* **SPARK ARCHITECTURE:**

# A screenshot of a computer Description automatically generated

Spark uses the **Master/Slave architecture**. Whenever we submit the command, a Driver program is launched on a node depending upon the mode of execution(client/cluster mode) which essentially runs the main method of the class in execution. **The Driver program creates a Spark Context/Session that stays until the application’s lifecycle.**

1. **The Spark Context helps create the** **Operator Graph or DAG**(Directed Acyclic Graph) based on the transformations in the running program. This DAG defines the several steps of the program and also consists of the RDD lineages which can be re-used to re-create the RDD in case of job failures.
2. **Once an action is encountered a Job is created.** [Action🡪show(), etc.]
3. **This Job is essentially submitted to DAG Scheduler**. The task of the DAG Scheduler is to divide the DAG/Operator Graph into different stages, which are further divided into Tasks.
4. **These tasks are then submitted to the Task Scheduler** which launches the tasks via the Cluster Manager on the different worker nodes and the executors execute these jobs. For each partition of the data, a task is launched.

The Cluster manager is responsible to launch and allocate the resources to the executors. It can request more resources or decrease the executors based on the workload of data being processed. These executors are responsible to run the tasks.

Each Executor is also a separate JVM process. They have their JVM and memory allocated based on the configurations passed into the Spark-Submit command. Each executor can cache data which can be re-used in further stages.

Once the tasks are completed, the results are shared back to the Driver program. Once the execution of the code is completed, the Driver program is exited and the Spark Context/Session is shut down.

A close up of text

Description automatically generated

* **DAG Visualization**

What is DAG?

In Apache Spark, a Directed Acyclic Graph (DAG) is a fundamental concept that **represents the logical execution plan of a Spark application**. It is a series of stages and tasks that describe the flow of data and the sequence of transformations to be applied to that data. The DAG is automatically created by Spark’s Catalyst Optimizer based on the high-level Dataframe or RDD transformations defined in your Spark code.

The DAG breaks the transformations into a series of stages. A stage represents a set of transformations that can be executed together in a pipelined manner, meaning the output of one transformation becomes the input of the next transformation without materializing the entire intermediate result. Stages are determined based on shuffling boundaries, which occur when data needs to be re-partitioned or shuffled across the cluster.

Within each stage, there are multiple tasks. A task is the smallest unit of work in Spark, representing the actual computation to be performed on a partition of data.

Understanding the DAG and monitoring its structure can help you identify potential performance bottlenecks in your Spark application. Tools like Spark UI and Spark History Server provide insights into the execution plan, allowing you to optimize and fine-tune your code.

Once the Logical and the best Physical Plan selected, it’s the time to generate the executable code (DAG of RDDs) for the query that is to be executed in a cluster in a distributed fashion. This process is called Codegen and that’s the job of Spark’s Tungsten Execution Engine.

1. Spark creates DAG.
2. Once action is triggered on the RDD, DAG is submitted to the DAGScheduler.
3. DAGScheduler looks at RDD lineage and comes up with the best execution plan by dividing it into stages of the task
4. Stages set is given to TaskScheduler and TaskScheduler will launch tasks through Cluster manager.
5. TaskScheduler with the help of cluster manager will check the data/resources availability in different nodes to execute the tasks. It will distribute the tasks to different executors.

\*DAG is read from the top down.\*

* **Spark for Parallel Processing**

ASK yourself:

* Can you open 1 GB of data in excel and find the average of a particular column?
* Can you run SQL script on 1 PB of data in MSSQL Server?

No, we can’t.

So, in order to take the advantage of available CPU and memory and process huge volume of data, a new technology emerged, parallel processing.

**Parallel Processing is a method in computing of running more than one CPU to handle separate parts of the whole task.**

Suppose we have 1 GB of data and we want aggregation of a particular column. In order to process this huge amount of data:

we split this data into several chunks/partitions, lets say 4, each of 250 MB.

then we have to write the code to be executed in parallel or convert code/script into parallel format

after this aggregation will be done.

As mentioned above, 4 different partitions can have different workstation/machine, or e.g., 4 HDDs, each partition have 1 HDD. But it is not feasible as it costs a lot for developers.

So, Spark is a technology that is going to get data and resolve this issue as it is going to run code over the available memory.

**Spark is a general purpose distributed computation engine. In layman’s terms, it can run across servers in a organized way and they can read distributed data and process that data based on code you have written to run within the spark engine.**

* **Overview of Spark Architecture**

Lets take 1 GB of data.

1. We have to send this data to Spark Engine. Spark Engine has a component called SparkContext or driver node, which is going to get data and put it in the cluster. SparkContext divides the whole data into no. of partitions(the number of partitions is decided by the user, suppose 4 partitions). Generally, spark itself runs job in parallel but if you still want parallel execution in the code you can use python/scala program for parallel processing to do it.
2. For a Cluster, we must know the no. of cores (Suppose 2 cores) and memory(RAM) available. Memory is shared between cores. These cores are called Working nodes. The job of Worker nodes is to execute the task and return the results to driver node
3. There is another component called Cluster manager/ Resource manager. It keeps tracks of the nodes available in the cluster.
4. Now, 4 partitions are going to divide equally between 2 cores. 2 partitions per core. And these partitions are going to reside in memory because memory is much faster than HDDs.

A diagram of a computer server

Description automatically generated

**Steps involved in executing jobs/task in Spark:**

Suppose, you have written a python program to calculate the average of a specific column.

1. Python code is send to the SparkContext
2. SparkContext asks the cluster manager to run the program over 4 partitions, and cluster manager ask the cores to be ready for processing as processing is going to be done in a specific time period.
3. Partitions are going to run parallelly. Partitions needs data structures to be stored in a memory. Some of the data structures which are widely used are RDD, DataFrames, Datasets etc.

* Partitions 1 & 3 are going to be executed in parallel
* Then partitions 2 & 4 are going to be executed in parallel.

1. After execution, aggregation is either done in the working nodes or sent back to the driver node to perform aggregation according to the code.

This is how Spark works in parallel processing!!

* **common performance problems when using Spark**
* **The 5 Ss**

The 5 Ss (**Spill, Skew, Shuffle, Storage, Serialization**) are the 5 most common performance problems in Spark. Two key general approaches which can be used to increase Spark performance under any circumstances are:

• Reducing the amount of data ingested.

• Reducing the time Spark spends reading data (e.g. using Predicate Pushdown with Disk Partitioning/Z Order Clustering).

We will now dive into each of the problems associated with the 5 Ss.

* **Spill**

Spill is caused by writing temporary files to disk when running out of memory (a partition is too big to fit in RAM). In this case, an RDD is first moved from RAM to disk and then back to RAM just to avoid Out Of Memory (OOM) errors. Disk reads and writes can although be quite expensive to compute and should therefore be avoided as much as possible.

Spill can be better understood when running Spark Jobs by examining the Spark UI for the Spill (Memory) and Spill (Disk) values.

• **Spill (Memory):** the size of data in memory for spilled partition.

• **Spill (Disk):** the size of data on the disk for the spilled partition.

Two possible approaches which can be used in order to mitigate spill are **instantiating a cluster with more memory per worker or increasing the number of partitions** (therefore making the existing partitions smaller).

* **Skew**

When using Spark, data is commonly read in evenly distributed partitions of 128 MB. Applying different transformations to the data can then result in some partitions becoming much bigger or smaller than their average.

Skew is the result of the imbalance in size between the different partitions. Small amounts of Skew can be perfectly acceptable but in some circumstances, Skew can result in Spill and OOM errors.

Two possible approaches to reduce Skew are :

• Salting the skewed column with random numbers to redistribute partition sizes.

• Using Adaptive Query Execution (Spark 3).

A graph of different colored bars

Description automatically generated with medium confidence

Figure 2: Partition Size Distribution Before and After Skew

* **Shuffle**

Shuffle results from moving data between executors when performing wide transformations (e.g. joins, groupBy, etc…) or some actions such as count. Mishandling of shuffle problems can result in Skew.

A diagram of a shuffle

Description automatically generated

Figure 3: Shuffling Process

Some approaches which can be used in order to reduce the amount of shuffling are:

• Instantiating fewer and larger workers (therefore reducing network IO overheads).

• Prefilter data to reduce its size before shuffling.

• Denormalize the datasets involved.

• Prefer using Solid State Drives over Hard Disk Drives for faster execution.

• When working with small tables, Broadcast Hash Join the smaller table. For big tables use instead SortMergeJoin (**Broadcast Hash Join** can lead to Out Of Memory issues with big tables).

* **Storage**

Storage issues arise when data is stored on disk in a non-optimal way. Issues related with storage can potentially cause excessive Shuffle. Three of the main problems associated with Storage are: Tiny Files, Scanning and Schemas.

• **Tiny Files**: handling partition files less than 128 MB.

• **Scanning**: when scanning directories we could either have a long list of files in a single directory or in the case of highly partitioned datasets multiple levels folders. In order to reduce the amount of scanning, we can register it as a table.

• **Schema**: depending on the file format used there can be different schema issues. For example, using JSON and CSV the whole data needs to be read to infer data types. For Parquet instead just a single file read is needed, but the whole list of Parquet files needs to be read if we need to handle possible schema changes over time. In order to improve performances, it could then help to provide schema definitions in advance.

* **Serialization**

Serialization encompasses all the problems associated with the distribution of code across clusters (the code is serialized, sent to the executors, and then deserialized).

In the case of Python, this process can even be more complicated since the code has to be pickled and an instance of the Python interpreter has to be allocated to each executor.

Serialization issues can arise when integrating codebases with legacy systems (e.g. Hadoop), 3rd party libraries, and custom frameworks. One key approach we can take to reduce serialization issues is avoiding using UDFs or Vectorized UDFs (which act like a black box for the Catalyst Optimizer).