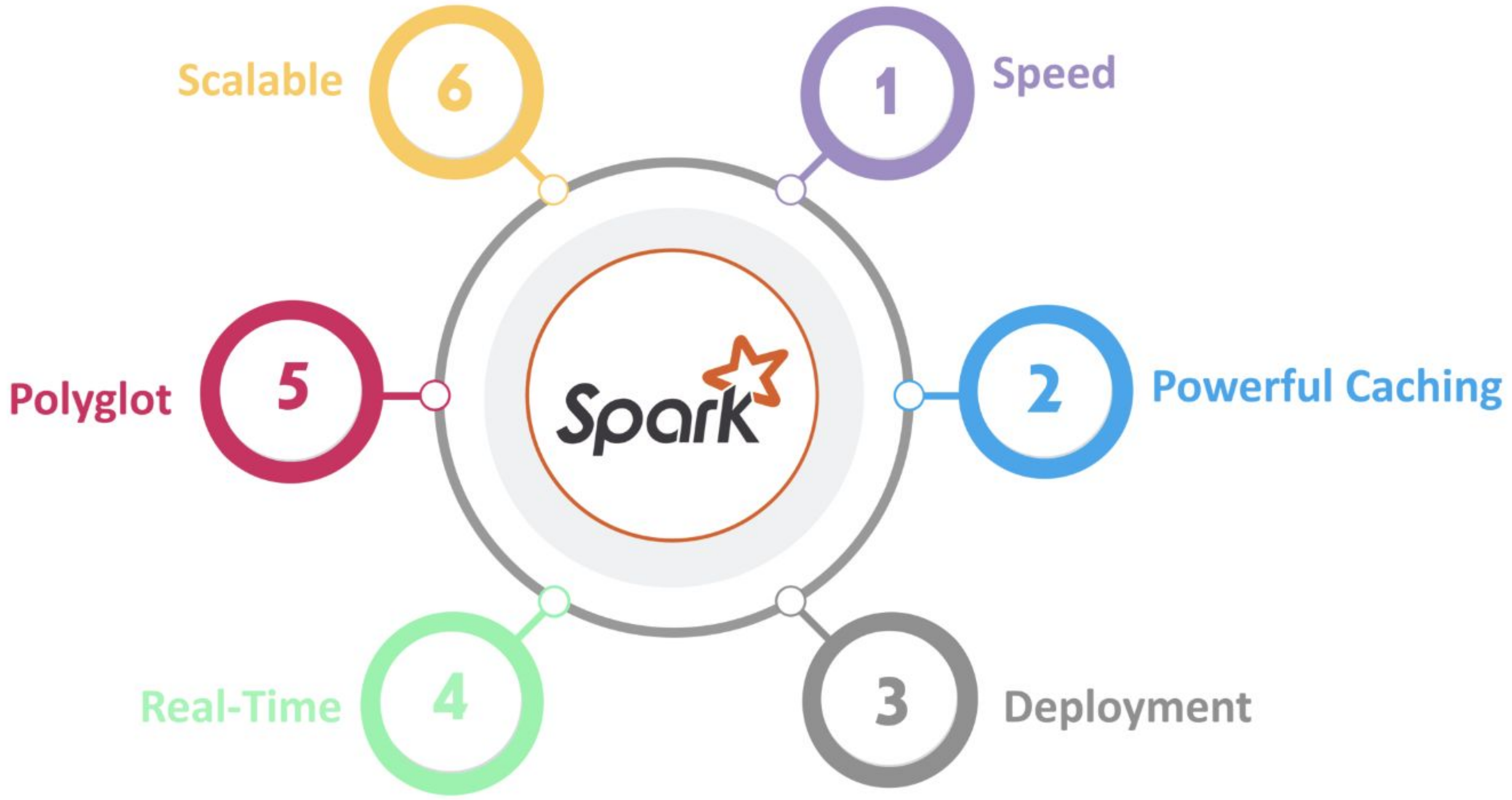
What is Apache Spark?

Apache Spark is an open-source, distributed computing system designed for big data processing and analytics. It provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark is known for its speed, ease of use, and versatility in handling multiple types of data workloads, including batch processing, real-time data streaming, machine learning, and interactive queries.



Features Of Spark

GROW DATA SKILLS



Features Of Spark

1. **Speed:** Compared to Hadoop MapReduce, Spark can execute large-scale data processing up to 100 times faster. This speed is achieved by leveraging controlled partitioning.

2. **Powerful Caching:** Spark's user-friendly programming layer delivers impressive caching and disk persistence capabilities.

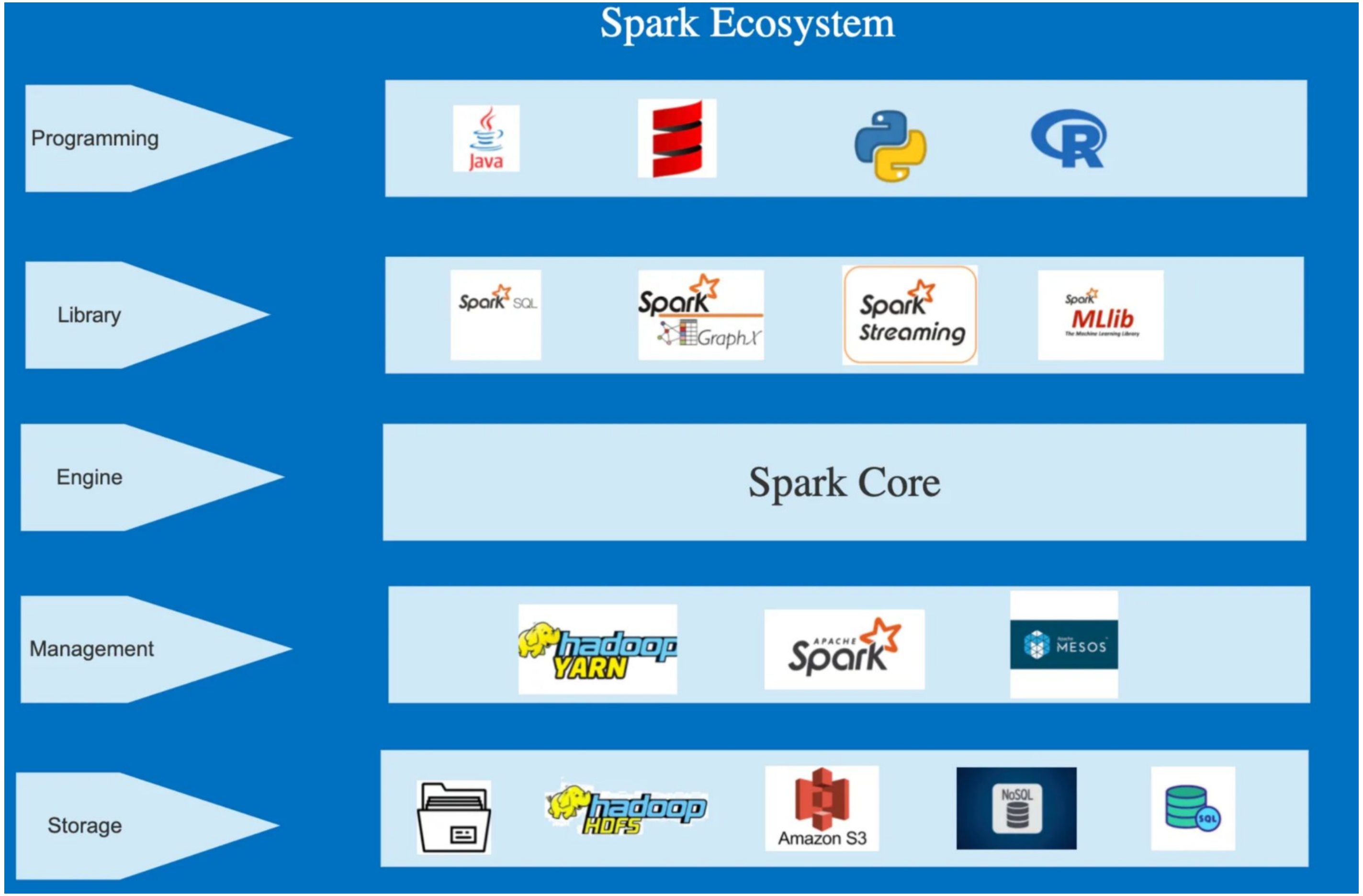
3. **Deployment:** Spark offers versatile deployment options, including through Mesos, Hadoop via YARN, or its own cluster manager.

4. **Real-Time Processing:** Thanks to in-memory computation, Spark facilitates real-time computation and offers low latency.

5. **Polyglot**: Spark provides high-level APIs in several languages - Java, Scala, Python, and R, allowing code to be written in any of these. It also offers a shell in Scala and Python.

6. **Scalability**: Spark's design is inherently scalable, capable of handling and processing large amounts of data by distributing tasks across multiple nodes in a cluster.

Spark Ecosystem



Spark Ecosystem

1. **Spark Core Engine:** The foundation of the entire Spark ecosystem, the Spark Core, handles essential functions such as task scheduling, monitoring, and basic I/O operations. It also provides the core programming abstraction, Resilient Distributed Datasets (**RDDs**).

2. **Cluster Management:** Spark's versatility allows for cluster management by multiple tools, including Hadoop YARN, Apache Mesos, or Spark's built-in standalone cluster manager. This flexibility accommodates varying requirements and operational contexts.

3. **Library**: The Spark ecosystem includes a rich set of libraries:

    a. **Spark SQL** allows SQL-like queries on **RDDs** or data from external sources, integrating relational processing with Spark's functional programming API.
    b. **Spark MLlib** is a library for machine learning that provides various algorithms and utilities.
    c. **Spark GraphX** allows for the construction and computation on graphs, facilitating advanced data visualization and graph computation.
    d. **Spark Streaming** makes it easy to build scalable, high-throughput, fault-tolerant streaming applications that can handle live data streams alongside batch processing.

4. **Polyglot Programming**: Spark supports programming in multiple languages including Python, Java, Scala, and R. This broad language support makes Spark accessible to a wide range of developers and data scientists.

5. **Storage Flexibility**: Spark can interface with a variety of storage systems, including HDFS, Amazon S3, local filesystems, and more. It also supports interfacing with both SQL and NoSQL databases, providing broad flexibility for various data storage and processing needs.

RDD in Spark

RDDs are the building blocks of any Spark application. RDDs Stands for:

- **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

Resilient Distributed Datasets (RDD) are a core abstraction in Apache Spark. Here are some key points about RDDs and their properties:

1. **Fundamental Data Structure:** RDD is the fundamental data structure of Spark, which allows it to efficiently operate on large-scale data across a distributed environment.

2. **Immutability**: Once an RDD is created, it cannot be changed. Any transformation applied to an RDD creates a new RDD, leaving the original one untouched.

3. **Resilience**: RDDs are fault-tolerant, meaning they can recover from node failures. This resilience is provided through a feature known as lineage, a record of all the transformations applied to the base data.

RDD in Spark

3. **Lazy Evaluation**: RDDs follow a lazy evaluation approach, meaning transformations on RDDs are not executed immediately, but computed only when an action (like count, collect) is performed. This leads to optimized computation.

4. **Partitioning**: RDDs are partitioned across nodes in the cluster, allowing for parallel computation on separate portions of the dataset.

5. **In-Memory Computation**: RDDs can be stored in the memory of worker nodes, making them readily available for repeated access, and thereby speeding up computations.

6. **Distributed Nature**: RDDs can be processed in parallel across a Spark cluster, contributing to the overall speed and scalability of Spark.

7. **Persistence**: Users can manually persist an RDD in memory, allowing it to be reused across parallel operations. This is useful for iterative algorithms and fast interactive use.

8. **Operations**: Two types of operations can be performed on RDDs - transformations (which create a new RDD) and actions (which return a value to the driver program or write data to an external storage system).

How Spark Perform Data Partitioning?

1. **Data Partitioning**: Apache Spark partitions data into logical chunks during reading from sources like HDFS, S3, etc

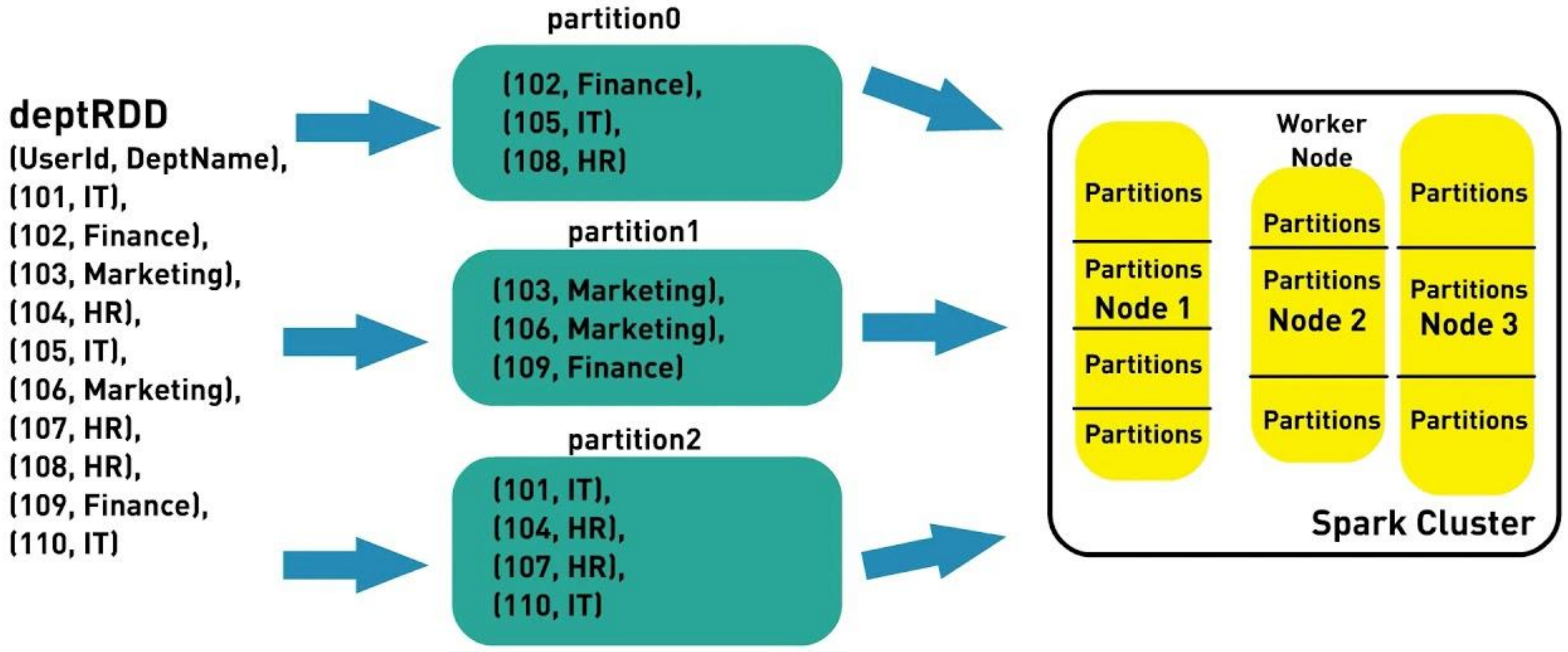
2. **Data Distribution**: These partitions are distributed across the Spark cluster nodes, allowing for parallel processing.

3. **Custom Partitioning**: Users can control data partitioning using Spark's **repartition(), coalesce()** and **partitionBy()** methods, optimizing data locality or skewness.

When Apache Spark reads data from a file on HDFS or S3, the number of partitions is determined by the size of the data and the default block size of the file system. In general, each partition corresponds to a block in HDFS or an object in S3.

For example, if HDFS is configured with a block size of **128MB** and you have a **1GB** file, it would be divided into 8 blocks in HDFS. Therefore, when Spark reads this file, it would create 8 partitions, each corresponding to a block.

How Spark Perform Data Partitioning?



Transformation in Spark

In Spark, a transformation is an operation applied on an RDD (Resilient Distributed Dataset) or DataFrame/Dataset to create a new RDD or DataFrame/Dataset. Transformations in Spark are categorized into two types: narrow and wide transformations.

Narrow Transformations: In these transformations, all elements that are required to compute the records in a single partition live in the same partition of the parent RDD. Data doesn't need to be shuffled across partitions. Examples include:

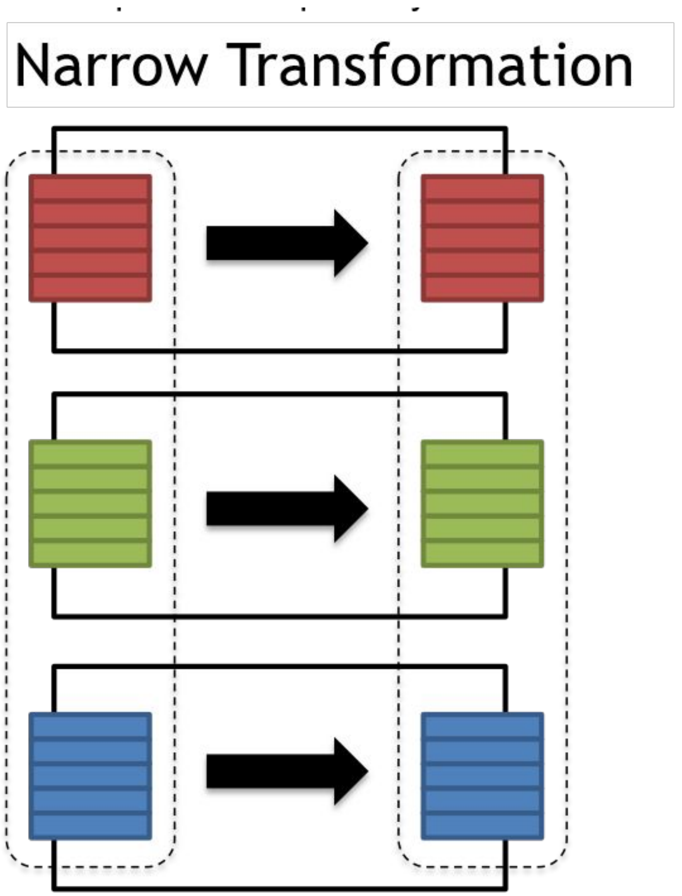
- **map():** Applies a function to each element in the RDD and outputs a new RDD.
- **filter():** Creates a new RDD by selecting only the elements of the original RDD that pass a function's condition.
- **flatMap():** Function in Spark applies a function to each element of an RDD, then flattens the multiple outputs into a single RDD.
- **sample():** Create a sample dataset from the original data.

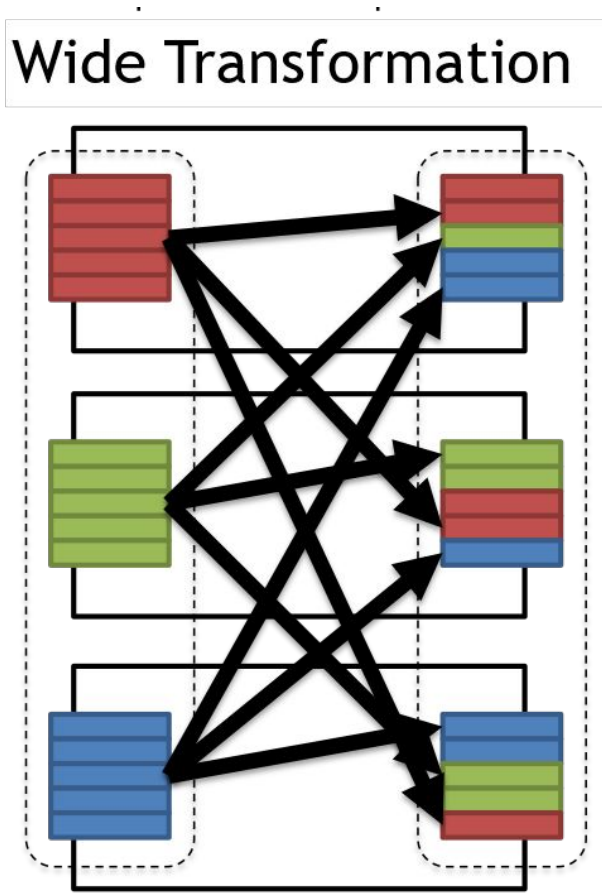
Wide Transformations: These transformations will have input data from multiple partitions. This typically involves shuffling all the data across multiple partitions. Examples include:

- **groupByKey():** Groups all the values of each key in the RDD into a single sequence.
- **reduceByKey():** Performs a reduction operation for each key in the RDD.
- **join():** Joins two RDDs based on a common key, similar to the SQL JOIN operation.
- **distinct():** Remove duplicates in the RDD.
- **coalesce():** Decreases the number of partitions in the RDD.
- **repartition():** Increases the number of partitions in the RDD.

Transformation in Spark

GROW DATA SKILLS





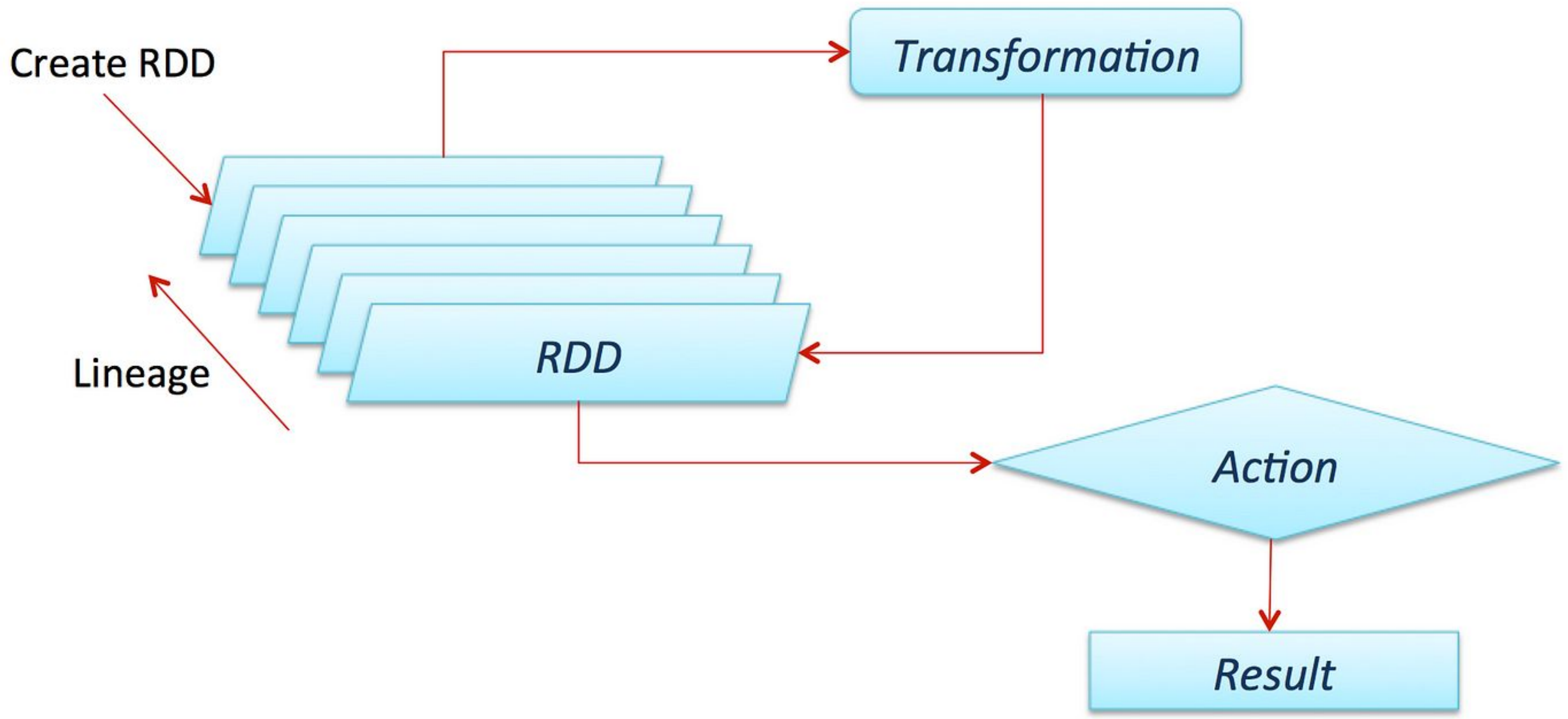
Action in Spark

Actions in Apache Spark are operations that provide non-RDD values; they return a final value to the **driver program** or **write** data to an external system. Actions trigger the execution of the transformation operations accumulated in the **Directed Acyclic Graph (DAG).**

Here are some of the commonly used actions in Spark:

- **Collect**: collect() returns all the elements of the RDD as an array to the driver program. This can be useful for testing and debugging, but be careful with large datasets to avoid out-of-memory errors.
- **Count**: count() returns the number of elements in the RDD.
- **First**: first() returns the first element of the RDD.
- **Take**: take(n) returns the first n elements of the RDD.
- **foreach**: foreach() is used for performing computations on each element in the RDD.
- **SaveAsTextFile**: saveAsTextFile() writes the elements of the dataset to a text file (or set of text files) in a specified directory in the local filesystem, HDFS, or any other Hadoop-supported file system.
- **SaveAsSequenceFile**: This action is used to save RDDs, which consist of key/value pairs, in SequenceFile format.

Action in Spark



Read & Write operation in Spark are Transformation/Action?

Reading and writing operations in Spark are often viewed as actions, but they're a bit unique. Let's clarify this:

- **Read Operation:** Transformations , especially read operations can behave in two ways according to the arguments you provide:

  - Lazily evaluated --> It will be performed only when an action is called
  - Eagerly evaluated --> A job will be triggered to do some initial evaluations

  In case of read.csv()

  - If it is called **without** defining the schema and **inferSchema** is **disabled**, it determines the columns as string types and it reads only the first line to determine the names (if **header**=**True**, otherwise it gives default column names) and the number of fields. Basically it performs a **collect** operation with limit 1, which means **one new job** is created instantly
  - Now if you specify **inferSchema**=**True**, Here above job will be triggered first as well as one more job will be triggered which will scan through entire record to determine the schema, that's why you are able to see **2 jobs** in spark UI
  - Now If you specify **schema explicitly** by providing StructType() schema object to 'schema' argument of read.csv(), then you can see no jobs will be triggered here. This is because, we have provided the number of columns and type explicitly and catalogue of spark will store that information and now it doesn't need to scan the file to get that information and this will be validated lazily at the time of calling action

- **Write Operation:** Writing or saving data in Spark, on the other hand, is considered an action. Functions like **saveAsTextFile**(), **saveAsSequenceFile**(), **saveAsObjectFile**(), or DataFrame write options trigger computation and result in data being written to an external system.

Lazy Evaluation in Spark

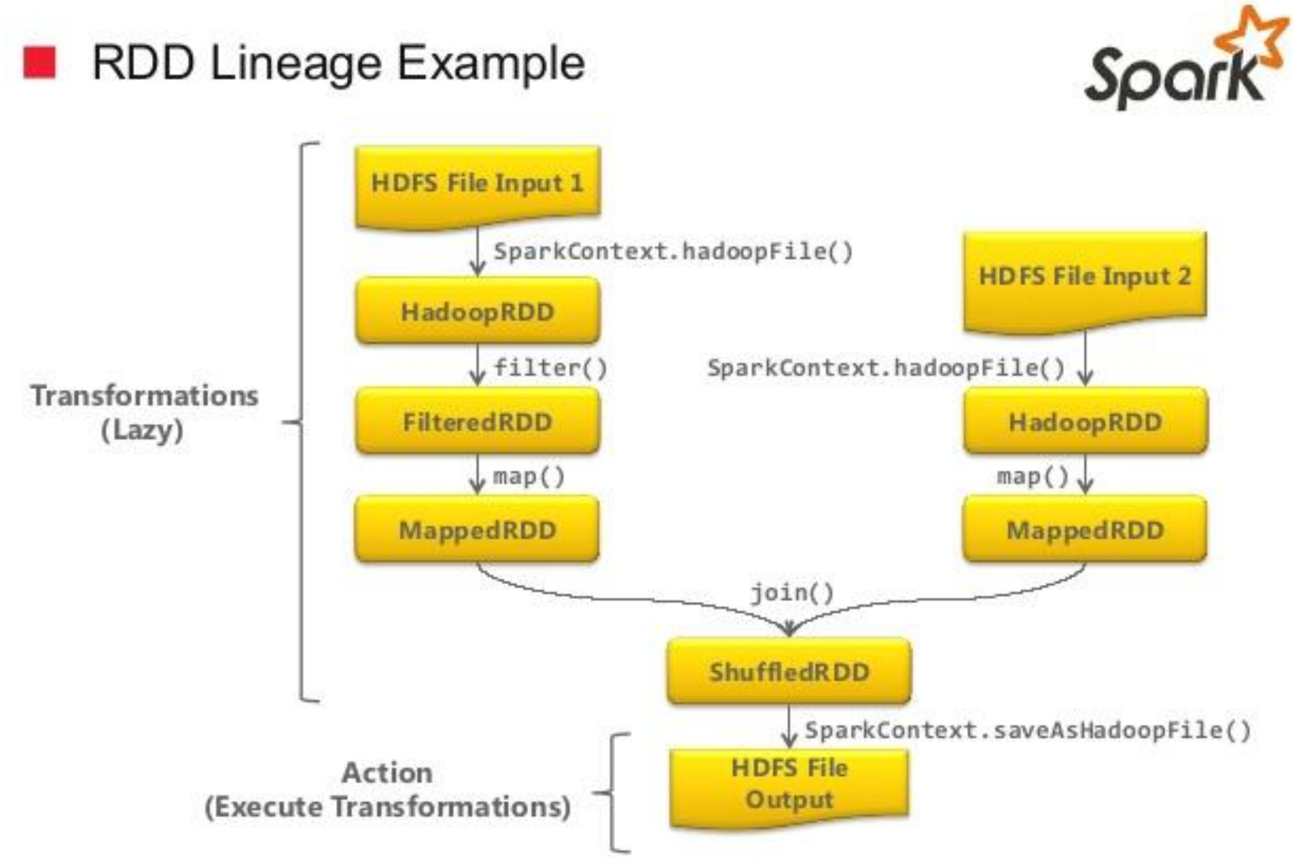
Lazy evaluation in Spark means that the **execution doesn't start until an action is triggered**. In Spark, transformations are lazily evaluated, meaning that the system records how to compute the new RDD (or DataFrame/Dataset) from the existing one without performing any transformation. The transformations are only actually computed when an action is called and the data is required.

Lineage Graph or DAG in Spark

Spark represents a sequence of transformations on data as a DAG, a concept borrowed from mathematics and computer science. A DAG is a directed graph with no cycles, and it represents a finite set of transformations on data with multiple stages. The **nodes** of the graph represent the **RDDs** or DataFrames/Datasets, and the edges represent the transformations or operations applied.

Each action on an RDD (or DataFrame/Dataset) triggers the creation of a **new DAG**. The DAG is optimized by the Catalyst optimizer (in case of DataFrame/Dataset) and then it is sent to the **DAG scheduler**, which splits the graph into **stages** of **tasks**.

Lineage Graph or DAG in Spark



Job, Stage and Task in Spark

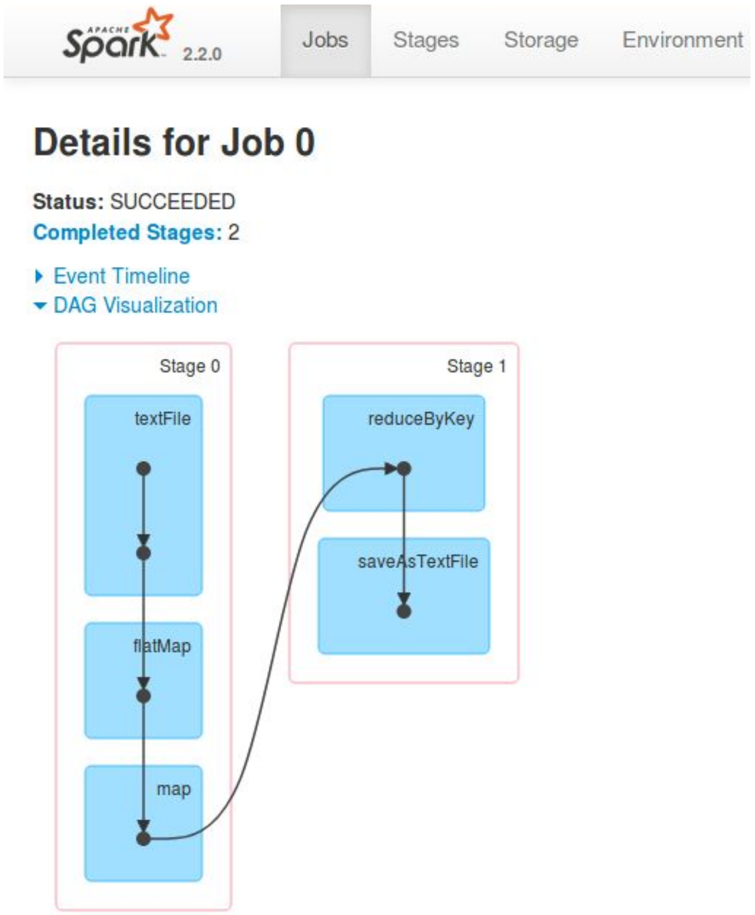
- **Job:** A job in Spark represents a single action (like count, collect, save, etc.) from a Spark application. When an action is called on a DataFrame or RDD in the program, a job is created. A job is a full program from start to finish, including reading the initial data, performing transformations, and executing actions. A Spark application can consist of multiple jobs, and each job is independent of the others.

- **Stage**: A stage in Spark is a sequence of transformations on an RDD or DataFrame/Dataset that can be performed in a single pass (i.e., without shuffling data around). Spark splits the computation of a job into a series of stages, separated by shuffle boundaries. Each stage represents a sequence of transformations that can be done in a single scan of the data. In essence, a stage is a step in the physical execution plan.

- **Task**: Within each stage, the data is further divided into partitions, and each partition is processed in parallel. A task in Spark corresponds to a single unit of work sent to one executor. So, if you have two stages with two partitions each, Spark will generate four tasks in total - one task per partition per stage. Each task works on a different subset of the data, and the tasks within a stage can be run in parallel.

How DAG looks on Spark Web UI?

GROW DATA SKILLS



Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

Example

```
from pyspark.sql import SparkSession

# Initialize Spark
spark = SparkSession.builder \
    .appName("PySpark Application") \
    .getOrCreate()

# Read CSV files
employees = spark.read.csv('employees.csv', inferSchema=True, header=True)
departments = spark.read.csv('departments.csv', inferSchema=True, header=True)
regions = spark.read.csv('regions.csv', inferSchema=True, header=True) # Adding a third DataFrame

# Narrow transformation: Filter
filtered_employees = employees.filter(employees.age > 30)

# Wide transformation: Join
result = filtered_employees.join(departments, filtered_employees.dept_id == departments.dept_id)

# Another wide transformation: Join with regions
result_with_regions = result.join(regions, result.region_id == regions.region_id)

# Action: Collect
result_list = result_with_regions.collect()

# Narrow transformation: Select a few columns
selected_data = result_with_regions.select('employee_name', 'department_name', 'region_name')

# Action: Save as CSV
selected_data.write.csv('result.csv')

# Stop Spark
spark.stop()
```

Example

The jobs and their associated stages in the PySpark script example would be as follows:

Job 0 & 1: To read **employees** csv data and infer schema part
Job 2 & 3: To read **departments** csv data and infer schema part
Job 4 & 5: To read **regions** csv data and infer schema part

Job 6:

Triggered by the collect() action. This job consists of three stages:

- **Stage 0:** Filter transformation on 'employees' DataFrame.
- **Stage 1:** Join transformation between 'filtered_employees' and 'departments' DataFrames.
- **Stage 2:** Join transformation between 'result' and 'regions' DataFrames.

Job 7:

Triggered by the write.csv() action. This job consists of one stage:

The **select**() transformation and the **write.csv**() action ***do not require a shuffle*** and therefore **do not trigger a new stage** within Job 1.

What if our cluster capacity is less than the size of data to be processed?

If your cluster memory capacity is less than the size of the data to be processed, Spark can still handle it by leveraging its ability to perform computations on disk and spilling data from memory to disk when necessary. Let's break down how Spark will handle a 60 GB data load with a 30 GB memory cluster:

- **Data Partitioning:** When Spark reads a 60 GB file from HDFS, it partitions the data into manageable blocks, according to the Hadoop configuration parameter dfs.blocksize or manually specified partitions. These partitions can be processed independently.

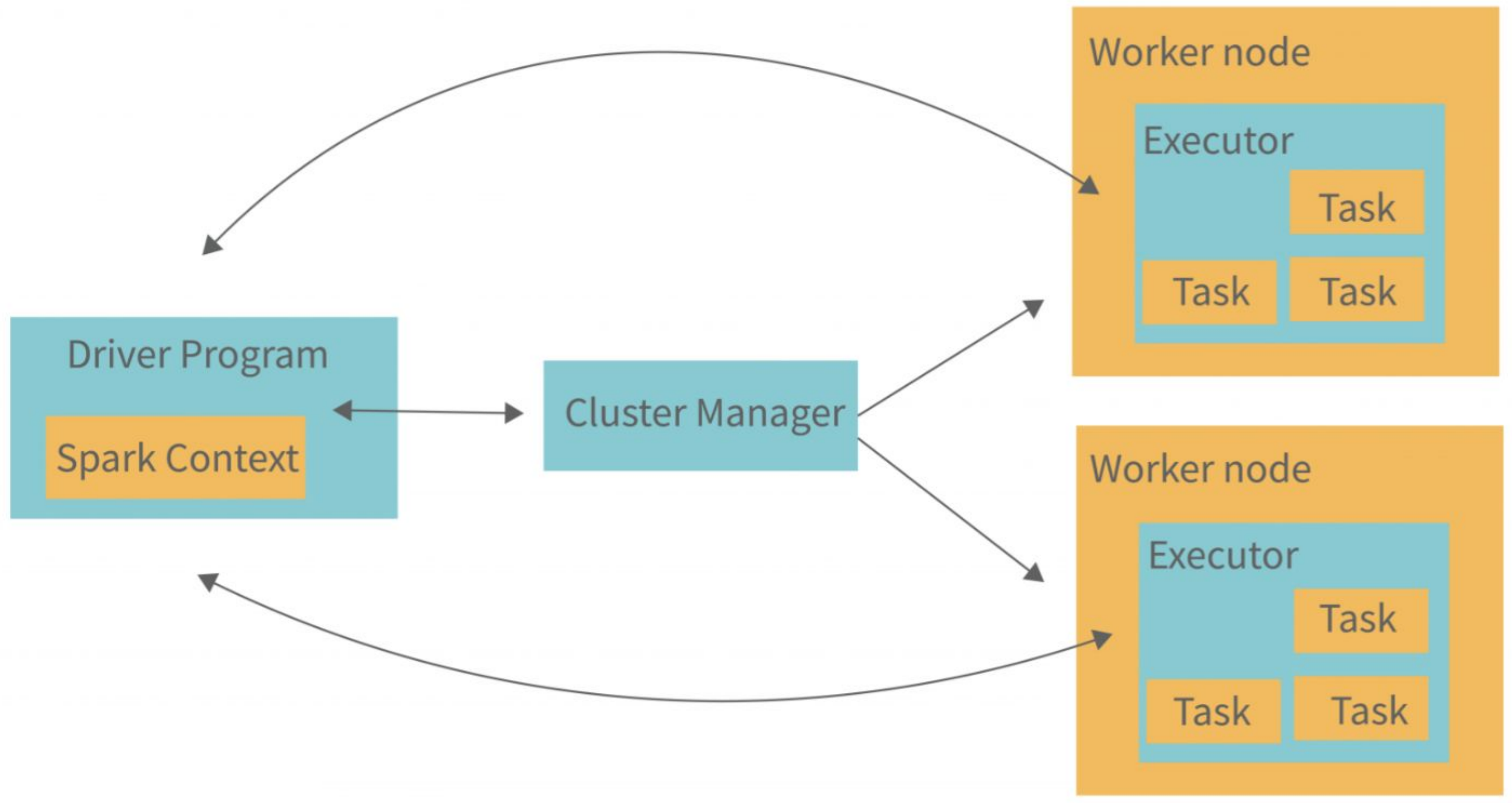
- **Loading Data into Memory:** Spark will load as many partitions as it can fit into memory. It starts processing these partitions. The size of these partitions is much smaller than the total size of your data (60 GB), allowing Spark to work within the confines of your total memory capacity (30 GB in this case).

- **Spill to Disk**: When the memory is full, and Spark needs to load new partitions for processing, it uses a mechanism called "**spilling**" to free up memory. Spilling means writing data to disk. The spilled data is the intermediate data generated during shuffling operations, which needs to be stored for further stages.

- **On-Disk Computation**: Spark has the capability to perform computations on data that is stored on disk, not just in memory. Although computations on disk are slower than in memory, it allows Spark to handle datasets that are larger than the total memory capacity.

- **Sequential Processing**: The stages of the job are processed sequentially, meaning Spark doesn't need to load the entire dataset into memory at once. Only the data required for the current stage needs to be in memory or disk.

Spark Architecture & Its components



Spark Architecture & Its components

- **Driver Program:** The driver program is the heart of a Spark application. It runs the main() function of an application and is the place where the SparkContext is created. SparkContext is responsible for coordinating and monitoring the execution of tasks. The driver program defines datasets and applies operations (transformations & actions) on them.

- **SparkContext:** The SparkContext is the main entry point for Spark functionality. It represents the connection to a Spark cluster and can be used to create RDDs, accumulators, and broadcast variables on that cluster.

- **Cluster Manager:** SparkContext connects to the cluster manager, which is responsible for the allocation of resources (CPU, memory, etc.) in the cluster. The cluster manager can be Spark's standalone manager, Hadoop YARN, Mesos, or Kubernetes.

- **Executors:** Executors are worker nodes' processes in charge of running individual tasks in a given Spark job. They run concurrently across different nodes. Executors have two roles. Firstly, they run tasks that the driver sends. Secondly, they provide in-memory storage for RDDs.

- **Tasks:** Tasks are the smallest unit of work in Spark. They are transformations applied to partitions. Each task works on a separate partition and is executed in a separate thread in executors.

- **RDD**: Resilient Distributed Datasets (RDD) are the fundamental data structures of Spark. They are an immutable distributed collection of objects, which can be processed in parallel. RDDs can be stored in memory between queries without the necessity for serialization.

- **DAG (Directed Acyclic Graph):** Spark represents a series of transformations on data as a DAG, which helps it optimize the execution plan. DAG enables pipelining of operations and provides a clear plan for task scheduling.

Spark Architecture & Its components

- **DAG Scheduler:** The Directed Acyclic Graph (DAG) Scheduler is responsible for dividing operator graphs into stages and sending tasks to the Task Scheduler. It translates the data transformations from the logical plan (which represents a sequence of transformations) into a physical execution plan. It optimizes the plan by rearranging and combining operations where possible, groups them into stages, and then submits the stages to the Task Scheduler.

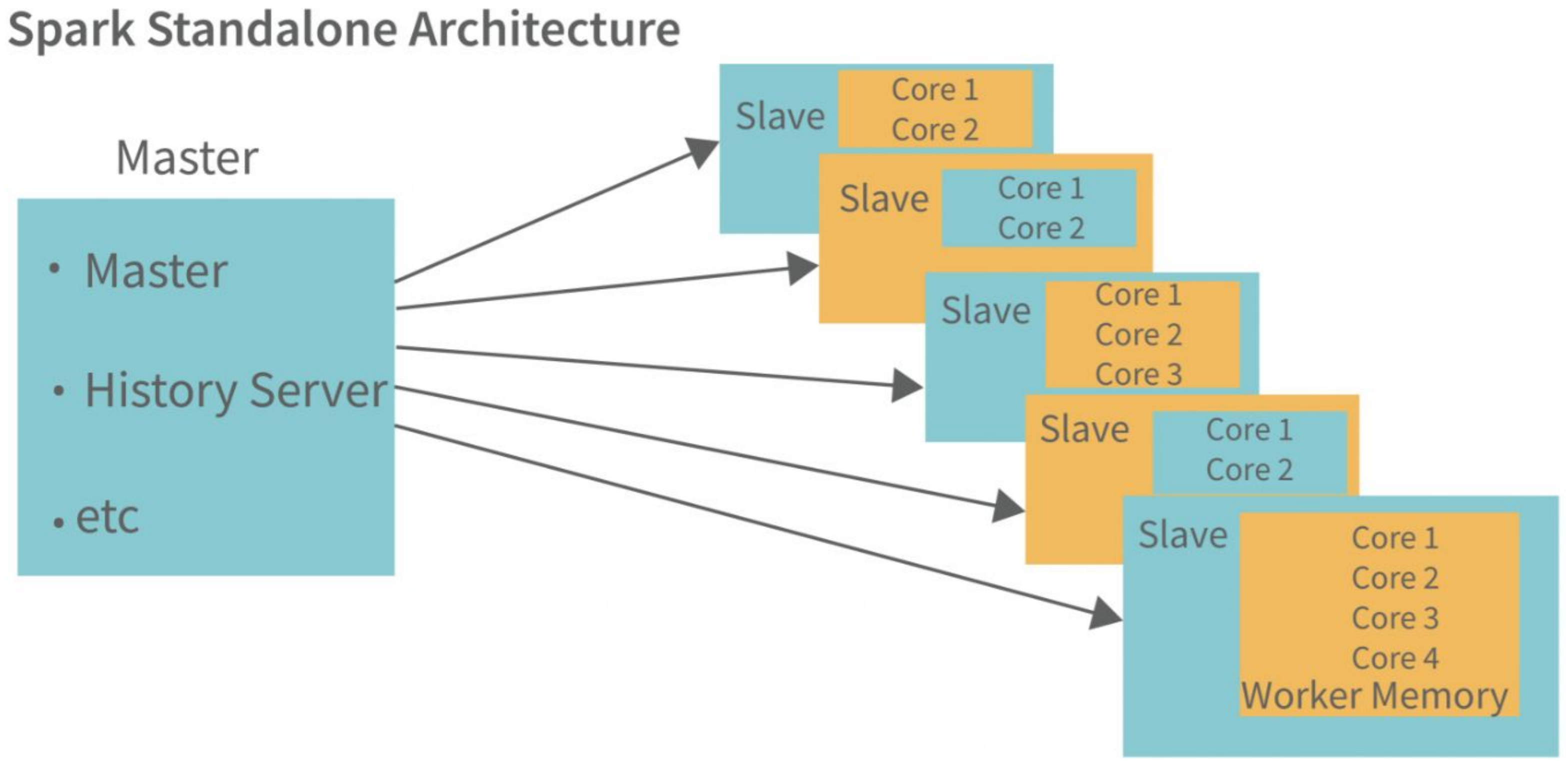
- **Task Scheduler:** The Task Scheduler launches tasks via cluster manager. Tasks are the smallest unit of work in Spark, sent by the DAG Scheduler to the Task Scheduler. The Task Scheduler then launches the tasks on executor JVMs. Tasks for each stage are launched in as many parallel operations as there are partitions for the dataset.

- **Master:** The Master is the base of a Spark Standalone cluster **(specific to Spark's standalone mode, not applicable if Spark is running on YARN or Mesos)**. It's the central point and entry point of the Spark cluster. It is responsible for managing and distributing tasks to the workers. The Master communicates with each of the workers periodically to check if it is still alive and if it has completed tasks.

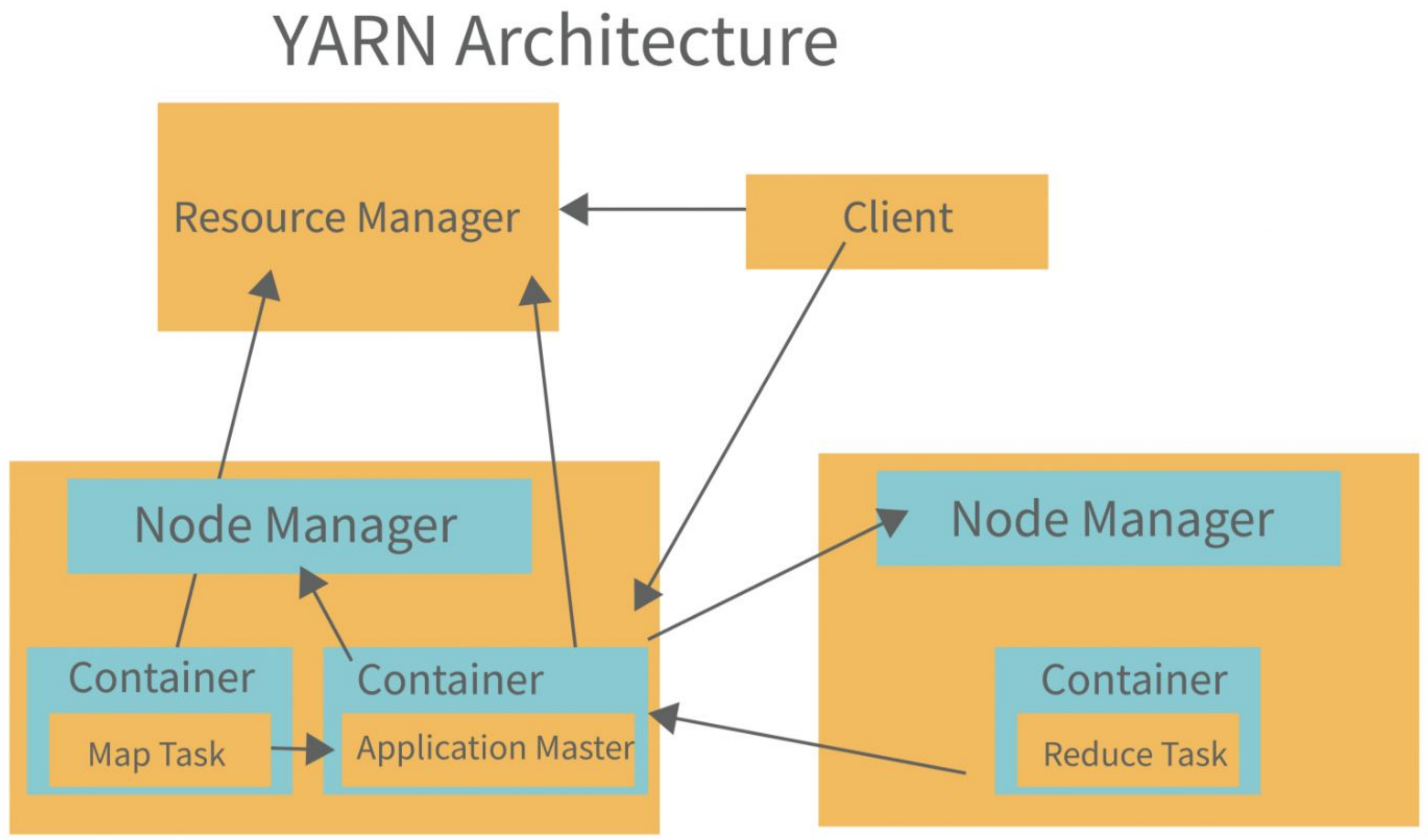
- **Worker:** The Worker is a node in the Spark Standalone cluster **(specific to Spark's standalone mode)**. It receives tasks from the Master and executes them. Each worker has multiple executor JVMs running on it. It communicates with the Master and Executors to facilitate task execution. The worker is responsible for managing resources and providing an execution environment for the executor JVMs.

If Spark is running on YARN (Yet Another Resource Negotiator), the concept of a "master" node doesn't directly apply in the same way it does in Spark's standalone mode. Instead, resource management, scheduling and coordination are handled by YARN's own components.

Spark With Standalone Cluster Manager Type



Spark With YARN Cluster Manager Type



Spark With YARN Cluster Manager Type

- **Resource Manager:** It controls the allocation of system resources on all applications. A **Scheduler** and an **Application Master** are included. Applications receive resources from the Scheduler.

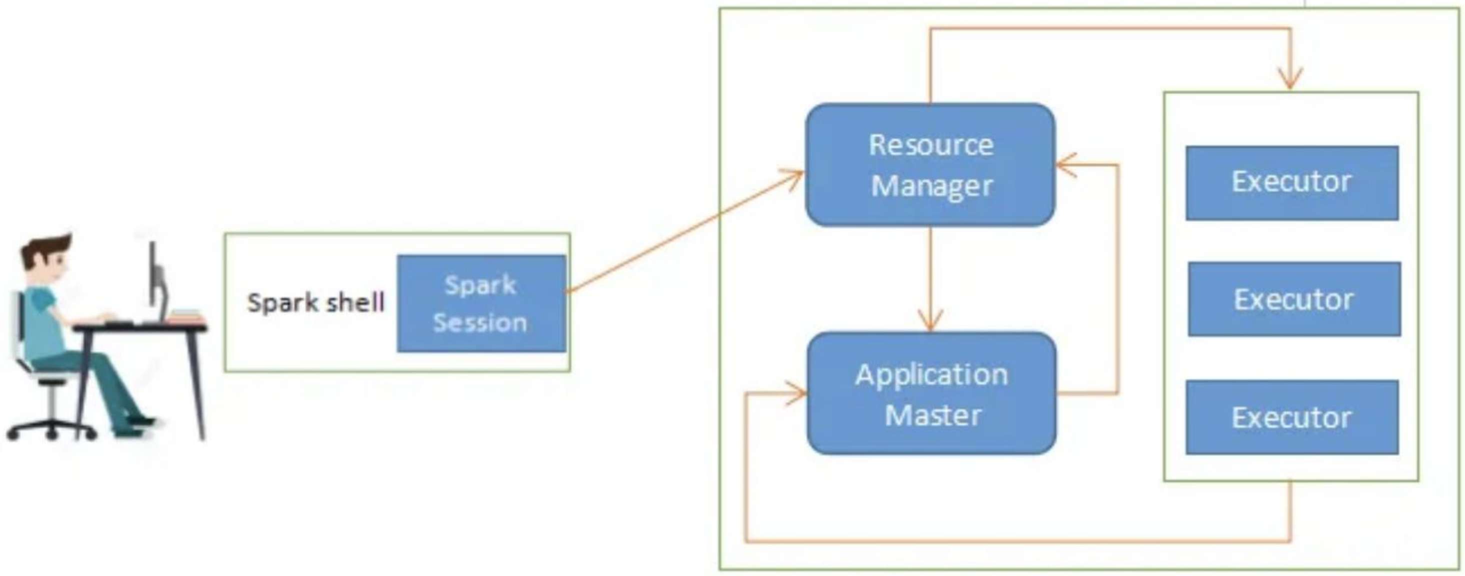
- **Node Manager:** Each job or application needs one or more containers, and the **Node Manager** monitors these containers and their usage. Node Manager consists of an **Application Master** and **Container**. The Node Manager monitors the containers and resource usage, and this is reported to the Resource Manager.
- **Application Master:** The ApplicationMaster (AM) is an instance of a framework-specific library and serves as the orchestrating process for an individual application in a distributed environment.

Deployment Modes Of Spark

GROW DATA SKILLS

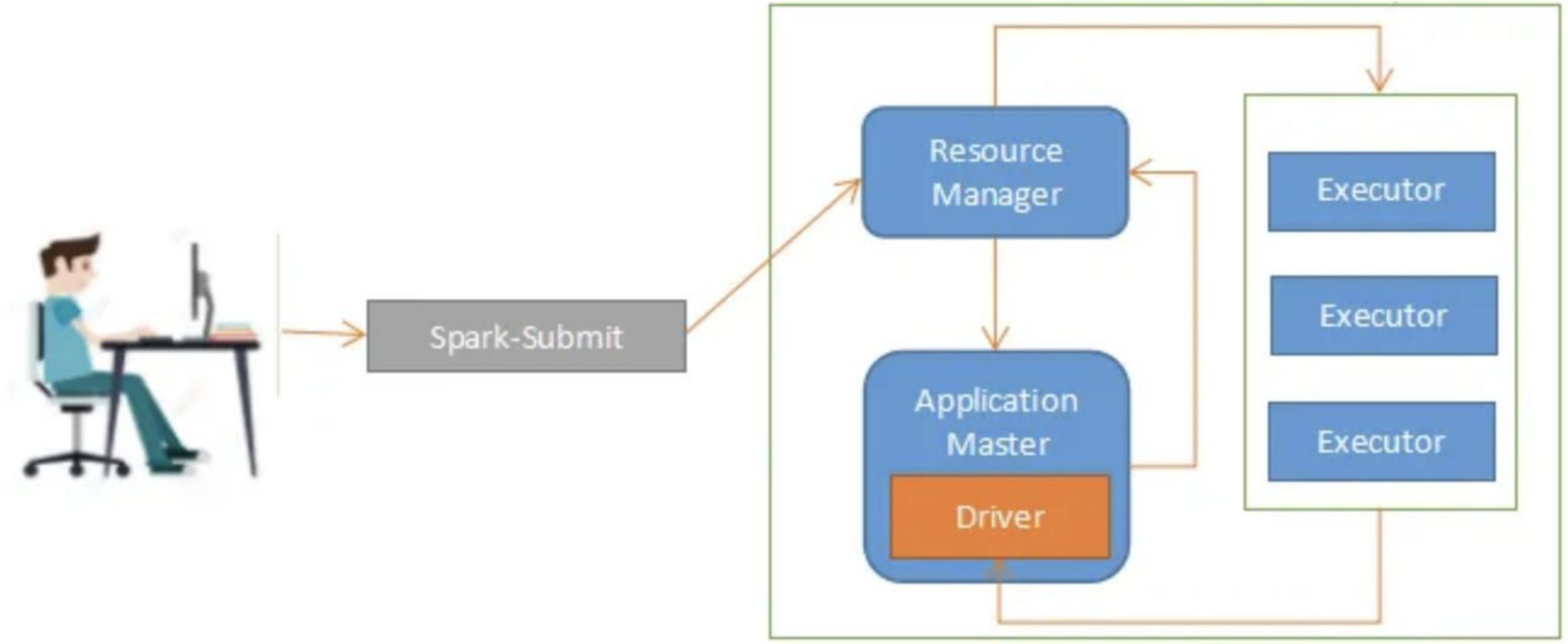
- **Client Mode:** When u start a spark shell, application driver creates the spark session in your local machine which request to Resource Manager present in cluster to create Yarn application. YARN Resource Manager start an Application Master (AM container). For client mode Application Master acts as the Executor launcher. Application Master will reach to Resource Manager and request for further containers. Resource manager will allocate new containers.

  These executors will directly communicate with Drivers which is present in the system in which you have submitted the spark application.



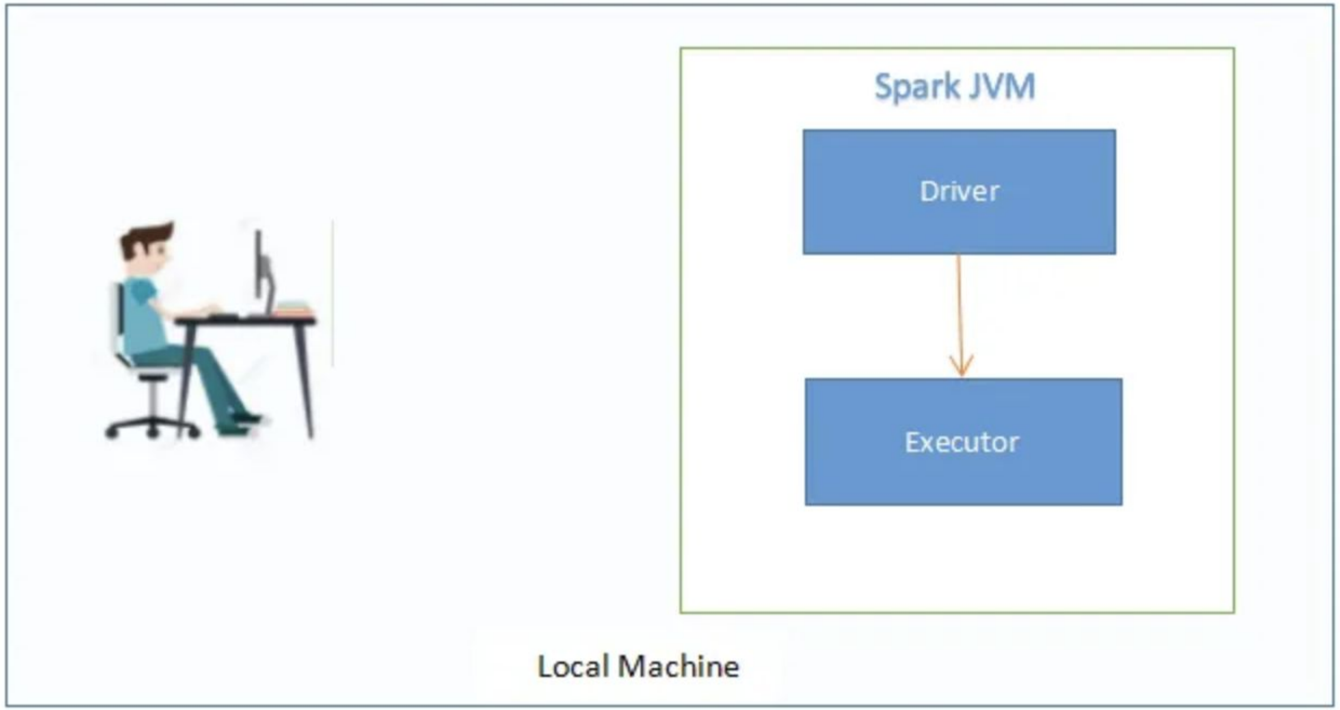
Deployment Modes Of Spark

- **Cluster Mode:** For cluster mode, there's a small difference compare to client mode in place of driver. Here Application Master will create driver in it and driver will reach to Resource Manager.

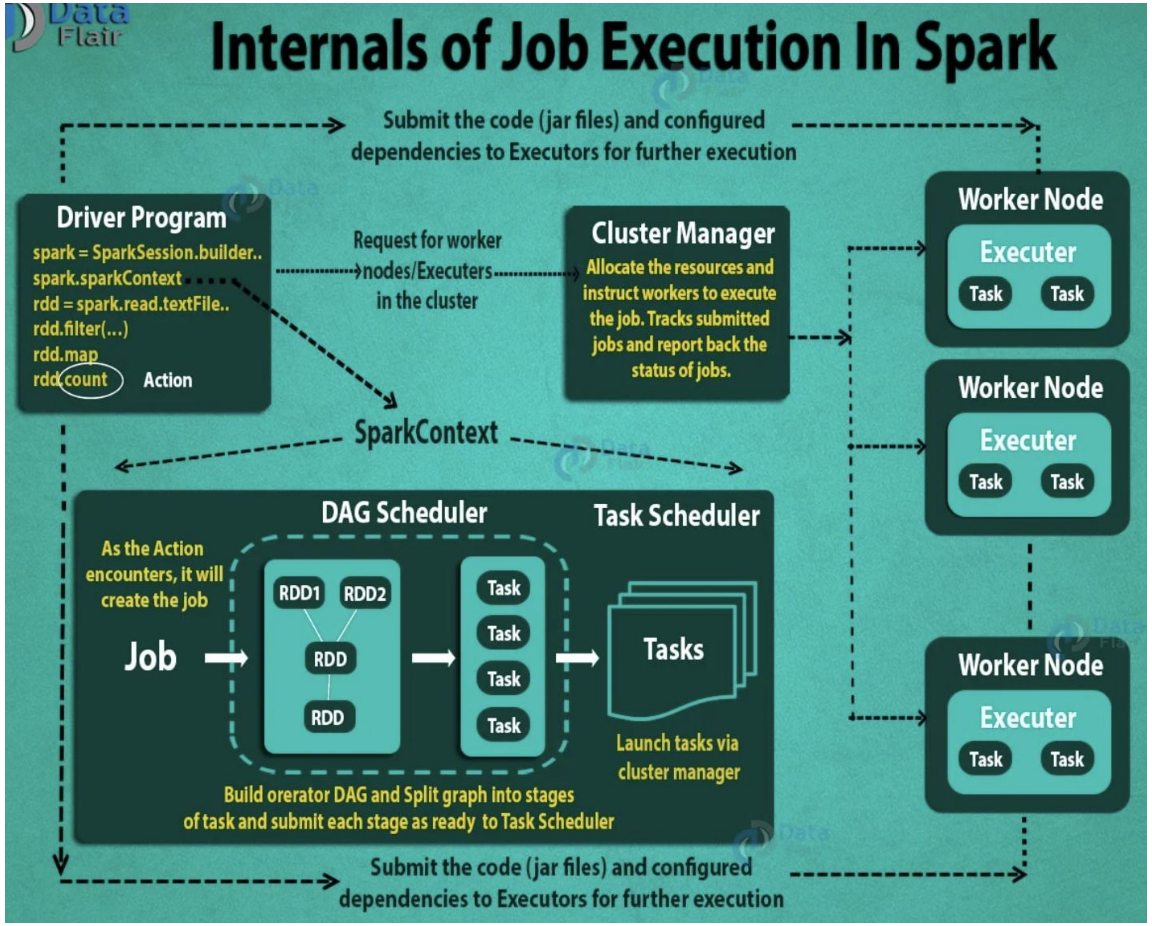


Deployment Modes Of Spark

- **Local Mode:** In local mode, Spark runs on a single machine, using all the cores of the machine. It is the simplest mode of deployment and is mostly used for testing and debugging.



How Spark Job Runs Internally?


Data Flair
Internals of Job Execution In Spark
Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution