* Spark Dataframe

- It is same as normal pandas dataframe in the form of table with each column has it's own data type.

DataFrame Schema

1. Column Name
2. Data Types

- DataFrame is a distributed data structure

Now, how does it help Spark to implement distributed processing?

⇒ Firstly, we know that SparkSession offers you a read() method to read data from CSV file

This CSV file is stored in a distributed storage such as HDFS or cloud storage (AWS S3)

All these distributed storage systems are designed to partition your data file and store the partitions across the distributed storage nodes.

distributed

So let's assume you have 10 node storage cluster. when you save your file on this cluster your file might be broken into 100 smaller partitions.

- One thing is clear, your data file is stored in as smaller partitions, and each storage node may have one or more partitions of your file.

- Next, you are going Read this file using Spark DataFrame Reader.
So basically, we are telling the driver that we want to read the data file, and the driver is going to reach out to cluster manager and storage manager to get the details about the data file partitions.

- So you can think of your Dataframe as a bunch of Smaller Dataframes, each logically representing a partition.

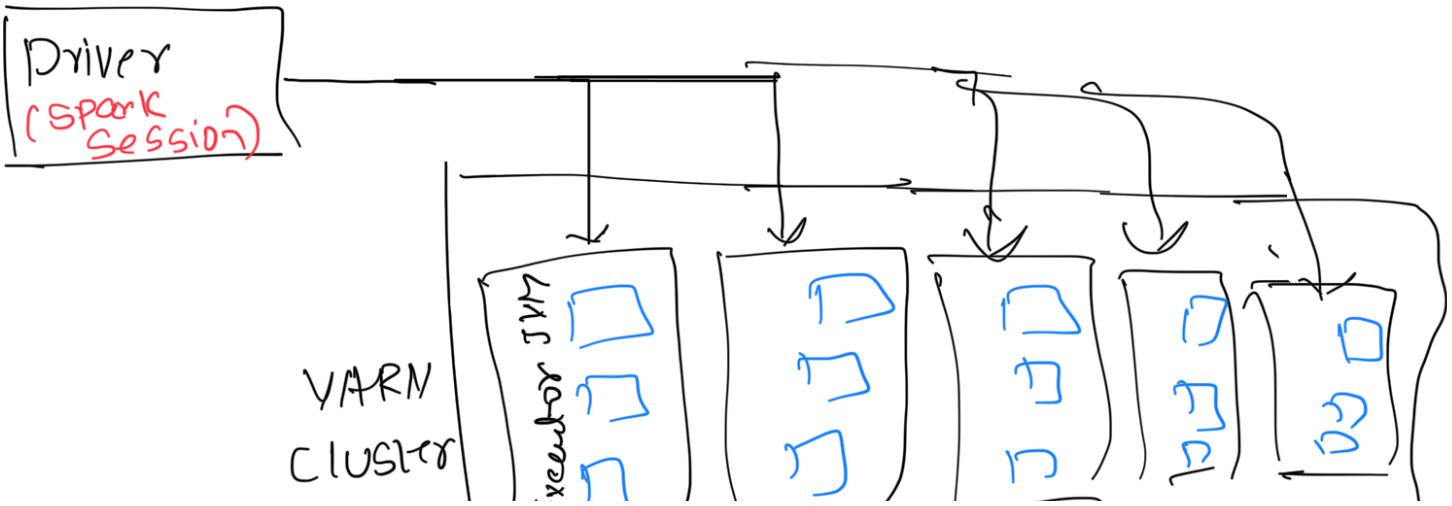
Now, let us shift our attention to Spark Executors.

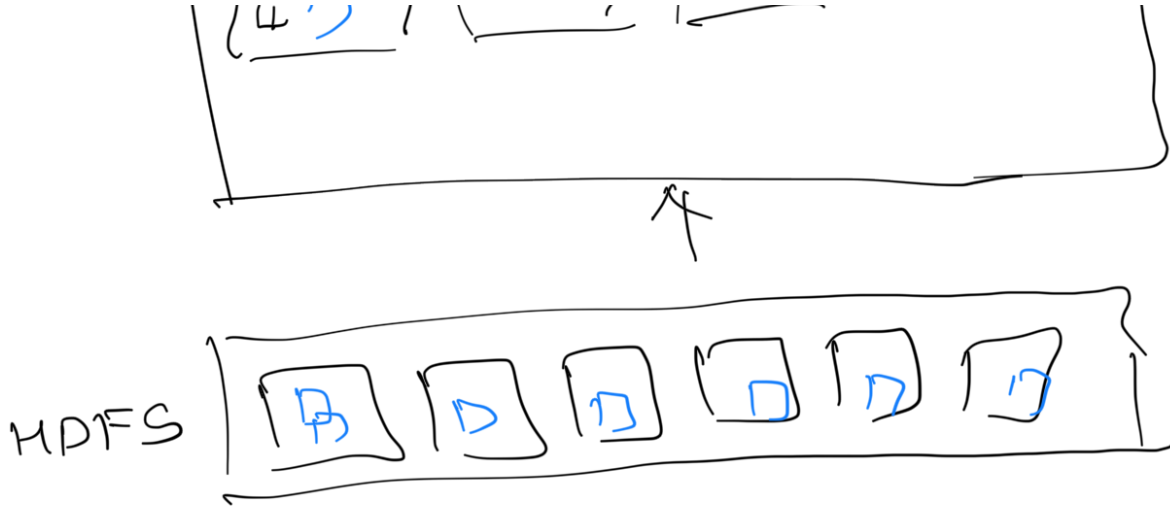
configured to start

Let's assume you have [illegible]
five executors, each with 10 GB memory
and 5 CPU cores.

Now, the driver is going to reach out
cluster manager and ask for the
containers.

Once, those containers are allocated,
the driver is going to start the
executors within those containers.
So, the driver is going to assign some
DataFrame partitions to each JVM
core and these executor cores will
load their respective partitions in
the memory.





* Spark Transformations And Actions

⇒ Spark Transformations :

Transformations are used to transform one Dataframe into another Dataframe without modifying the original Dataframe

For example :

The where() clause transformation.

It works on the input Dataframe and produces the output Dataframe.

- The initial input DataFrame remains immutable. And that's how the Spark Dataframe is an immutable data structure

- Spark Transformation is further classified into Two Types:

Transformation

(1) Narrow Dependency Transformation

(2) Wide Dependency Transformation

1] Narrow Dependency Transformation:

A transformation that can be performed on a single partition and still produce valid result is Narrow Dependency

For example:

where() clause transformation is a narrow dependency

Let's assume you have 2 partitions of a data frame.

If you apply a where() clause transformation on this data frame, spark executor will be able to perform this filtering on each partition. They do not depend on any other partition. And we still get a valid overall result on combining these filtered partitions.

This is what we mean by narrow dependency transformation. Such transformations are not dependent on other things, and it

can be easily accomplished by
executor on their partitions.

2] Wide Dependency Transformations:

— A transformation that requires data from other partitions to produce valid results

For example: A groupBy() transformation

Suppose we want to apply a groupBy() transformation and then count number of records in each group.

The Spark Executor will perform groupBy on each partition, but combining the results of each partition won't give us correct output.

— But, how can we fix it?

We can do that using repartitioning of grouped data.

It means,

combine all the partitions and then create some new partitions to make sure that all the records of the same group are collected into the

same partition.

This operation of combining and repartitions of data is caused by wide dependency transformation and known as shuffle and sort operation.

Now, you can apply the count() aggretion on these new partitions.

The count() aggregation remains a narrow dependency, because you combine the outcome of count() operation, and you get a valid result.

⇒ Spark Actions :

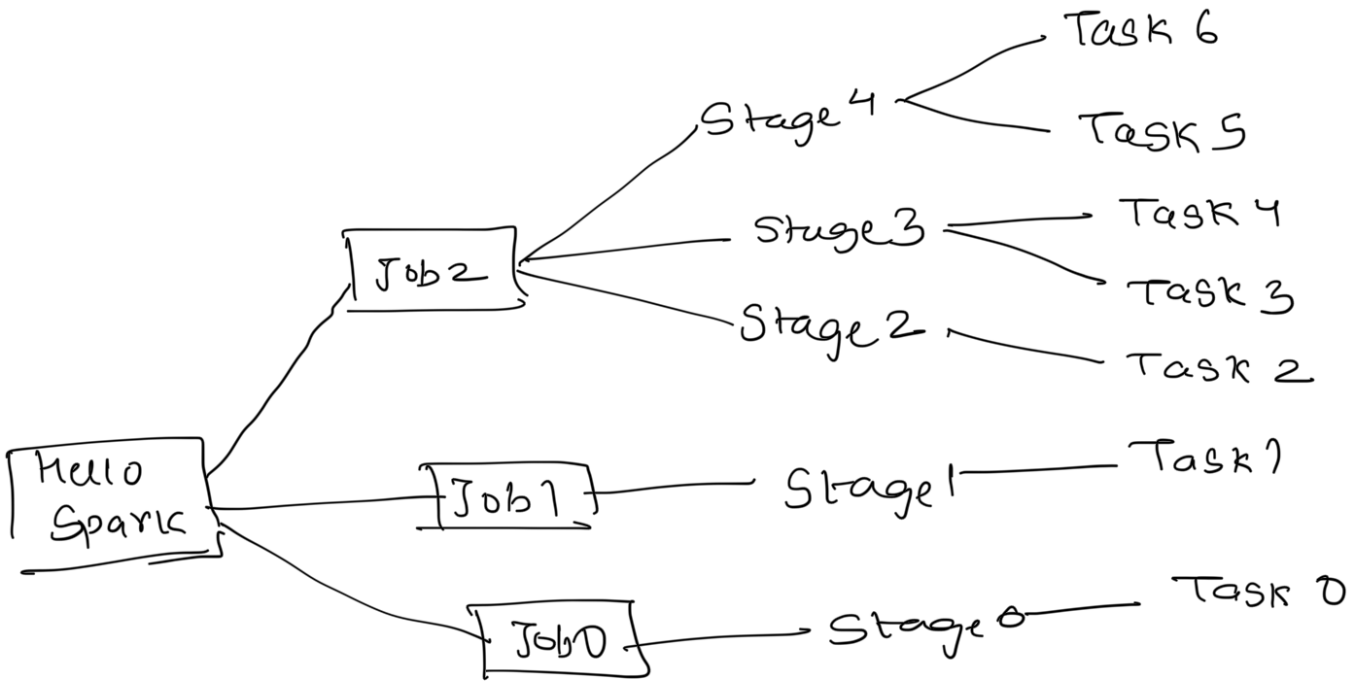
```
df = spark.read.csv("data.csv")
filtered_df = df.where("age<40")
selected_df = filtered_df.select("age",
                "gender", "country", "state")
grouped_df = selected_df.groupBy("country")
count_df = grouped_df.count()
count_df.show()
```

- Look at these transformations, a typical programer is going to look at these as individual statements that are executed and evaluated line by line.
- However, Spark Programs are not going to behave the same.
  Because we are using a builder pattern to create a DAG of transformations.
- All of this goes to Spark Driver. The driver is going to look at these operations, rearrange them to optimize certain activities, and finally create an execution plan which will be executed by the executors
- So, these statements are not executed as individual operations, but they are converted into an optimized execution plan which is terminated and triggered by an Action.
  So, the operations that require you to read, write, collect to show the data is an Action.

* Spark Job Stages and Task :

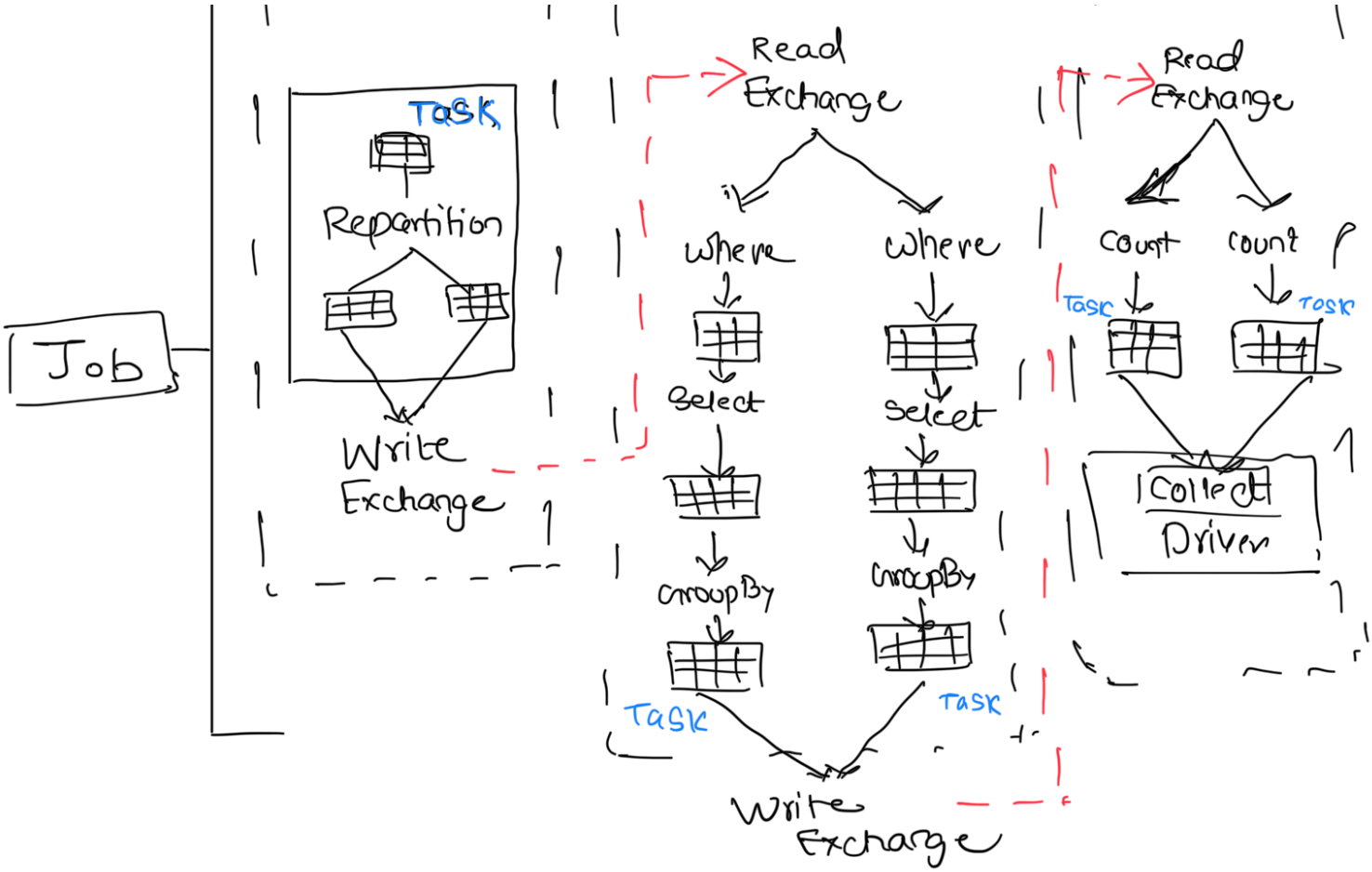
- Your Spark Application will internally generate a bunch of Jobs, Stages and tasks. And these tasks are the unit of work that is finally assigned to executors

Spark Execution Plan



(You can check this for your application on: localhost:4040)

⇒ Understanding Your Execution Plan



- In summary, each action will result in a Job. Each wide-transformation will result in a separate stage.
- And every stage executes in parallel depending upon the number of DataFrame partitions.
- The first stage operated on a single partition DataFrame, so we do not have any parallel processing here.
  The next two stages worked on two partitions, so we have two parallel tasks.