Spark DataFrame:

- **DataFrame** is a **distributed collection** of **data organized** into **named columns**. It is **conceptually** equivalent to a **table** in a **relational database**.
- DataFrame **appeared** in **Spark Release 1.3.0**.
- The idea behind DataFrame is it **allows processing** of a **large amount** of **structured data. DataFrame contains rows** with **Schema**.
- **DataFrame** in **Spark overcomes RDD** but **contains** the **features** of **RDD** as well.
- The features **common** to **RDD** and **DataFrame** are **immutability**, **in-memory** and **distributed computing capability**.
- It **allows** the **user** to **impose** the **structure** onto a **distributed collection** of **data.**
- We can **build DataFrame** from **different data sources**. For **e.g. structured data file**, **tables** in **Hive**, **external databases** or **existing RDDs.**

DF Reference to structure data like RDBMS:

Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

Data Frame Schema

1. Column Names
2. Data Types

Limitations of Spark RDD:

- It **does not** have any **built-in optimization engine.**
- There is **no provision** to **handle structured data.**
- It **does not** have **schema.**
- Thus to **overcome** these **limitations, DataFrame came** into **existence**.

Creating DataFrame in Apache Spark:

To **all** the **functionality** of **Spark**, **SparkSession class** is the **entry point.** For the **creation** of basic **SparkSession** just **use**:

sc → SparkContext
spark → SparkSession

Using **SparkSession**, an **application** can **create DataFrame** from an **existing RDD**, **Hive table** or from **Spark data sources.** Using **Spark SQL DataFrame** we **can create** a **temporary view** and run **SQL queries** on the **data.**

Convert RDD to DF:

- In PySpark, **toDF() function** of the **RDD** is **used** to **convert RDD** to **DataFrame**.
- We **need** to **convert RDD** to **DF** as **DF** provides **more advantages** over **RDD**.
- **DF** is a **distributed collection** of **data organized** into **named columns** similar to **DB tables** and **provides optimization** and **performance improvements**.

First, let's **create** an **RDD** by **passing Python list object** to **sparkContext.parallelize function**. We would need this "**rdd**" **object** for all our examples below.

A **list** is a **data structure** in **Python** that's **holds** a **collection** of **items**. **List items** are **enclosed** in **square brackets**, like this ***[data1, data2, data3]*.**

In PySpark, when you have **data** in a **list meaning** you **have** a **collection** of **data** in a PySpark **driver memory** when you **create** an **RDD**, this **collection** is **going** to be **parallelized**.

1) Using rdd.toDF ( ) function:
PySpark **provides toDF ( ) function** in **RDD** which **can** be **used** to **convert RDD** into **Dataframe**.

Note: By default, **toDF ( )** function **creates column names** as **"_1" and "_2".** This snippet yields below schema.

toDF ( ) has **another signature** that **takes arguments** to **define column names** as **shown below**.

2) Using PySpark createDataFrame ( ) function:
SparkSession class provides **createDataFrame ( )** method to **create DataFrame** and it takes **rdd object** as an **argument** and **chain** it with **toDF ( )** to specify **names** to the **columns**.

3) Using createDataFrame ( ) with StructType schema:

- When you **infer** the **schema**, **by default** the **datatype** of the **columns** is **derived** from the **data** and set's **nullable** to **true** for **all columns**.
- We can **change** this **behavior** by **supplying schema** using **StructType** – where we **can specify** a **column name**, **data type** and **nullable** for **each field/column**.

4) Create DataFrame from Data sources:

- In **real-time mostly** you **create DataFrame** from **data source files** like **CSV**, **Text**, **JSON**, and **XML** etc.
- PySpark **provides csv** ("path") on **DataFrameReader** to **read** a **CSV file** into **PySpark**

DataFrame and **dataframeObj.write.csv** ("path") to **save** or **write** to the **CSV file**.

- PySpark **supports** reading a **CSV file** with a **pipe**, **comma**, **tab**, **space**, or any **other delimiter/separator** files.

PySpark Read CSV file into DataFrame:

- Using **csv** ("path") or **format** ("csv")**.load** ("path") of **DataFrameReader**, you can **read** a **CSV file** into a **PySpark DataFrame**.
- These **methods** take a **file path** to **read from** as an **argument**.

a) Reading CSV file & see the difference in data for headers & schema

Note: This example **reads** the **data** into **DataFrame columns "_c0"** for the **first column** and "**_c1**" for the **second** and **so on** and by **default data type** for **all these columns** is **treated** as **String**.

Using Header record for column names:
This **option** is **used** to **read** the **first line** of the **CSV file** as **column names**.
By **default** the **value** of this **option** is **False**, and **all column types** are **assumed** to be a **string**. **Not mentioning** this, the **API treats header** as a **data record**.

b) Reading the Header & see the difference in data:

c) delimiter:
delimiter option is **used** to **specify** the **column delimiter** of the **CSV file**.
By **default**, it is **comma (,) character**, but **can** be **set** to **any character** like **pipe**(|), **tab** (\t), **space** using this **option**.

d) inferSchema:
The **default value set** to this **option** is **False** when **setting** to **true** it **automatically infers column types** based on the **data**. Note that, it **requires reading** the **data one more** time to **infer** the **schema**.

e) Read multiple CSV files:
Using the **read.csv( ) method** you can also **read multiple csv** files, just pass **all file names** by **separating comma** as a **path**, for example:

f) Read all CSV files in a directory:
We can **read all CSV files** from a **directory** into **DataFrame** just by **passing directory** as a **path** to the **csv** ( ) method.

g) Reading CSV files with a user-specified custom schema:
If you **know** the **schema** of the **file ahead** and **do not want** to **use**

the **inferSchema** option for **column names** and **types**, use **user-defined custom** column names and **type** can be given using **schema** option.

h) Write PySpark DataFrame to CSV file:
Use the **write()** method of the PySpark **DataFrameWriter object** to **write** PySpark **DataFrame** to a **CSV** file.

While **writing** a **CSV file** you can **use several options**.
E.g. header to **output** the **DataFrame column names** as **header record**
and **delimiter** to **specify** the **delimiter** on the **CSV output file**.

Saving modes:
PySpark DataFrameWriter also has a **method mode ( )** to specify **saving mode**.
1) error: This is a **default option** when the **file already exists**, it **returns** an **error**.
2) ignore: Ignores write operation when the **file already exists**.
3) append: To **add** the **data** to the **existing file**.
4) overwrite: This **mode** is **used** to **overwrite** the **existing** file.

i) Select single & multiple columns from PySpark:
- You can **select** the **single** or **multiples column** of the **DataFrame** by **passing** the **column names** you **wanted** to **select** to the **select ( ) function**.
- Since **DataFrame's** are **immutable**, this **creates** a **new DataFrame** with a **selected columns**. **show ( )** function is **used** to **show** the **Dataframe contents**.

 # Single & Multiple Columns:
```
df.select("firstname").show()
df.select("firstname", "lastname").show()
```

 # Using Dataframe object name:
```
df.select(df.firstname, df.lastname).show()
```

 # Using col function:
```
df.select(col("firstname"), col("lastname")).show()
```

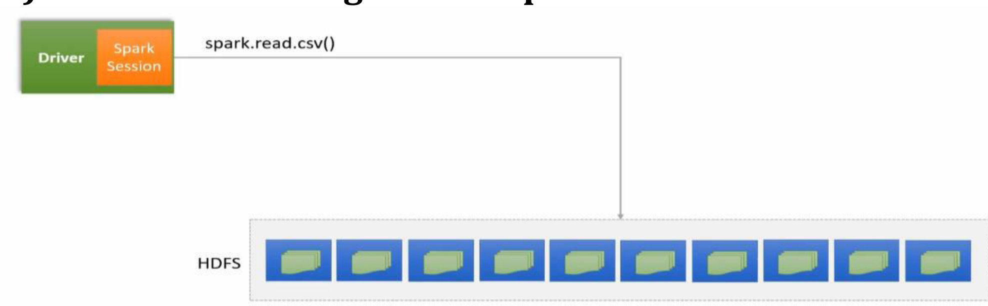
j) Select nested struct columns from PySpark:
If you have **struct (StructType) column** on PySpark **DataFrame**, you **need** to **use** an **explicit column qualifier** in **order** to **select**.

k) In order the get the specific column from a struct, you need to explicitly qualify.

l) In order to get all columns from struct column.

DataFrame Partitions & Executors:

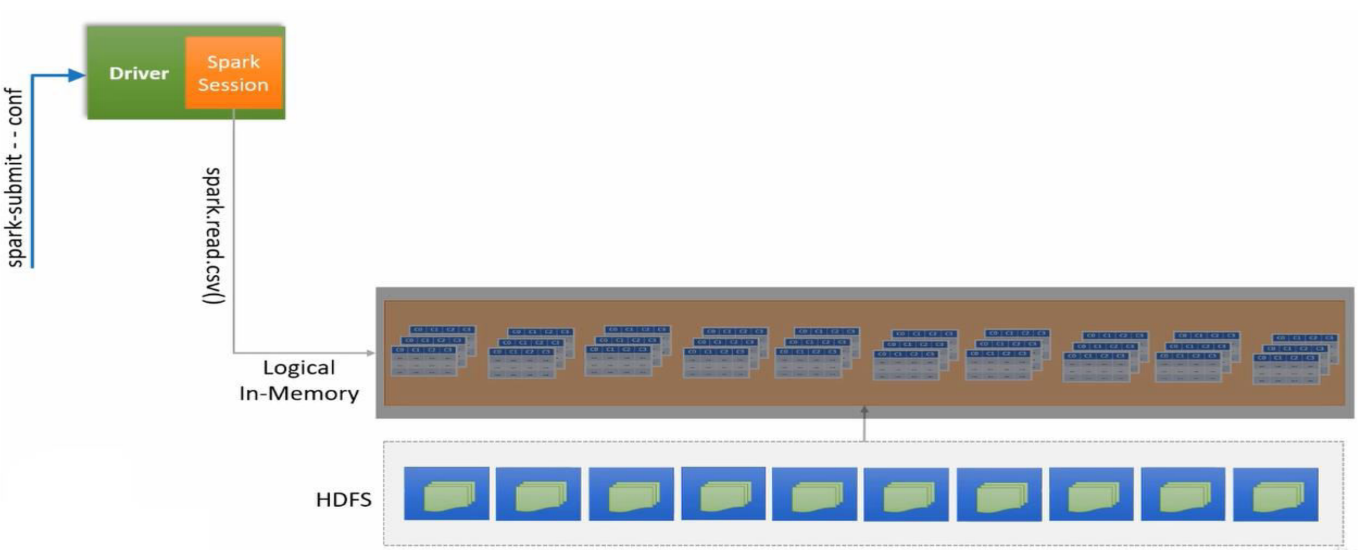
1) Distributed Storage in **multiple nodes**:



2) Brings the data **in-memory** in **partitions**:

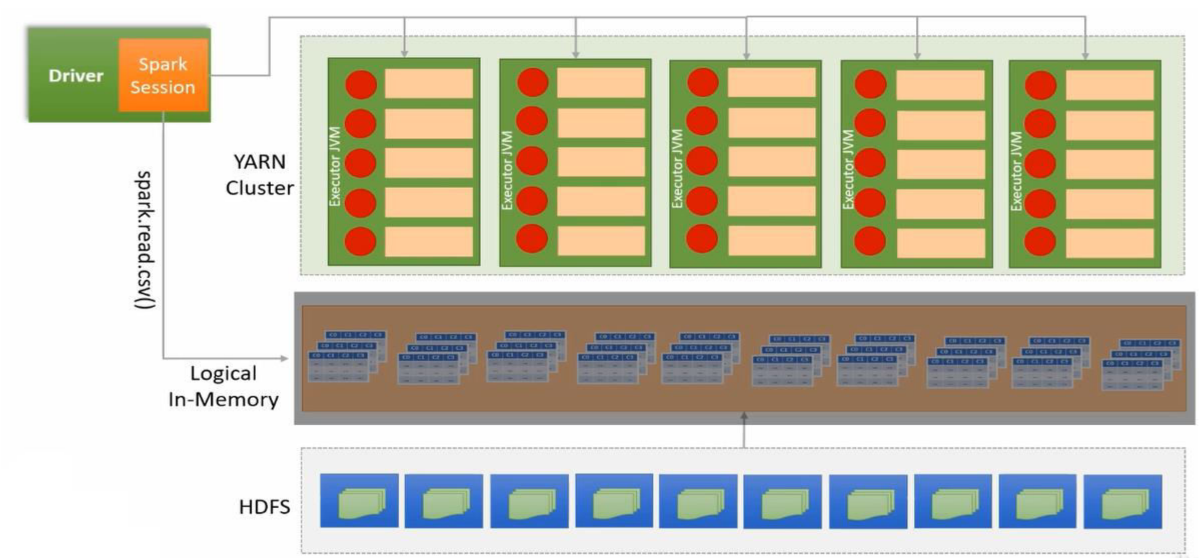
3)

- Driver reaches out to SM & CM to get the details of file partitions.
- So at runtime your driver know how to read the data files & how many partitions are there.
- So it creates logical in-memory structure which we see as a DF.



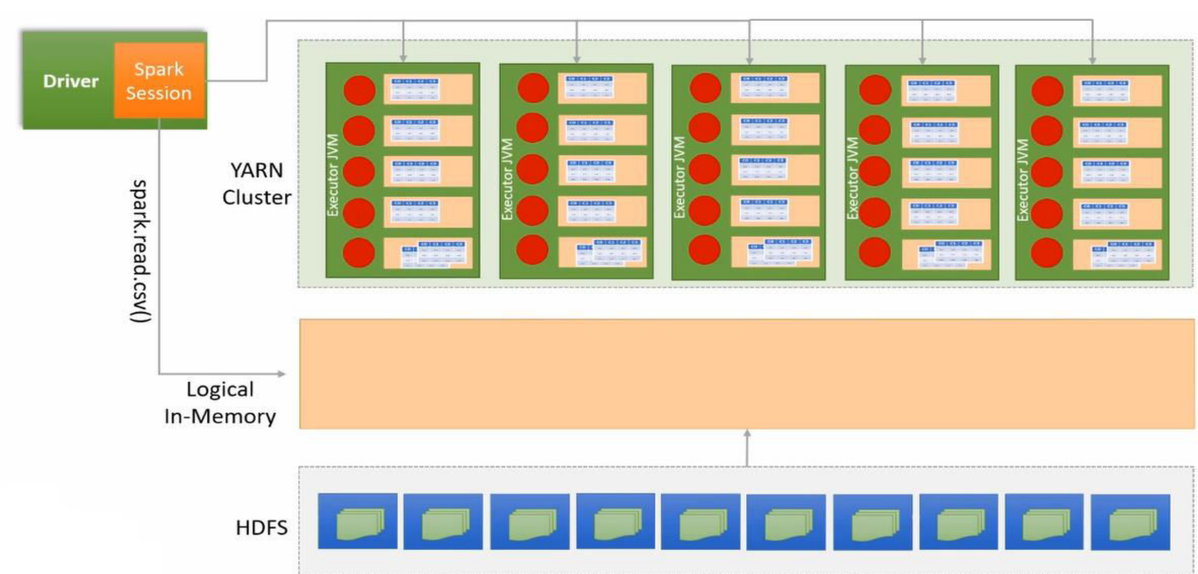
4)

- We can calculate how many executors, memory can be allocated to these executors which is given in your spark-submit.
- So all these configurations are available to your driver. Let's assume we configured to start 5 Executors each with 10GB memory and 5 Cores.
- Now the driver again will reach out to CM and ask for containers. One those containers are allocated the driver starts executors within these containers.
- Each executor is nothing but a JVM process with some assigned cores & memory.
- So here each executor is started with 5 executor cores & 10GB memory.

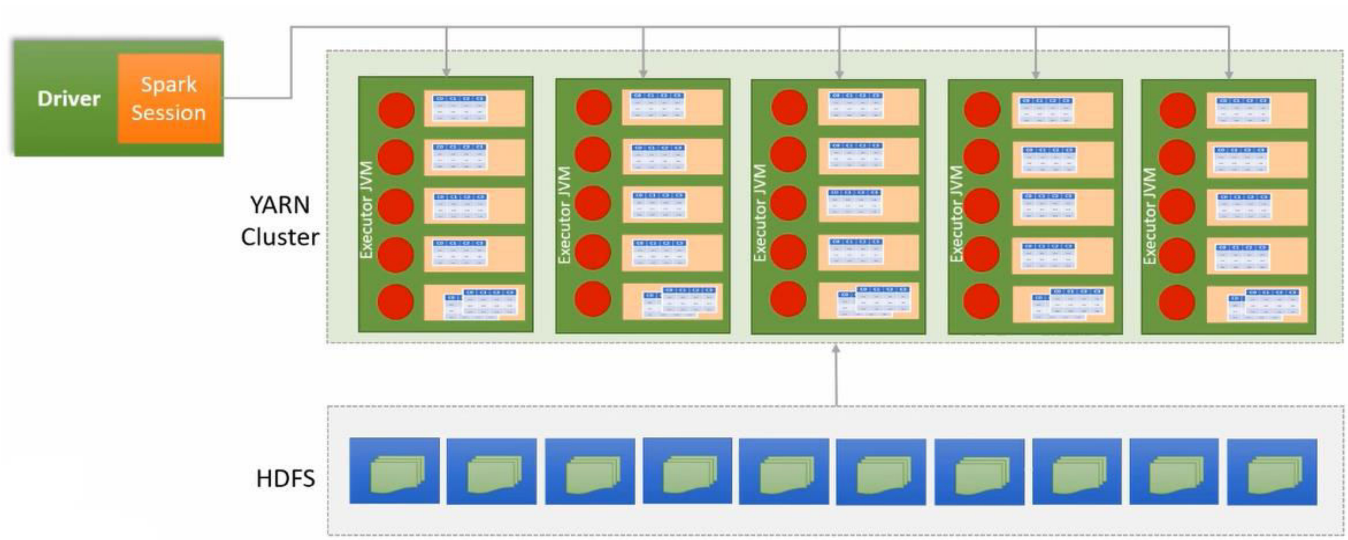


5)

- Now the Driver is ready to distribute the work to these executors.
- So driver assigns some DF partitions to each JVM core.
- These executor core will load their respective partitions in-memory.



6)
- Now you are ready with your distributed dataframe setup where each executor core is assigned its own data partition to work on.
- In all this process spark will also try to minimize the network bandwidth for loading data from physical storage to the JVM memory. How? That's the internal spark optimization.
- While assigning partition to these executors spark will try its best to allocate the partitions which are closed to the executors to the network.
- However such data locality is not always possible so spark & CM will work together to achieve best possible localization.



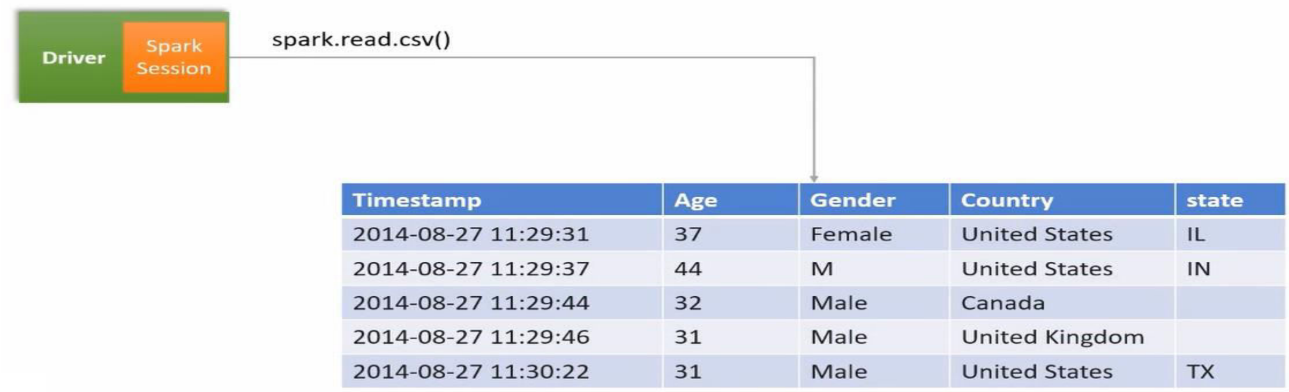
Transformations and Actions:

a) Transformations:
- Transformations are operations which will transform your RDD data from one form to another.
- When you apply this operation on any RDD, you will get a new RDD of transformed data (RDDs in Spark are immutable).
- Operations like **map, filter and flatMap** are transformations.

b) Actions:
- When action is triggered new RDD is not formed like transformations. Thus, actions are operation that gives non-RDD values.
- The values of action are stored to drivers or to the external storage system.
- It brings laziness of RDD into motion.
- An action is one of the ways of sending data from executer to the driver.
- Operations like **show, read, write, count, collect, save** are actions.

Transformations *(lazy)*	Actions
select	show
distinct	count
groupBy	collect
sum	save
orderBy	
filter	
limit	

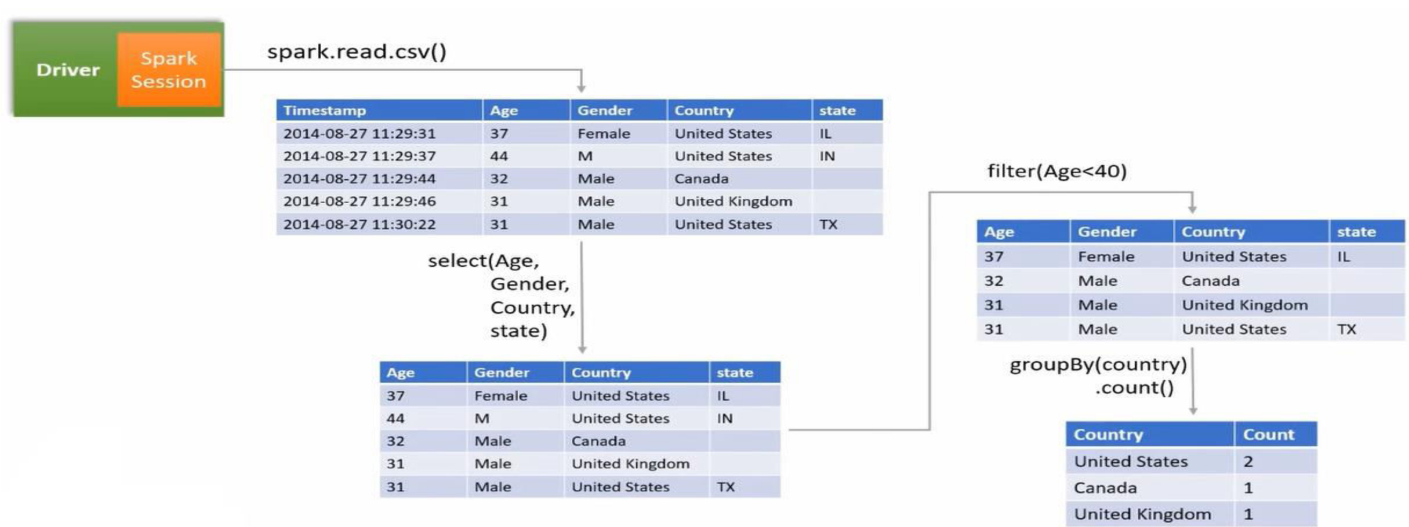
1)
When we read DF it looks like simple DF by hiding all the complex data distribution.



Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

2) Immutability:

- Spark DF are immutable however you can give instructions to your driver what you want to do & let driver decide how to achieve it with the executors.
- These instructions to the driver are called transformations & they can be select, filter, groupBy.



Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

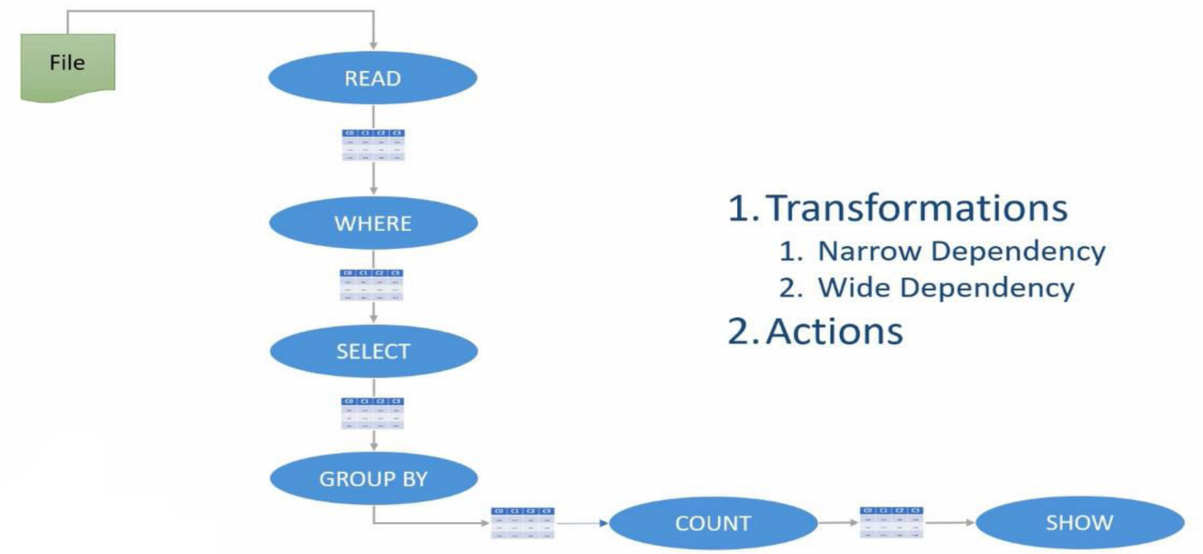
Age	Gender	Country	state
37	Female	United States	IL
44	M	United States	IN
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX

Age	Gender	Country	state
37	Female	United States	IL
32	Male	Canada	
31	Male	United Kingdom	
31	Male	United States	TX

Country	Count
United States	2
Canada	1
United Kingdom	1

3)

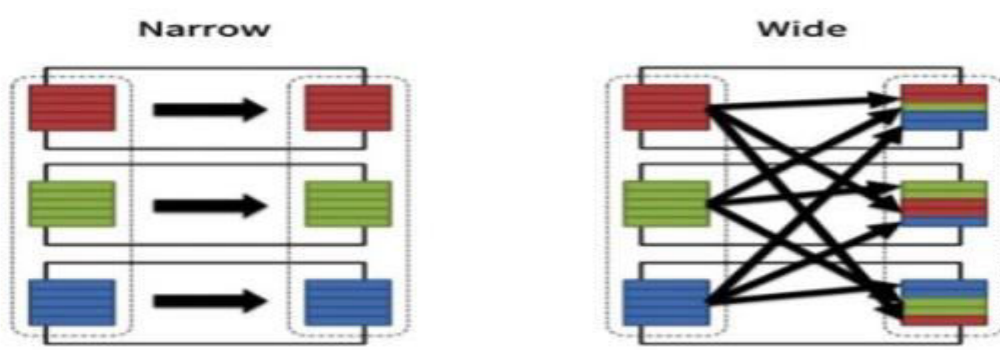
Graph of transformation operations which will create DAG.s



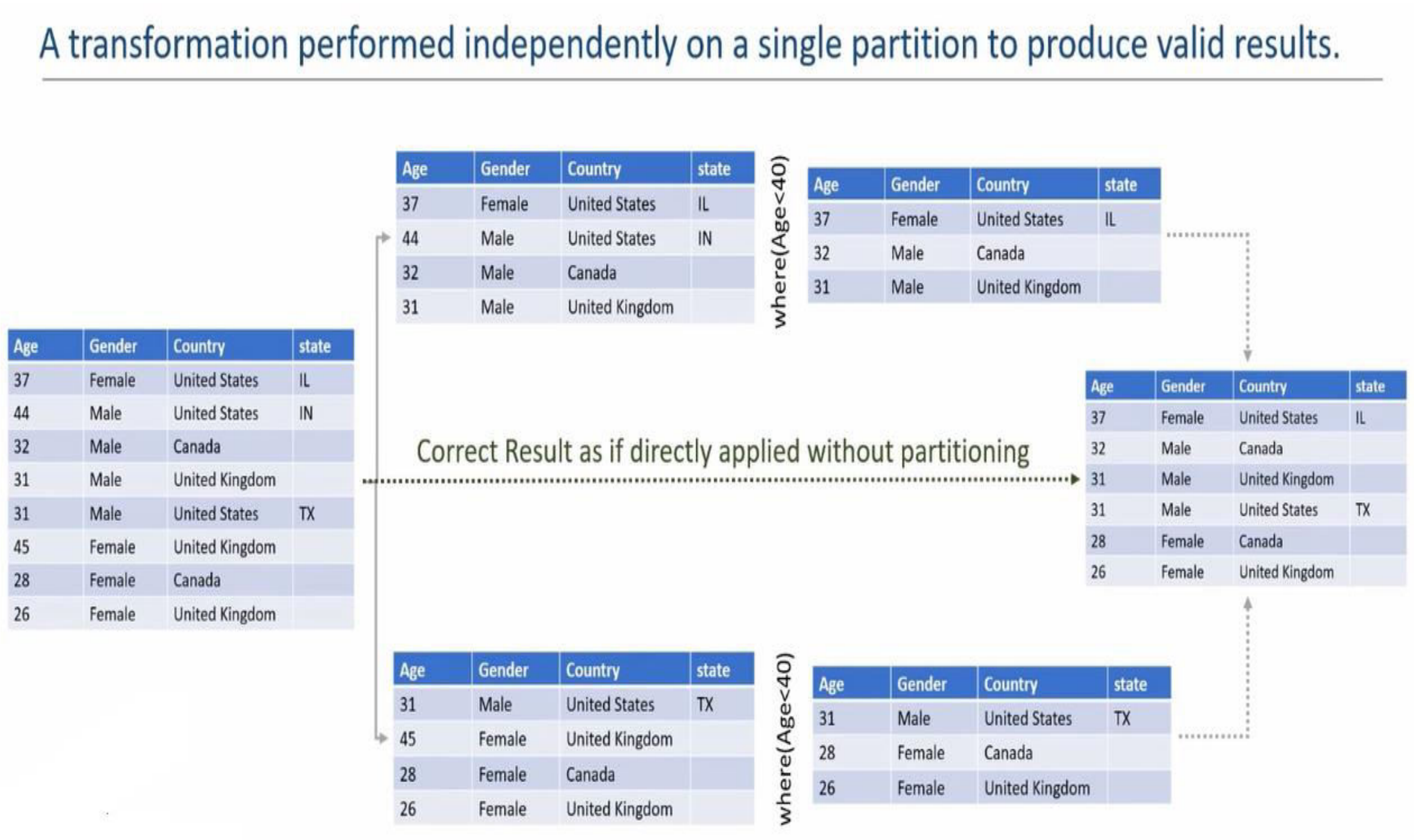
There are two types of transformations: **Narrow and Wide.**

a) Narrow: In Narrow transformation all the elements that are required to compute the records in a single partition lives in the single partition of parent RDD.
E.g. map ( ), filter ( ), coalesce ( ) etc.

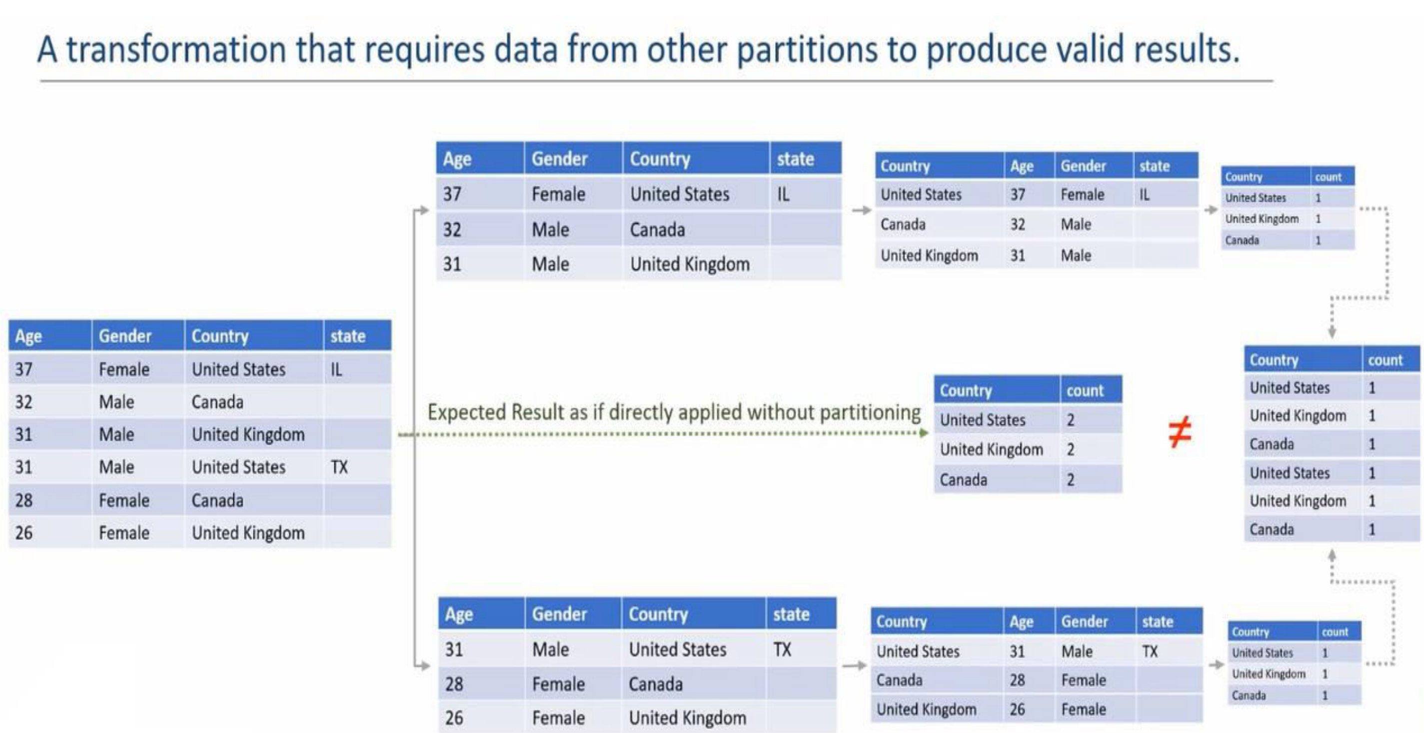
b) Wide: In wide transformation all the elements that are required to compute the records in a single partition lives in many partitions of parent RDD.
E.g. distinct ( ), groupBy ( ), repartition ( ) etc.



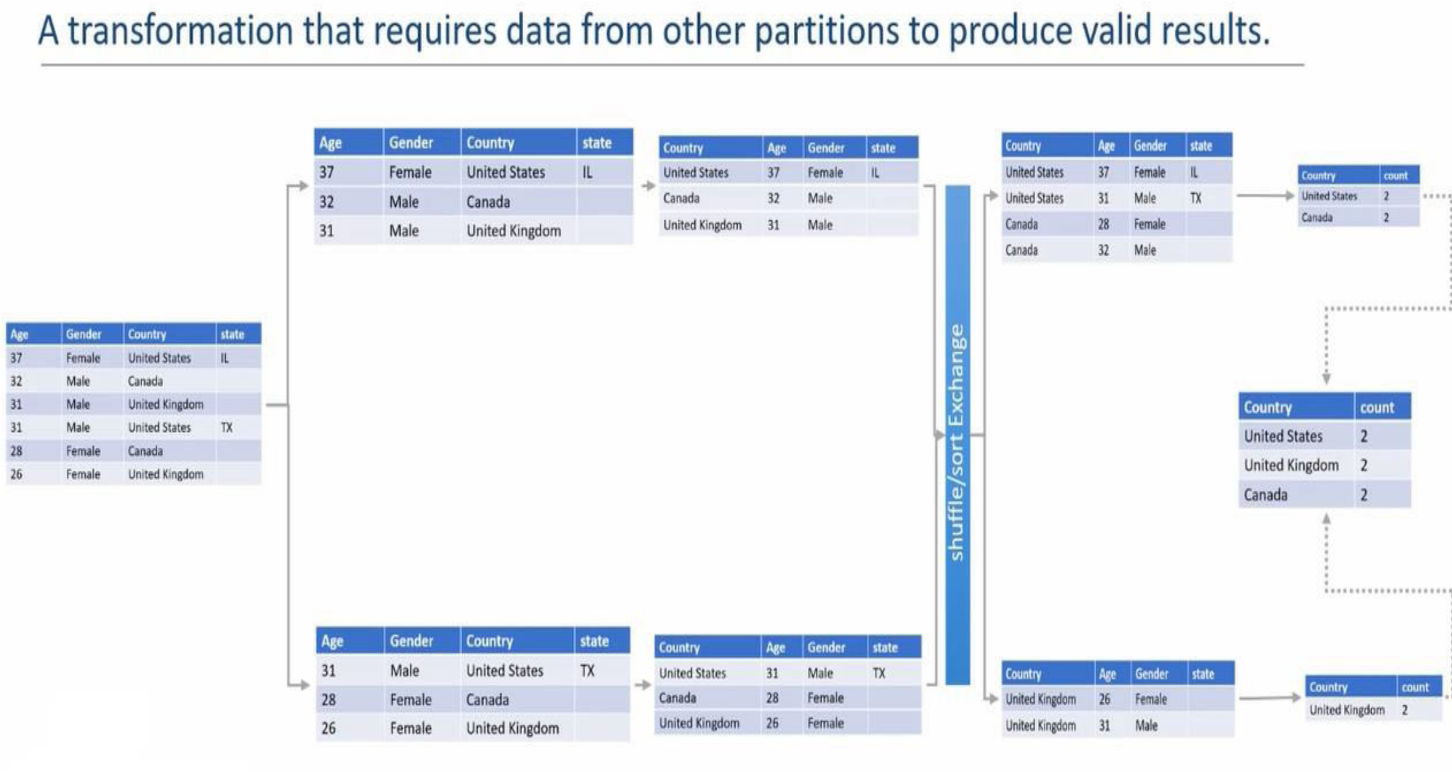
4) Narrow Transformation:



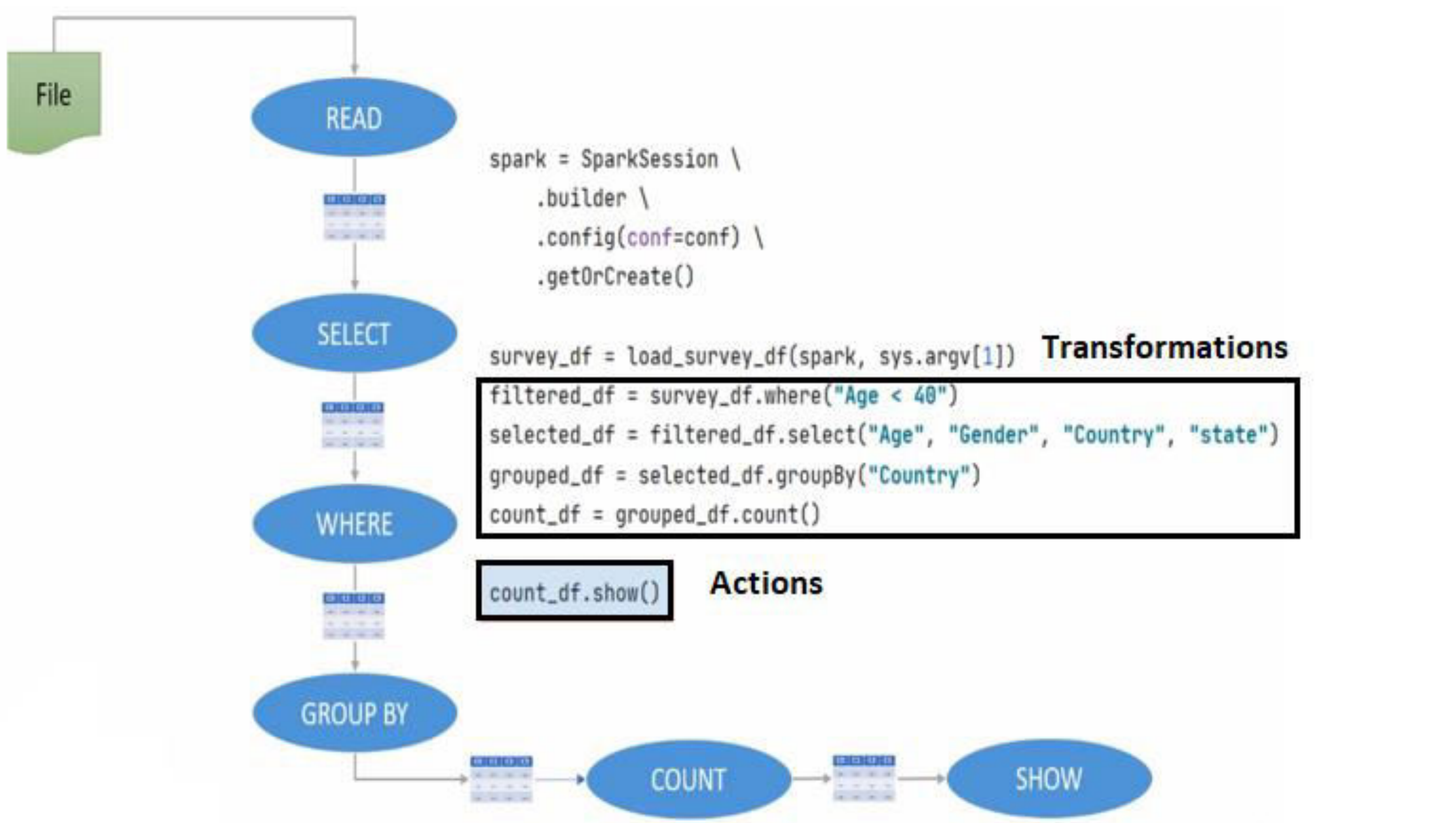
5) Wide:



6)



7) Lazy Evaluation



Spark Jobs, Stages & Tasks:

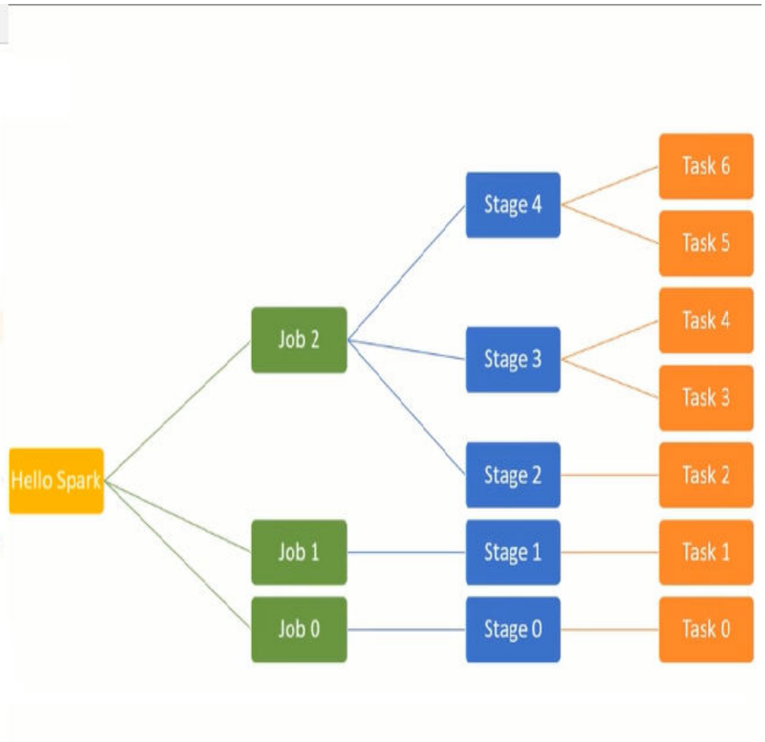
jobs_Stages_Tasks.py

```python
if __name__ == '__main__':
    spark = SparkSession.builder \
        .master("local[3]") \
        .appName("My First Program") \
        .config("spark.sql.shuffle.partitions", "2") \
        .getOrCreate()

    df = spark.read.format("csv") \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .load(input_path)
    partition_df = df.repartition(2)

    filter_df = partition_df.where("Age < 40")
    select_df = filter_df.select("Age", "Gender", "Country", "State")
    group_df = select_df.groupBy("Country")
    count_df = group_df.count()

    count_df.collect()
    input("Enter to Quit")
```

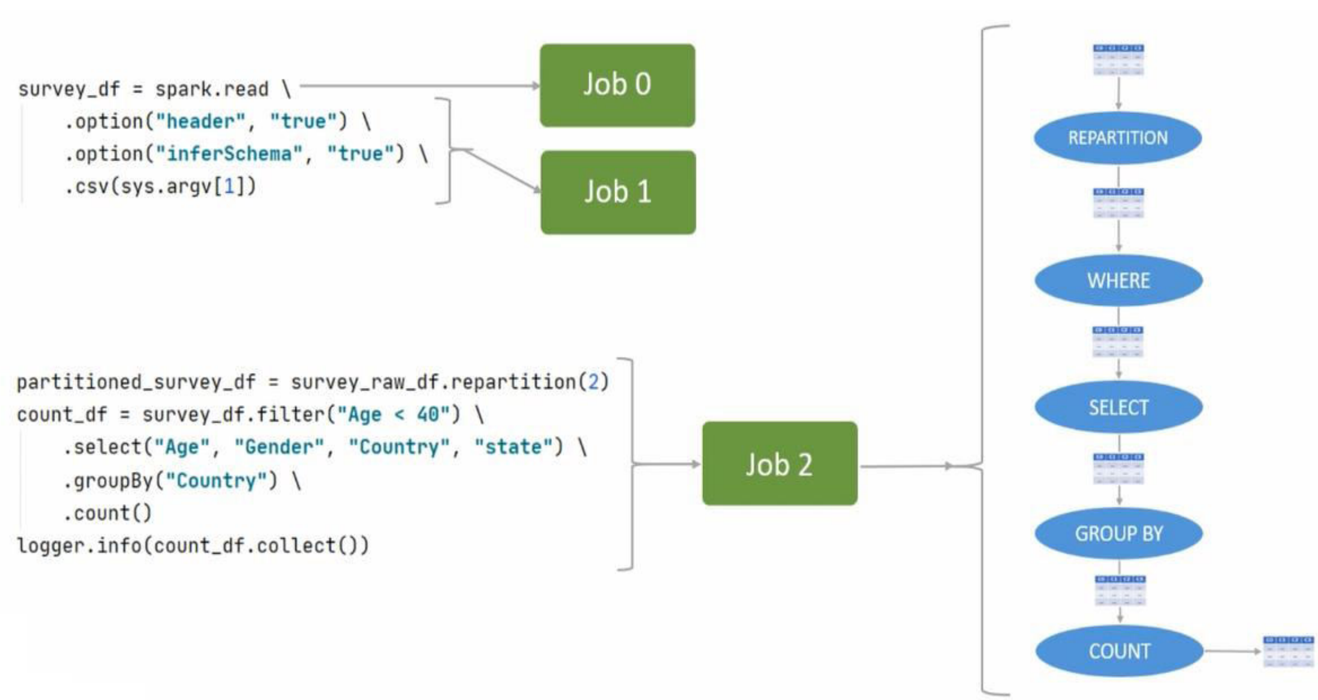


Understanding Your Execution Plan:

Step 1: Only Read csv file → 1 Job → 1 Stage → 1 Task

Step 2: Read Header & inferSchema → 1 Job 1 Stage → 1 Task

Step 3: Collect DF → 2 Jobs → 3 Stages → Tasks



DAG for collect action: