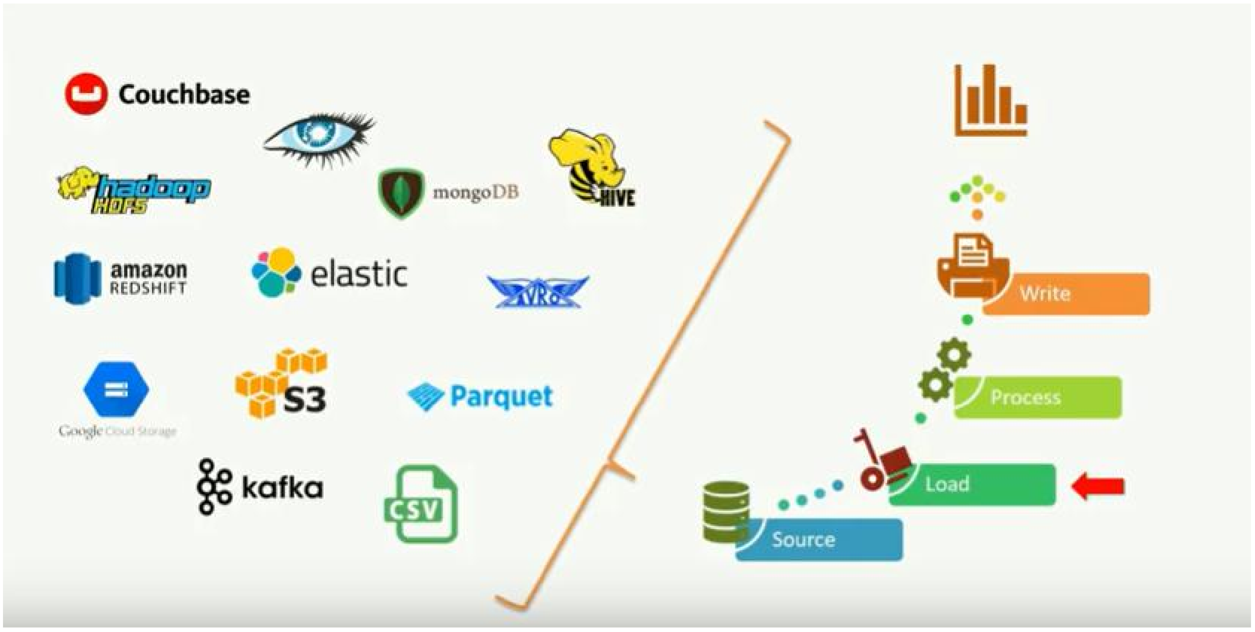
Apache Spark Foundation Course - Dataframe Basics

We had an introduction to Apache Spark and also learned Spark Architecture in the earlier videos. Now it is time to start getting into the Spark application development. Sometimes, it is quite convenient to break things into separate areas and narrow down our study focus. For that purpose, I can classify Apache Spark programming in following areas.

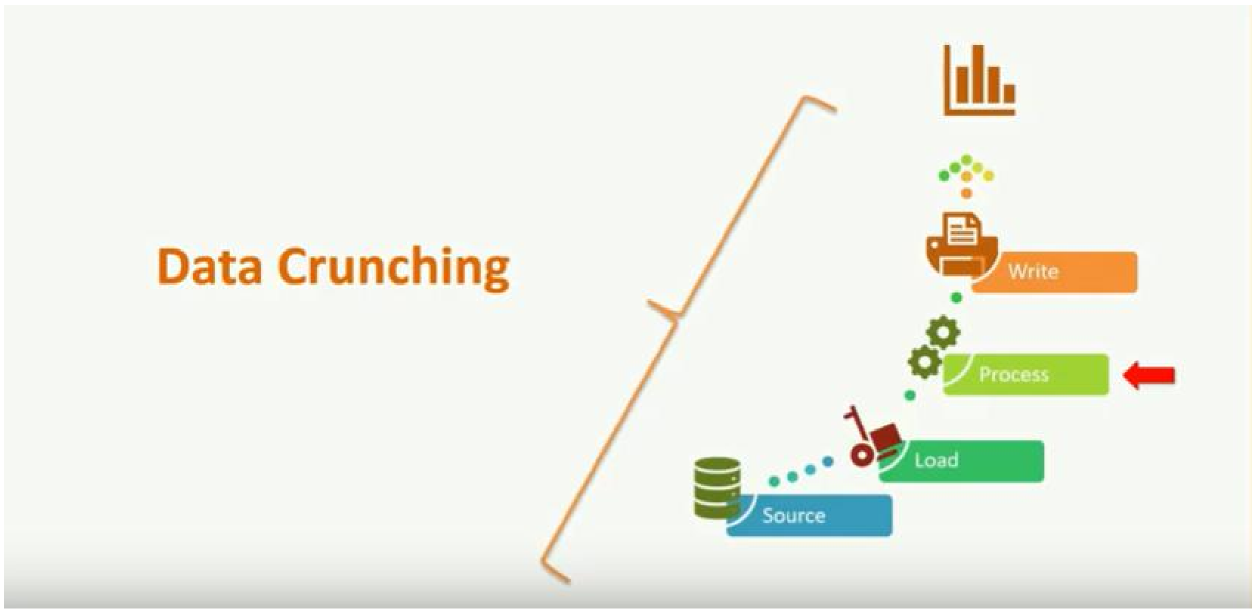
1. Batch Processing
2. Interactive Query
3. Streaming
4. Machine Learning
5. Graph processing

In this video and some of the upcoming videos, we will try to restrict ourselves to the batch processing in Apache Spark.
We also learned that every Spark application does three things.

1. Load
2. Process
3. Write



The first thing is to load the data from a source system. There are hundreds of possible source systems, even if we leave out the streaming sources, the number is still quite large. However, all of them comes under the scope of loading data into Apache Spark. We cannot cover all of them, but we will try to pick up some fundamental sources and add separate content for other sources over an extended period.



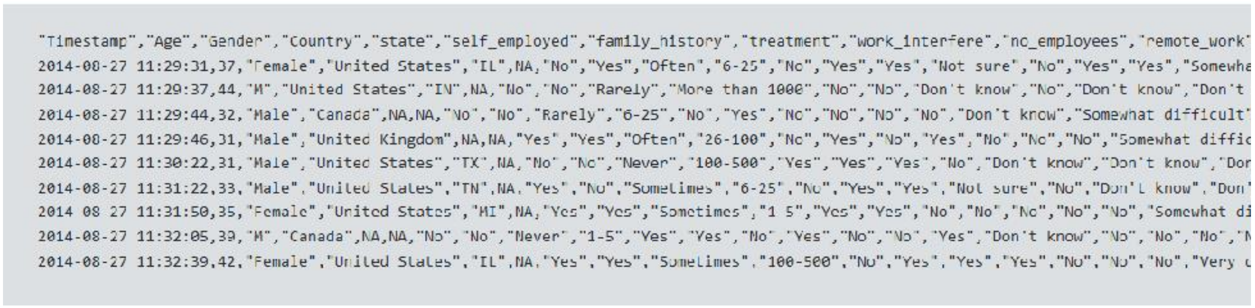
The final part is to write your outcome to a destination system. Again, there are hundreds of options to be considered for a destination system. We will cover some essential destinations types and add on few more at later stages.
In the middle, we have the data processing. It might be simple aggregation, grouping, and sorting or maybe some other complicated things. Whatever you do with the loaded data, It comes under the umbrella of processing the data. However, contrary to a large number of sources and destinations, we have just three options to hold the data in Apache Spark.

1. Spark Dataframe
2. Spark DataSet
3. Spark RDD

It may be the raw data that you loaded from a source system or the intermediate data, or it may be an outcome. Everything resides in one of these three structures.

Spark Dataframes

A Spark data Frame is a distributed collection of structured data. Since they hold structured data, you can think of them as a database table with a schema attached to it. Well, that is a lot to understand. Let's take an example. To create a data frame, we need to load some data. Let me download a sample data file from kaggle.com.
If you have a Kaggle account, you can also download the same data file as I am using for this video. Let's take a quick look at the data file.



```
"Timestamp","Age","Gender","Country","state","self_employed","family_history","treatment","work_interfere","no_employees","remote_work"
2014-08-27 11:29:31,37,"Female","United States","IL",NA,"No","Yes","Often","6-25","No","Yes","Yes","Not sure","No","Yes","Yes","Somewha
2014-08-27 11:29:37,44,"M","United States","IN",NA,"No","No","Rarely","More than 1000","No","No","Don't know","No","Don't know","Don't
2014-08-27 11:29:44,32,"Male","Canada",NA,NA,"No","No","Rarely","6-25","No","Yes","No","No","No","No","Don't know","Somewhat difficult"
2014-08-27 11:29:46,31,"Male","United Kingdom",NA,NA,"Yes","Yes","Often","26-100","No","Yes","No","Yes","No","No","No","Somewhat diffic
2014-08-27 11:30:22,31,"Male","United States","TX",NA,"No","No","Never","100-500","Yes","Yes","Yes","No","Don't know","Don't know","Don
2014-08-27 11:31:22,33,"Male","United States","TN",NA,"Yes","No","Sometimes","6-25","No","Yes","Yes","Not sure","No","Don't know","Don'
2014-08-27 11:31:50,35,"Female","United States","MI",NA,"Yes","Yes","Sometimes","1-5","Yes","Yes","No","No","No","No","No","Somewhat di
2014-08-27 11:32:05,39,"M","Canada",NA,NA,"No","No","Never","1-5","Yes","Yes","No","Yes","No","No","Yes","Don't know","No","No","No","N
2014-08-27 11:32:39,42,"Female","United States","IL",NA,"Yes","Yes","Sometimes","100-500","No","Yes","Yes","Yes","No","No","No","Very
```

I notice following things about the data.

1. It is a CSV file.
2. The first row is a header that contains the column names.
3. String values are surrounded by a pair of double quotes.
4. Null values are marked as *NA*
5. The timestamp is in the following format - *YYYY-MM-DD HH24: MM: SS*

These observations are good enough to load the data correctly. Now, all that we need is an API call to load this data into Apache Spark. However, Apache Spark is continuously evolving, and a lot of the APIs are still in experimental stage. So, I recommend you to refer the documentation for the version that you are planning to use. Let me take the opportunity to introduce you to the Apache Spark API documentation.
Go to the Apache Spark home page. Navigate to the latest release documentation. Choose API docs and then Scala.
If you prefer using Python, you can jump to the Python documentation as well. The good news is that the Data Frame APIs are synonymous in Scala and Python.

Spark Session

Once you open the Spark documentation, you might wonder, Where to start? A Spark application starts with a Spark Session. So, look for the Spark Session in the search bar.
A Spark session is the entry point to programming Spark with Data Frame APIs. We will be using an interactive client, so we already have a Spark Session available to us. Spark shell creates a Spark Session upfront for us. However, I will come back to Spark session builder when we build and compile our first Spark application. The most critical Spark Session API is the read method. It returns a Data Frame Reader. So let's jump to the Data Frame Reader.

Spark DataFrameReader

A Data Frame Reader offers many APIs. There is one specifically designed to read a CSV files. It takes a file path and returns a Data Frame. The CSV method could be the most convenient and straightforward method to load CSV files into a Data Frame. It also allows you to specify a lot many options. We use these settings to the tell the API about our source data file so that the API could interpret the file correctly. For example the header option. You can set the header option as TRUE, and the API knows that the first line in the CSV file is a header. The header is not a data row so that the API should skip the first row from loading.
Following other options are immediately relevant for our example.

1. **quote ->** Quote is the character used to enclose the string values. Quoting your string value is critical if you have a field that contains a comma. The default value is the double quote character, and hence we can rely on the default value for our example.

2. **inferSchema** -> Infer schema will automatically guess the data types for each field. If we set this option to TRUE, the API will read some sample records from the file to infer the schema. If we want to set this value to false, we must specify a schema explicitly.

3. **nullValue ->** The null value is used to define the string that represents a null.

4. **timestampFormat ->** This one is to declare the timestamp format used in your CSV file.

5. **mode ->** This one is crucial. It defines the method for dealing with a corrupt record. There are three supported modes.

PERMISSIVE, DROPMALFORMED, and **FAILFAST**.
The first two options allow you to continue loading even if some rows are corrupt. The last one throws an exception when it meets a corrupted record. We will be using the last one in our example because we do not want to proceed in case of data errors.
We can use the following code to load the data from the CSV file.

Spark Example to load a csv file in Scala

```
val df = spark.read.options( Map("header" -> "true",
                                 "inferSchema" -> "true",
                                 "nullValue" -> "NA",
                                 "timestampFormat" -> "yyyy-MM-dd'T'HH:mm:ss",
                                 "mode" -> "failfast")
                            ).csv("/home/prashant/spark-data/survey.csv")
```

Let me explain the code.
Spark is the variable name for the Spark Session object. The read method is an API defined over a Spark Session object. The read method returns an instance of Data frame Reader object, and hence we call the options API over a Data Frame Reader. The Options API takes a key value pair of all the options that we want to specify. Finally, we call the CSV method and provide the file location.
I am using Spark in local mode and hence I am giving the local file path. If you are trying to do it on a Hadoop cluster, you must move your file to HDFS and specify the HDFS file location. We will implement the next example in a Hadoop Cluster. I will show you the details for loading data from an HDFS location as well as from Gooogle Storage bucket.

Great. Copy the code. Go to Spark Shell. Activate the paste mode (type :paste). Paste the code. Press enter. Then press *Control+D* to close the paste mode. The value df is your Data Frame.

The method CSV is a convenient approach to load a CSV file. However, we have a more generic API structure for reading data. Here is an equivalent API call.

Spark API to load a csv file in Scala

```scala
val df = spark.read
              .format("csv")
              .option("header","true")
              .option("inferSchema","true")
              .option("nullValue","NA")
              .option("timestampFormat","yyyy-MM-dd'T'HH:mm:ss")
              .option("mode","failfast")
              .option("path","/home/prashant/spark-data/survey.csv")
              .load()
```

The above API structure is more generic and supports many different sources. Sometimes, it is a good idea to use a consistent universal API structure across your code instead of using one for CSV and another one for JSON.
If you compare it with the earlier API chain, you will notice following differences.

1. We are using a format method to specify the data source type. There are many formats available to us, and the community is working to add a lot more. Some of my favorites are CSV, JSON, parquet, JDBC, Kafka, and Cassandra.
2. In the earlier API call, we created a MAP of all the options and passed them at once. You can do that in this API call as well. However, I prefer to supply each option individually. It is just a personal choice.
3. You might notice that I provided the file location as an option. However, the load method can also take the path as an argument. Which one to use is again a personal preference.

We learned two Spark Classes.

1. SparkSession
2. DataFrameReader

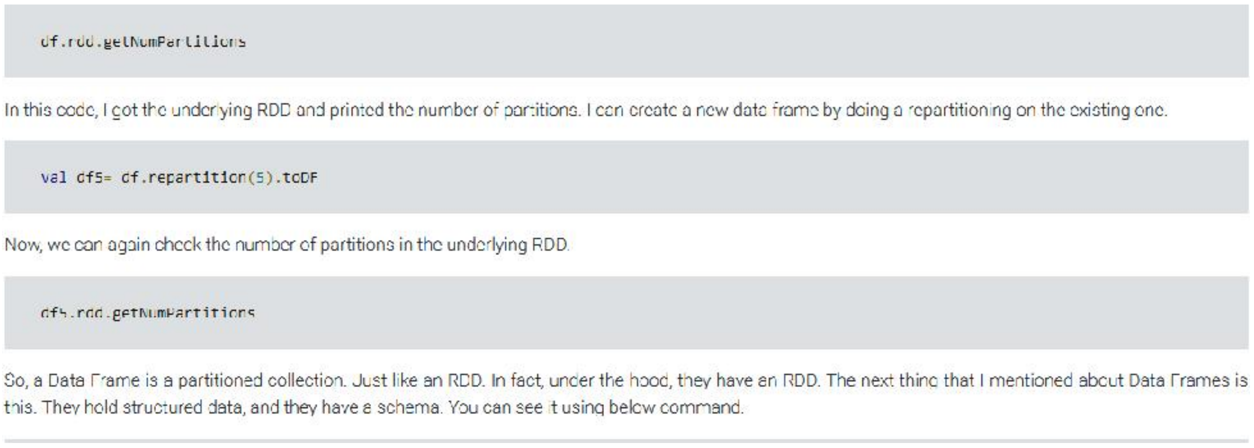
I recommend you to spend some time reading the documentation for these two classes.

Where is DataFrame documentation?

Surprisingly, there is no Scala Class for Data Frame. You might find it in Python documentation, but in Scala, Data Frame is not a class. A Scala Data Frame is a data set of the following type.
type DataFrame = Dataset[Row]
So, we have to look into the DataSet class. All the methods available to a DataSet is also available to the Data Frames. Well, A hell lot of methods, But don't worry about them because 80% of your requirement can be satisfied by just a handful of these APIs, and I will cover the most critical ones.

Final point on Spark DataFrames

A Spark data Frame is a distributed collection of structured data. Since they hold structured data, you can think of them as a database table with a schema attached to it.
I said that a data frame is a distributed collection. Let's try to understand that.
I have already covered some basics about RDDs in the earlier video. You already know that the RDD is a partitioned collection. A data frame is nothing but an abstraction on top of RDD. You can think of it as a simplified and optimized RDD for structured data. Since an RDD is broken down into partitions, and Data Frame is just an abstraction over RDD, hence a data frame is also partitioned. You can bring that fact to your terminal using the below code.



```
df.rdd.getNumPartitions
```

In this code, I got the underlying RDD and printed the number of partitions. I can create a new data frame by doing a repartitioning on the existing one.

```
val df5= df.repartition(5).toDF
```

Now, we can again check the number of partitions in the underlying RDD.

```
df5.rdd.getNumPartitions
```

So, a Data Frame is a partitioned collection. Just like an RDD. In fact, under the hood, they have an RDD. The next thing that I mentioned about Data Frames is this. They hold structured data, and they have a schema. You can see it using below command.

```
df.select("Timestamp", "Age","remote_work","leave").filter("Age > 30").show
```

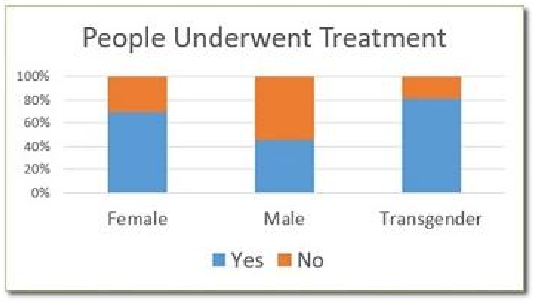
Does it look like a select statement on a database table? The data frame follows a row-column structure like a database table. You can pick up the desired columns and apply filters on the rows. Every column has a name and data type attached to it. You can print the Data Frame schema using the below command.

```
df5.printSchema
```

It shows the list of all the columns. Every column has a name and a data type attached to it. We also have a constraint. Great. In this session, I gave you a formal introduction to Spark Data Frames. We also learned a primary method to load data into Spark Data Frames. In the next video, we will deep dive further into Data Frames.

In the earlier video, we started our discussion on Spark Data frames. In this video, we will deep dive further and try to understand some internals of Apache Spark data frames. I am assuming that you have already watched the Architecture video. We covered some RDD basics in that lesson. You might find that half of this video is reiterating the RDD concepts. And that happens because a data frame ultimately compiles down to an RDD. However, we will go ahead of those notions and also cover things that are specific to data frames. So, Let's start.

We already learned a method to load data from a source. What do you want to do next?
If you have a business requirement, you might want to do some data processing and generate an output to meet your requirement. Let me give you a simple requirement. We already loaded mental health survey dataset. Can you generate following bar chart using that dataset?



Gender	Yes	No
Female	170	77
Male	450	541
Transgender	17	4

Fig.1-Spark Dataframe Example Graph and Table.

Well, we don't want to get into the visualization so let's reduce the requirement to an output dataset. The table represents the final output that we want to achieve. I created the Bar Chart in MS Excel using the above table.
Does it appear to be an easy requirement? Well, It is indeed an easy example. We are going to solve this problem, and while we develop a solution, you will also

discover some important data frame concepts.
We already learned that RDDs are immutable. Once loaded, you cannot modify them. However, you can perform following types of operations on RDDs.

1. Transformations
2. Actions

Spark data frames carry the same legacy from RDDs. So, Spark data frames are also immutable. But you can perform transformations on them to generate new data frames. You can also perform actions on a Spark data frames. If you check the documentation, you will see two categories of transformations.

1. Typed Transformations
2. Untyped Transformations

Both of these are available to data frames. The untyped transformations might return you a dataset. But you can convert a dataset to a data frame using the *toDF* function. However, you don't need to worry too much about it because Spark can take care of that automatically.
Transformations are the basic building blocks for Spark developers. They are the key tools for the developers because we use transformations to express our business logic.
Let's come back to our example. What do we want to do? If you check the dataset . You will see a field for treatment. This field records the response to the following question.
Have you taken a treatment for your mental illness?
Some people responded as *Yes* and others as *No*. All we need to do is to count these *Yes* and *No*. Then group them by the gender. That's it. You will get the output table. We can meet this requirement by applying a set of transformations. Let's try that.
The first activity is to load the data into a DataFrame. You can use below code to load the data.

```
val df = spark.read
            .format("csv")
            .option("header","true")
            .option("inferSchema","true")
            .option("nullValue","NA")
            .option("timestampFormat","yyyy-MM-dd'T'HH:mm:ss")
            .option("mode","failfast")
            .option("path","/home/prashant/spark-data/mental-health-in-tech-survey/survey.csv")
            .load()
```

There are many fields in this dataset. However, for my requirement, I am only concerned with two fields.

1. Gender
2. Treatment

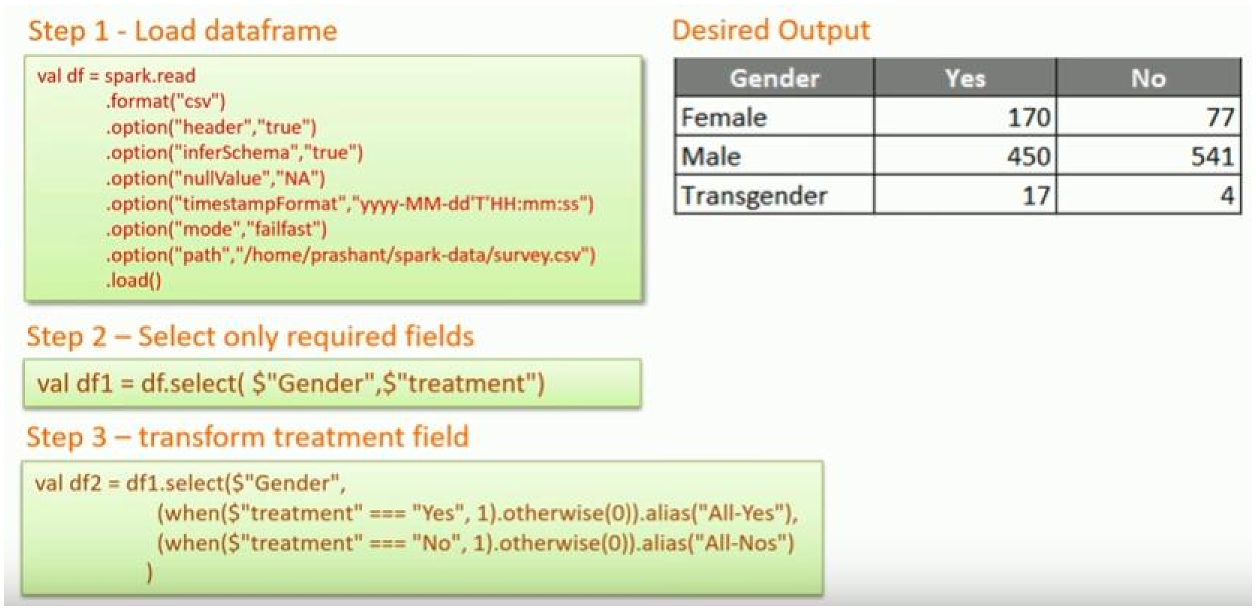
So, let me apply a select transformation to select only two fields.

```
val df1 = df.select( $"Gender",$"treatment")
          df1.show
```

So the *df1* is a new data frame that contains only two fields. But I need to count *Yes* and *Nos*. So I might need to separate them into two different columns. Once I have all the *Yes* in one column and all the *Nos* in another column, I can easily count them.
So, let's apply another transformation.

```
val df2 = df1.select($"Gender",
            (when($"treatment" === "Yes", 1).otherwise(0)).alias("All-Yes"),
            (when($"treatment" === "No", 1).otherwise(0)).alias("All-Nos")
                     )
```

This time, I am taking *df1* and applying another select transformation. I take the gender field as it was earlier. I am not applying any changes to that field. However, I am transforming the treatment field. When the treatment is *Yes*, I take it as a numeric one. Otherwise, I take it as a numeric zero. Now, I can simply take a sum of *All-Yes* column to get the total number of *Yes*. Right? You might want to do it differently.



Step 1 - Load dataframe

```
val df = spark.read
          .format("csv")
          .option("header","true")
          .option("inferSchema","true")
          .option("nullValue","NA")
          .option("timestampFormat","yyyy-MM-dd'T'HH:mm:ss")
          .option("mode","failfast")
          .option("path","/home/prashant/spark-data/survey.csv")
          .load()
```

Step 2 – Select only required fields

```
val df1 = df.select( $"Gender",$"treatment")
```

Step 3 – transform treatment field

```
val df2 = df1.select($"Gender",
                (when($"treatment" === "Yes", 1).otherwise(0)).alias("All-Yes"),
                (when($"treatment" === "No", 1).otherwise(0)).alias("All-Nos")
              )
```

Desired Output

Gender	Yes	No
Female	170	77
Male	450	541
Transgender	17	4

For example, When the treatment is *Yes*, take it as *Yes* otherwise take it as *Null*. Then you can apply a count function on that column.
However, I prefer to take it as a numeric value and apply the sum function instead of a count function. That helps me to avoid unnecessary dealing with nulls. You might also wonder that why do we apply two transformations for this. I mean, We could have done it in a single step.
Let me show you.

```
val df2 = df.select($"Gender",
                    (when($"treatment" === "Yes", 1).otherwise(0)).alias("All-Yes"),
                    (when($"treatment" === "No", 1).otherwise(0)).alias("All-Nos")
                     )
```

Instead of applying this transformation over df2, we could have applied it directly to df.
I mean, we could have avoided the first transformation and done it in a single step. You might argue that if we can avoid one transformation, and do it as a single step, Spark might perform better. Right? But that's not the truth in this case. There is no difference in performance. At least not in this example.
Why?

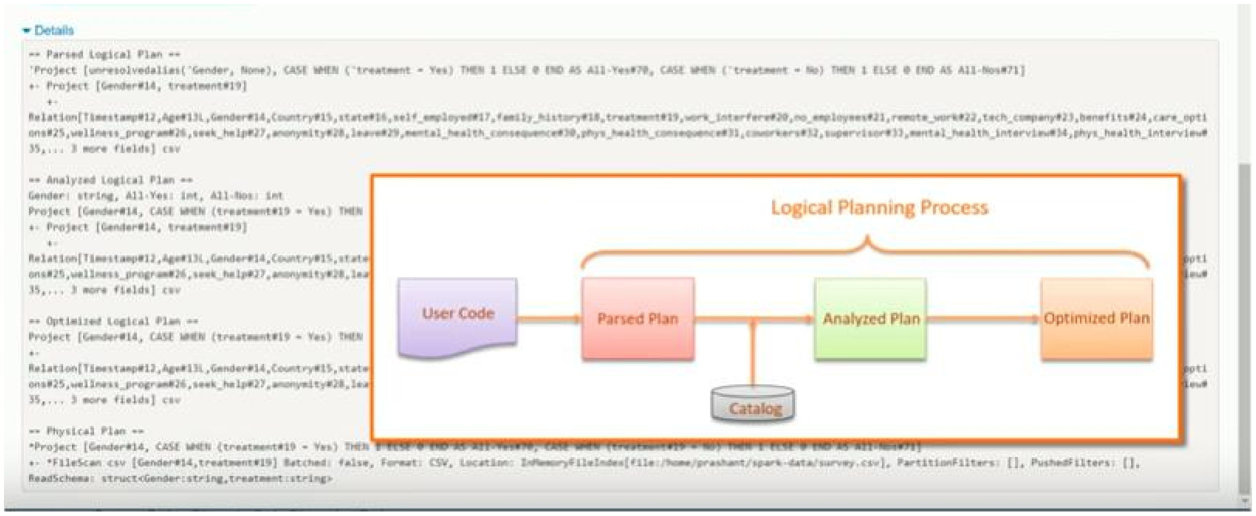
Let's try to understand that.
We already learned in RDDs that the transformations are lazy. They don't execute until we fire an action. Right? Spark implements data frames to be lazy because that design gives them at least two clear benefits.

1. Spark engine can act as a compiler.
2. They get an opportunity to apply necessary optimizations.

In our example, we applied two transformations. We haven't executed any action yet. So, Spark has executed nothing yet on the cluster. You might see a load operation on Spark UI. But I am talking about those two select transformations. None of them are executed yet.
Let's apply an action.

```
df2.collect
```

Now you will see one job in Spark UI.
If you jump to the SQL tab in the Spark UI and click on the collect job, you will get a lot of details.



```
== Parsed Logical Plan ==
'Project [unresolvedalias('Gender, None), CASE WHEN ('treatment = Yes) THEN 1 ELSE 0 END AS All-Yes#70, CASE WHEN ('treatment = No) THEN 1 ELSE 0 END AS All-Nos#71]
+- Project [Gender#14, treatment#19]
   +-
Relation[Timestamp#12,Age#13L,Gender#14,Country#15,state#16,self_employed#17,family_history#18,treatment#19,work_interfere#20,no_employees#21,remote_work#22,tech_company#23,benefits#24,care_opti
ons#25,wellness_program#26,seek_help#27,anonymity#28,leave#29,mental_health_consequence#30,phys_health_consequence#31,coworkers#32,supervisor#33,mental_health_interview#34,phys_health_interview#
35,... 3 more fields] csv

== Analyzed Logical Plan ==
Gender: string, All-Yes: int, All-Nos: int
Project [Gender#14, CASE WHEN (treatment#19 = Yes) THEN
+- Project [Gender#14, treatment#19]
   +-
Relation[Timestamp#12,Age#13L,Gender#14,Country#15,state                                                              opti
ons#25,wellness_program#26,seek_help#27,anonymity#28,lea                                                              iew#
35,... 3 more fields] csv

== Optimized Logical Plan ==
Project [Gender#14, CASE WHEN (treatment#19 = Yes) THEN
+-
Relation[Timestamp#12,Age#13L,Gender#14,Country#15,state                                                              opti
ons#25,wellness_program#26,seek_help#27,anonymity#28,lea                                                              iew#
35,... 3 more fields] csv

== Physical Plan ==
*Project [Gender#14, CASE WHEN (treatment#19 = Yes) THEN 1 ELSE 0 END AS All-Yes#70, CASE WHEN (treatment#19 = No) THEN 1 ELSE 0 END AS All-Nos#71]
+- *FileScan csv [Gender#14,treatment#19] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/prashant/spark-data/survey.csv], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<Gender:string,treatment:string>
```

You will see four plans.

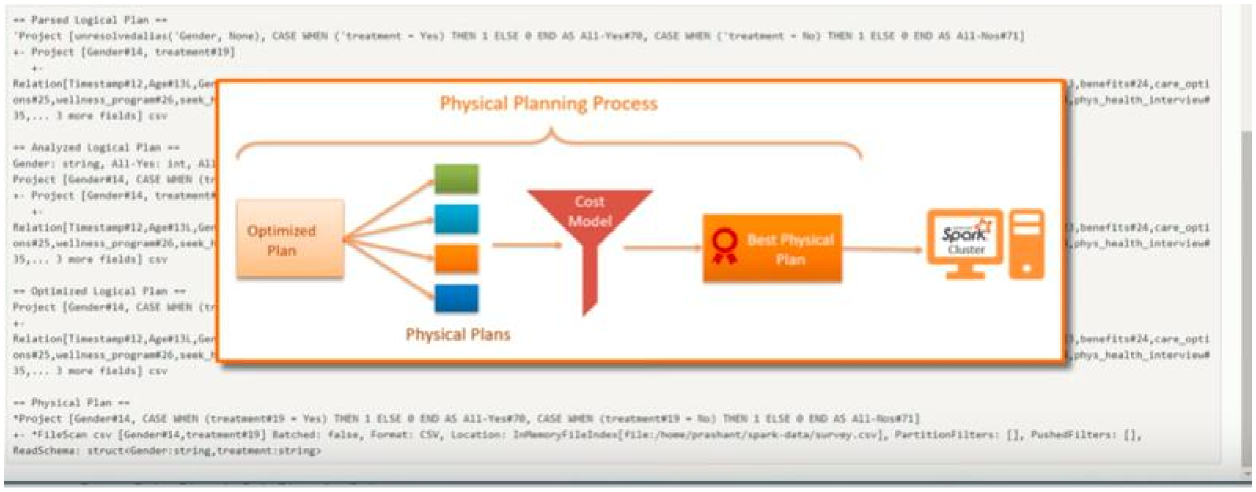
1. Parsed Logical Plan
2. Analyzed Logical Plan
3. Optimized Logical Plan

4. Finally the Physical Plan

When we execute an action, Spark takes the user code. In our case, It takes those two select transformations. It will then parse the user code and generate a parsed logical plan.
The second step is to analyze the initial plan and resolve the column names and their data types. The output of the second step is an analyzed logical plan. Apache Spark maintains a catalog of all the tables and data frame information. The analyzer makes a call to the catalog and resolves the initial plan. The analyzed plan clearly shows the column names and the datatypes.
The analyzed logical plan goes to an optimizer. As of now, the optimizer mainly performs two types of optimizations.

1. Pipelining
2. Predicate pushdown

Pipelining is as simple as combining multiple transformations together. We created two transformations. Both were the select operations. Spark realizes that it can combine them together into a single transformation. So, it simply does that. You can cross check it by looking at the optimized plan. The pipelining optimization doesn't only apply to a select transformation. Spark will look for all such opportunities and apply the pipelining where ever it is applicable.
The other type of optimization is the predicate pushdown. That simply means pushing down the filter conditions to the early stage instead of applying it at the end.

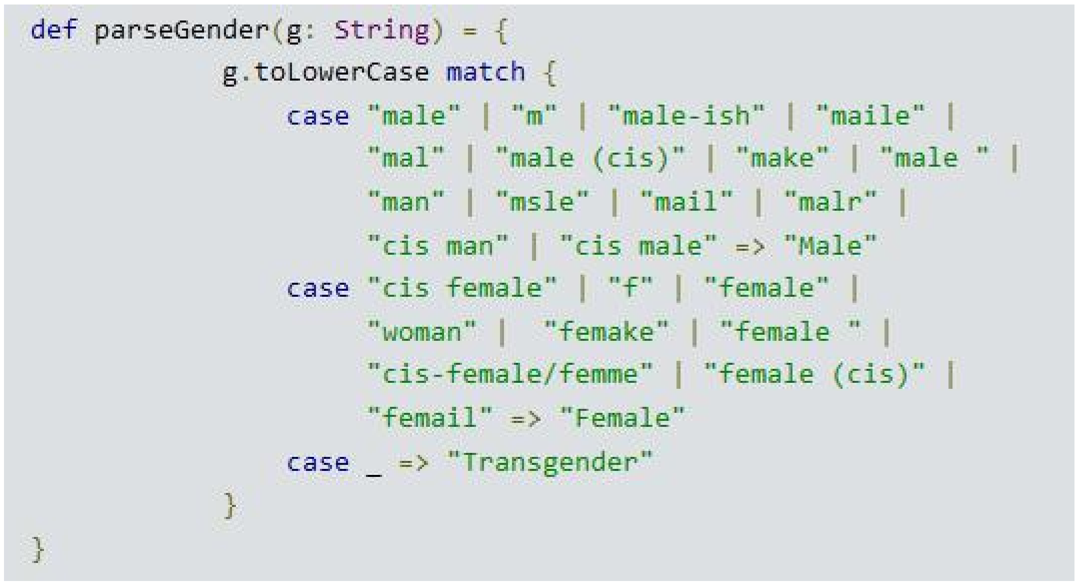


```
== Parsed Logical Plan ==
'Project [unresolvedalias('Gender, None), CASE WHEN ('treatment = Yes) THEN 1 ELSE 0 END AS All-Yes#70, CASE WHEN ('treatment = No) THEN 1 ELSE 0 END AS All-Nos#71]
+- Project [Gender#14, treatment#19]
   +-
Relation[Timestamp#12,Age#13L,Ge                                                                          ],benefits#24,care_opti
ons#25,wellness_program#26,seek_                                                                          ,phys_health_interview#
35,... 3 more fields] csv

== Analyzed Logical Plan ==
Gender: string, All-Yes: int, All
Project [Gender#14, CASE WHEN (tr
+- Project [Gender#14, treatment#
   +-
Relation[Timestamp#12,Age#13L,Ge                                                                          ],benefits#24,care_opti
ons#25,wellness_program#26,seek_                                                                          ,phys_health_interview#
35,... 3 more fields] csv

== Optimized Logical Plan ==
Project [Gender#14, CASE WHEN (tr
+-
Relation[Timestamp#12,Age#13L,Ge                                                                          ],benefits#24,care_opti
ons#25,wellness_program#26,seek_                                                                          ,phys_health_interview#
35,... 3 more fields] csv

== Physical Plan ==
*Project [Gender#14, CASE WHEN (treatment#19 = Yes) THEN 1 ELSE 0 END AS All-Yes#70, CASE WHEN (treatment#19 = No) THEN 1 ELSE 0 END AS All-Nos#71]
+- *FileScan csv [Gender#14,treatment#19] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/prashant/spark-data/survey.csv], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<Gender:string,treatment:string>
```

The optimized logical plan goes to the Spark compiler that generates a bunch of physical execution plans. The physical execution plan is nothing but a series of RDD transformations. The Spark engine generates multiple physical plans based on various considerations. Those considerations might be a different approach to perform a join operation. It may be the physical attributes of the underlying data file. It may be something else. However, Spark settles down to a single physical plan that it evaluates to be the best among others. Finally, The best plan goes for the execution on the cluster.
Great. Let's come back to our example. We loaded data. We applied one transformation. Then we applied another transformation. But we haven't reached the desired output. I think if group by the gender and compute a sum over the second and third column, we should get the desired output.
Let's try that.

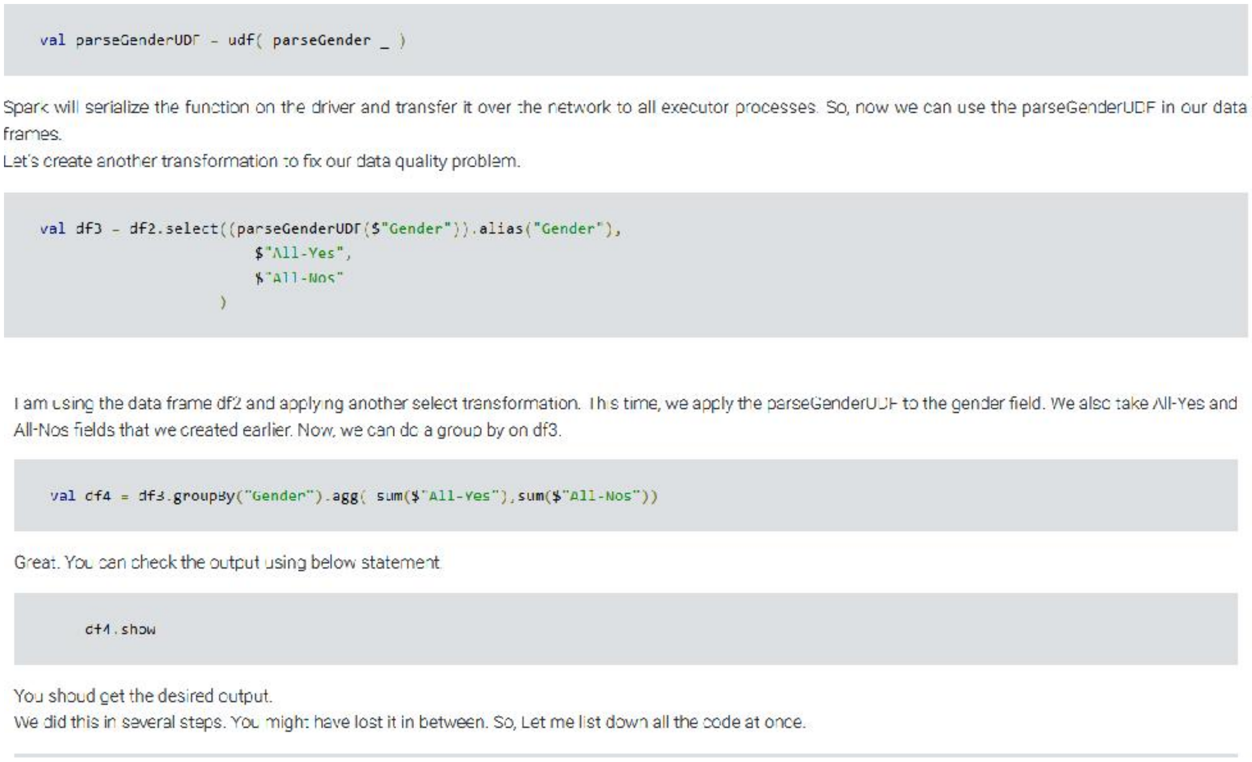
```
val df3 = df2.groupBy("Gender")
              .agg( sum($"All-Yes"),sum($"All-Nos"))
```

Now we apply a group by on gender. Then aggregate to calculate the sum of the other two columns. All we are doing here is nothing but simply chaining the data frame APIs. The syntax might look weird in the beginning, but you will be comfortable with this style in few days. All the methods that I am using in this example are available in dataset documentation with easy to follow examples.
If you check the output for the above transformation, you will realize that the Gender field is not very well coded. We have a data quality problem. There are many ways to deal with that issue. You might want to use a tricky regular expression and translate each value to one of the genders. However, we have a small dataset, and I think the safest method is to make a list of all unique values and handle them using a match case expression. I can quickly create a Scala function to handle it.
Here is the code. Well, this is a plain scala code.



```
def parseGender(g: String) = {
            g.toLowerCase match {
                case "male" | "m" | "male-ish" | "maile" |
                     "mal" | "male (cis)" | "make" | "male " |
                     "man" | "msle" | "mail" | "malr" |
                     "cis man" | "cis male" => "Male"
                case "cis female" | "f" | "female" |
                     "woman" |  "femake" | "female " |
                     "cis-female/femme" | "female (cis)" |
                     "femail" => "Female"
                case _ => "Transgender"
            }
}
```

We want this function to be available to all the executors. you can do that by registering this function as a UDF. Registering a UDF is as simple as passing the function to the UDF function.



```
val parseGenderUDF = udf( parseGender _ )
```

Spark will serialize the function on the driver and transfer it over the network to all executor processes. So, now we can use the parseGenderUDF in our data frames.
Let's create another transformation to fix our data quality problem.

```
val df3 = df2.select((parseGenderUDF($"Gender")).alias("Gender"),
                     $"All-Yes",
                     $"All-Nos"
                  )
```

I am using the data frame df2 and applying another select transformation. This time, we apply the parseGenderUDF to the gender field. We also take All-Yes and All-Nos fields that we created earlier. Now, we can do a group by on df3.

```
val df4 = df3.groupBy("Gender").agg( sum($"All-Yes"),sum($"All-Nos"))
```

Great. You can check the output using below statement

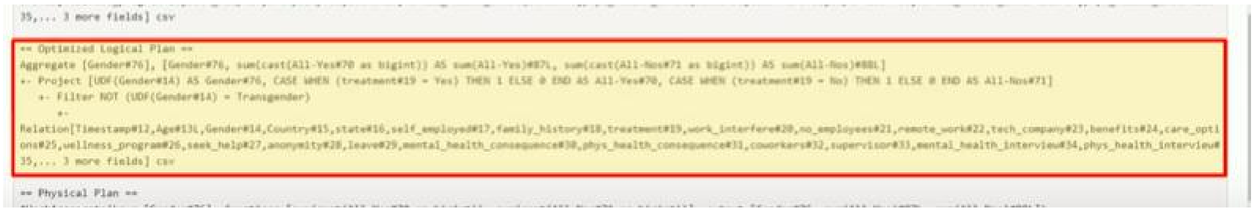
```
    df4.show
```

You shoud get the desired output.
We did this in several steps. You might have lost it in between. So, Let me list down all the code at once.

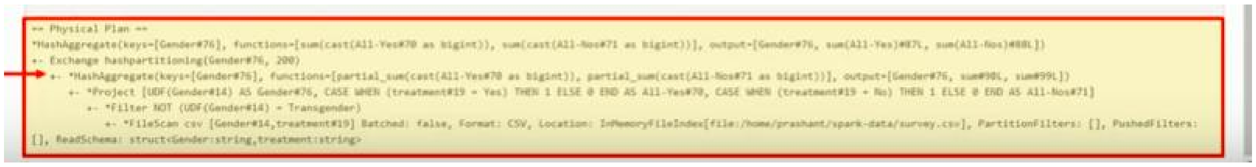
```
spark.conf.set("spark.sql.shuffle.partitions", 2)
val df = spark.read
             .format("csv")
             .option("header","true")
             .option("inferSchema","true")
             .option("nullValue","NA")
             .option("timestampFormat","yyyy-MM-dd'T'HH:mm?:ss")
             .option("mode","failfast")
             .option("path","/home/prashant/spark-data/mental-health-in-tech-survey/survey.csv")
             .load()
val df1 = df.select( $"Gender",$"treatment")
val df2 = df.select($"Gender",
                    (when($"treatment" === "Yes", 1).otherwise(0)).alias("All-Yes"),
                    (when($"treatment" === "No", 1).otherwise(0)).alias("All-Nos")
                  )
```

```
def parseGender(g: String) = {
  g.toLowerCase match {
    case "male" | "m" | "male-ish" | "maile" |
         "mal" | "male (cis)" | "make" | "male " |
         "man" | "msle" | "mail" | "malr" |
         "cis man" | "cis male" => "Male"
    case "cis female" | "f" | "female" |
         "woman" |  "femake" | "female " |
         "cis-female/femme" | "female (cis)" |
         "femail" => "Female"
    case _ => "Transgender"
   }
   }
val parseGenderUDF = udf(parseGender _)
val df3 = df2.select((parseGenderUDF($"Gender")).alias("Gender"),
                      $"All-Yes",
                      $"All-Nos"
                    )
val df4 = df3.groupBy("Gender").agg( sum($"All-Yes"),sum($"All-Nos"))
val df5 = df4.filter($"Gender" =!= "Transgender")
df5.collect
```

There are few more things covered in the video. So don't miss the video.



```
35,... 3 more fields] csv

== Optimized Logical Plan ==
Aggregate [Gender#76], [Gender#76, sum(cast(All-Yes#70 as bigint)) AS sum(All-Yes)#87L, sum(cast(All-Nos#71 as bigint)) AS sum(All-Nos)#88L]
+- Project [UDF(Gender#14) AS Gender#76, CASE WHEN (treatment#19 = Yes) THEN 1 ELSE 0 END AS All-Yes#70, CASE WHEN (treatment#19 = No) THEN 1 ELSE 0 END AS All-Nos#71]
   +- Filter NOT (UDF(Gender#14) = Transgender)
      +-
Relation[Timestamp#12,Age#13L,Gender#14,Country#15,state#16,self_employed#17,family_history#18,treatment#19,work_interfere#20,no_employees#21,remote_work#22,tech_company#23,benefits#24,care_options#25,wellness_program#26,seek_help#27,anonymity#28,leave#29,mental_health_consequence#30,phys_health_consequence#31,coworkers#32,supervisor#33,mental_health_interview#34,phys_health_interview#35,... 3 more fields] csv

== Physical Plan ==
```



```
== Physical Plan ==
*HashAggregate(keys=[Gender#76], functions=[sum(cast(All-Yes#70 as bigint)), sum(cast(All-Nos#71 as bigint))], output=[Gender#76, sum(All-Yes)#87L, sum(All-Nos)#88L])
+- Exchange hashpartitioning(Gender#76, 200)
   +- *HashAggregate(keys=[Gender#76], functions=[partial_sum(cast(All-Yes#70 as bigint)), partial_sum(cast(All-Nos#71 as bigint))], output=[Gender#76, sum#98L, sum#99L])
      +- *Project [UDF(Gender#14) AS Gender#76, CASE WHEN (treatment#19 = Yes) THEN 1 ELSE 0 END AS All-Yes#70, CASE WHEN (treatment#19 = No) THEN 1 ELSE 0 END AS All-Nos#71]
         +- *Filter NOT (UDF(Gender#14) = Transgender)
            +- *FileScan csv [Gender#14,treatment#19] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/prashant/spark-data/survey.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Gender:string,treatment:string>
```

Apache Spark Foundation Course - SQL over Dataframes

In the earlier videos, we started our discussion on Spark Data frames. By now, you must have realized that all that you need to learn is to model your business requirements using Spark Transformations. Once you learn to write the transformation that meets your business requirement, you have almost completed Apache Spark foundation course.
You can continue learning Spark internals, tuning, optimizations, and other things like streaming and machine learning. However, modelling your business requirement into a series of transformations is the most critical part of Spark development.
It is like learning SQL. Once you know the SQL, you can claim to be a database developer. Similarly, once you master the transformations, you can claim to be a Spark Developer. Moreover, if you know SQL, you are already a good Spark developer. That is the topic of this video.
In this video, we will augment our Data Frame knowledge with our SQL skills. So, let’s start.

Amazing SQL

You have already seen some transformation code earlier. We used the read API to load the data from a CSV file.

```
val df = spark.read
.format("csv")
.option("header","true")
.option("inferSchema","true")
.option("nullValue","NA")
.option("timestampFormat","yyyy-MM-dd'T'HH:mm:ss")
.option("mode","failfast")
.load("/home/prashant/spark-data/survey.csv")

//Then we applied a select transformation and a filter condition.
val sel = df.select("Timestamp", "Age","remote_work","leave").filter("Age > 30")
```

If you are a database developer, you will see the above transformation as an SQL expression.

```
select timestamp, age,remote_work,leave
from survey_tbl
where age > 30;
```

To make this SQL work, all you need is a table and an SQL execution engine. The good news is that the Spark offers you both of these things.
How?
We will see that in a minute. Before that, let's look at the other things that we did in the earlier video. We wanted to create a bar chart, and for that purpose, we did several things.

1. Created a user-defined function.
2. Applied the UDF in a select transformation.
3. Applied to filter out the transgender.
4. Applied a group by gender.
5. Calculated the sum of Yes and No.

Assuming you are good at writing SQL, If I allow you to do all of that using a SQL statement, you would be able to do it as a single SQL expression.

```
select gender, sum(all_yes), sum(all_nos)
from (select case when lower(trim(gender)) in ('male','m','male-ish','maile','mal',
                                               'male (cis)','make','male','man','msle',
                                               'mail', 'malr','cis man', 'cis male')
                  then 'Male'
                  when lower(trim(gender)) in ('cis female','f','female','woman',
                                             'femake','female ','cis-female/femme',
                                             'female (cis)','femail')
                  then 'Female'
                  else 'Transgender'
             end as gender,
             case when treatment == 'Yes' then 1 else 0 end as all_yes,
             case when treatment == 'No'  then 1 else 0 end as all_nos
      from survey_tbl)
      where gender != 'Transgender'
      group by gender
```

I hope you already get the sense of the point that I want to make. SQL is an excellent tool for many transformation requirements. Most of us are already skilled and comfortable with SQL. And for that reason, Apache Spark allows us to use SQL over a data frame.

Spark SQL requires Schema

Before we execute the above SQL in Spark, let's talk a little about the schema. A schema is nothing more than a definition for the column names and their data types. In our earlier example, we allowed the API to infer the schema. However, there are two approaches to handle schema.

1. Let the data source define the schema, and we infer it from the source.
2. Define a schema explicitly in your program and read the data using your schema definition.

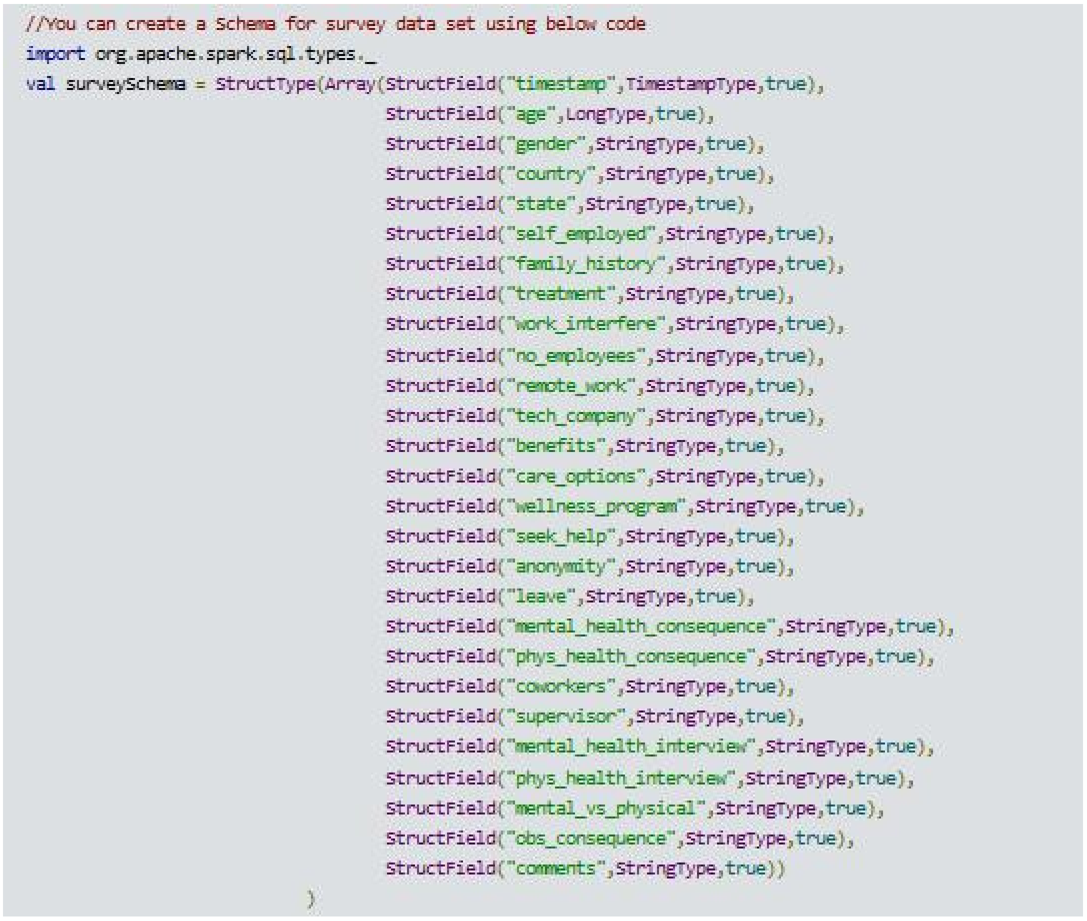
When your source system offers a well-defined schema, schema inference is a reasonable choice. However, it is a good idea to define your schema manually while working with untyped sources such as CSV and JSON.
In our current example, we are loading data from a CSV file. So, the recommendation is to define the schema instead of using the *inferSchema*.

Spark Types to define Schema

In my earlier video, I said that the Spark is a programming language in itself. The Spark type system is the main reason behind that statement. Apache Spark maintains its own type information, and they designed data frames to use Spark types.
What does that mean?
That means, Data frames do not use Scala types or Python types. No matter which language are you using for your code, A Spark data frame API always uses Spark types. And I believe, that was a design decision to bring SQL over the data frames across the languages.
You can get the list of Spark Types in org.apache.spark.sql.types package.

How to define a Spark Schema

We are all set with the theoretical fundamentals. Let's do something practical.
The below code defines a schema for the survey data set.



```
//You can create a Schema for survey data set using below code
import org.apache.spark.sql.types._
val surveySchema = StructType(Array(StructField("timestamp",TimestampType,true),
                                    StructField("age",LongType,true),
                                    StructField("gender",StringType,true),
                                    StructField("country",StringType,true),
                                    StructField("state",StringType,true),
                                    StructField("self_employed",StringType,true),
                                    StructField("family_history",StringType,true),
                                    StructField("treatment",StringType,true),
                                    StructField("work_interfere",StringType,true),
                                    StructField("no_employees",StringType,true),
                                    StructField("remote_work",StringType,true),
                                    StructField("tech_company",StringType,true),
                                    StructField("benefits",StringType,true),
                                    StructField("care_options",StringType,true),
                                    StructField("wellness_program",StringType,true),
                                    StructField("seek_help",StringType,true),
                                    StructField("anonymity",StringType,true),
                                    StructField("leave",StringType,true),
                                    StructField("mental_health_consequence",StringType,true),
                                    StructField("phys_health_consequence",StringType,true),
                                    StructField("coworkers",StringType,true),
                                    StructField("supervisor",StringType,true),
                                    StructField("mental_health_interview",StringType,true),
                                    StructField("phys_health_interview",StringType,true),
                                    StructField("mental_vs_physical",StringType,true),
                                    StructField("obs_consequence",StringType,true),
                                    StructField("comments",StringType,true))
                            )
```

```
//You can load the data using above schema
val df = spark.read
              .format("csv")
              .schema(surveySchema)
              .option("header","true")
              .option("nullValue","NA")
              .option("timestampFormat","yyyy-MM-dd'T'HH:mm:ss")
              .option("mode","failfast")
              .load("/home/prashant/spark-data/survey.csv")
```

Excellent. So, Spark data frame schema is a StructType that contains a set of StructFields. Each StructField defines a column. The StructField is a serializable class under Scala AnyRef.
The S StructField constructor can take four values.

1. The name of the column
2. The data type of the column.
3. A boolean that tells if the field is nullable. This parameter defaults to true.
4. You can also supply some metadata for each column. The metadata is nothing but a map of key-value pairs. The default value is empty.

The StructType is also a class that holds an array of StructFields. If you are using Python, both of those structs are same in Python as well. However, the Python StructType is a list of StructFields whereas Scala StrcutType is an array of StructField.
You can use the above code to create a schema and load your data using an explicit schema. Once loaded, you should have a data frame.

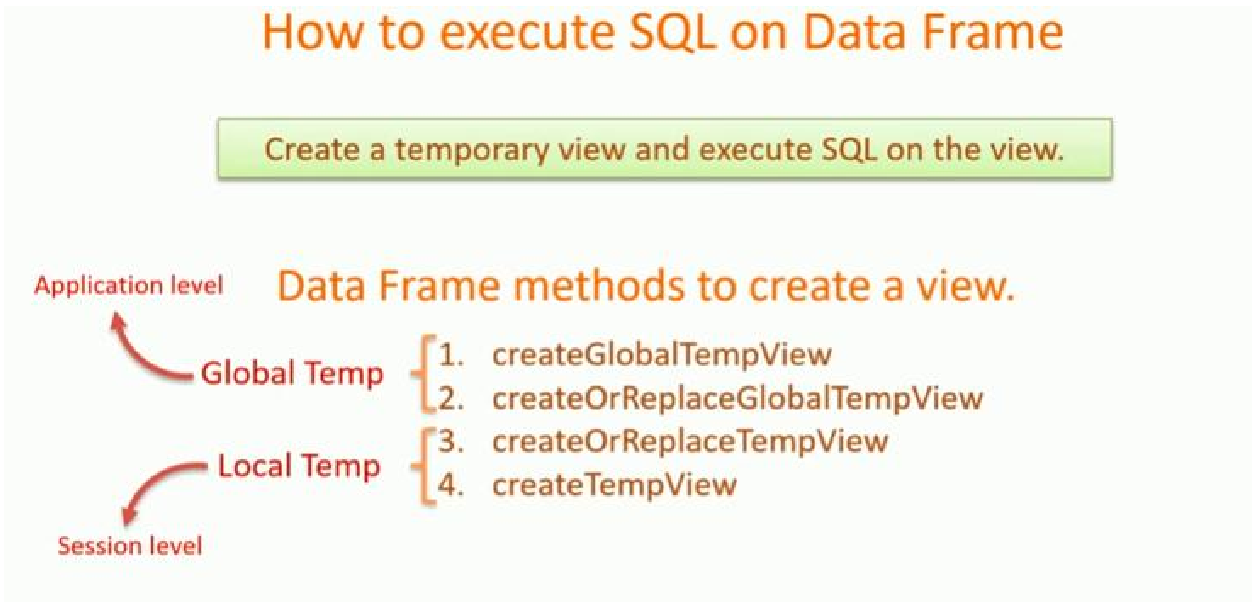
Spark Temporary View

Apache Spark allows you to create a temporary view using a data frame. It is just like a view in a database. Once you have a view, you can execute SQL on that view. Spark offers four data frame methods to create a view.

1. createGlobalTempView
2. createOrReplaceGlobalTempView
3. createOrReplaceTempView
4. createTempView

As you can guess by just looking at the method names, there are two types of temporary views. Temporary view and a Global Temporary view. We can also refer it as a Local Temporary view and a Global Temporary view. Let's try to understand the difference.

Global vs Local Temp view



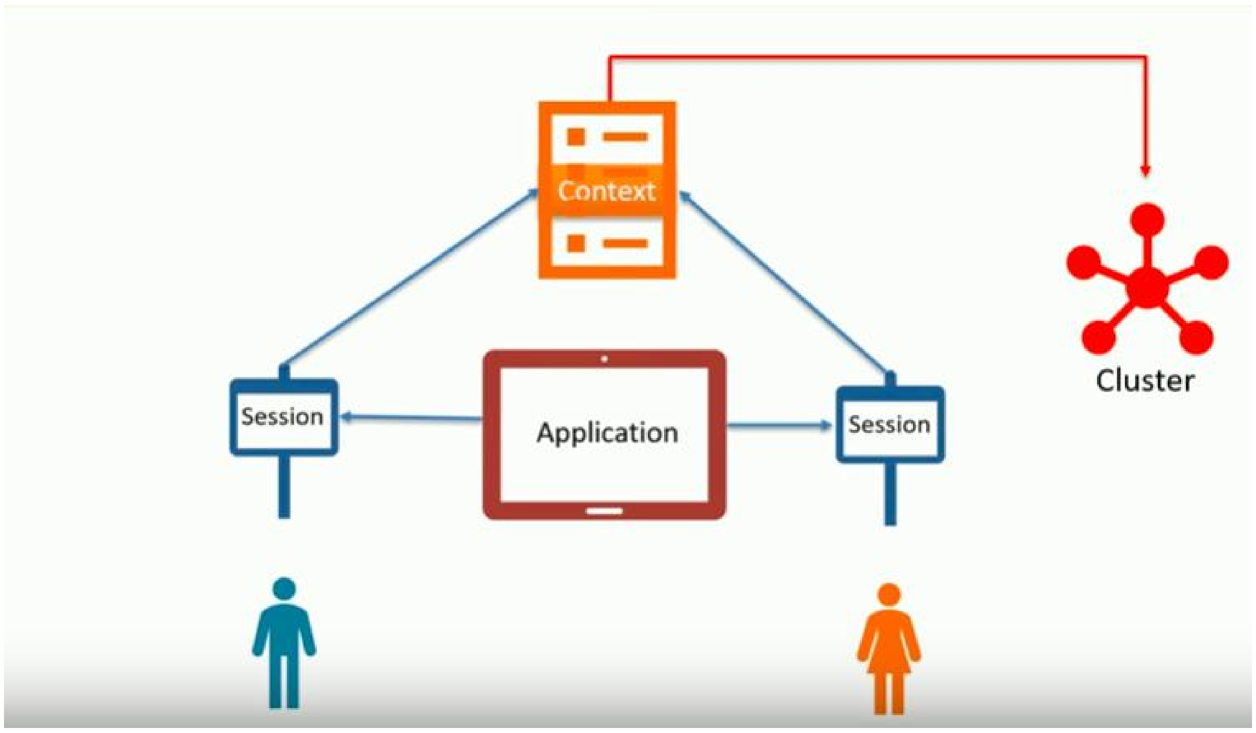
How to execute SQL on Data Frame

Create a temporary view and execute SQL on the view.

Data Frame methods to create a view.

Application level ← Global Temp
1. createGlobalTempView
2. createOrReplaceGlobalTempView

Session level ← Local Temp
3. createOrReplaceTempView
4. createTempView



The local temporary view is only visible to the current spark session. However, a Global temporary view is visible to the current spark application across the sessions.
Wait a minute. Do you mean a SparkSession and a Spark Application are two different things?
Yes. We normally start a Spark Application by creating a Spark session. To a beginner, it appears that a Spark Application can have a single session. However, that is not true. You can have multiple sessions in a single Spark application. The Spark session internally creates a Spark context. A SparkContext represents the connection to a Spark cluster. It also keeps track of all the RDDs, cached data as well as the configurations.
You cannot have more than one Spark Context in a single JVM. That means, one instance of an application can have only one connection to the cluster and hence a single Spark context. You cannot have more than one Spark context. However, your application can create multiple Spark Sessions. All those sessions will point to the same context, but you can have multiple sessions.
In your standard applications, you may not need to create multiple spark sessions. However, if you are developing an application that needs to support multiple interactive users, you might want to create one Spark Session for each user session. Ideally, we should be able to create multiple connections to Spark cluster for each user in the above use case, but creating multiple contexts is not yet supported by Spark. The documentation claims that they will remove this restriction in the future releases.
So, coming back to local temporary views, they are only visible to the current session. However, global temporary views are visible across the spark sessions within the same application.
In all this discussion, one thing is crystal clear. None of them are visible to other applications. So, you create a global temporary view or a local temporary view, they are always local to your application, and they live only till your application is alive.
Great, since I am not going to create multiple sessions, let me create a local temporary view.

```
df.createOrReplaceTempView("survey_tbl")
```

The method takes the name of the view as an argument. The above statement must have created a temporary table or a view. Where can you find it?

Well, a temporary view is maintained by the Spark session. So, let's check the Spark session.

```
spark.catalog.listTables.show
```

Spark session offers you a catalog. A catalog is an interface that allows you to create, drop, alter or query underlying databases, tables, and functions. I recommend you to at least go through the documentation for the catalog interface . The above statement is using the listTables method in the catalog. If you check the output of the above statement, you can see that the view that we created is a temporary table that doesn't belong to any databases.
Let's create a global temporary table and see if we can list that as well.

```
df.createOrReplaceGlobalTempView("survey_gtbl")
```

We used the appropriate method to create a Global temporary view on our data frame. I named the view as survey_gtbl. If you call the catalog listTables method once again. You won’t see that global table. There is a reason for that. A Global temp table belongs to a system database called global_temp. So, if you want to access the global temp table, you must look into the global_temp database. So, the correct method call should also specify the database name.

```
spark.catalog.listTables("global_temp").show
```

The output of the above statement should list your global temp table. Once you register the temp table, executing your SQL statement is a simple thing.

```
spark.sql("""select timestamp, age,remote_work,leave
from survey_tbl
where age > 30""")
```

Execute an SQL on the spark session, and you get a data frame in return. So, if you think that the SQL is simpler to solve your problems, instead of using lengthy data frame API chains, you are free to use SQL. And surprisingly, you do not have a performance penalty. The SQL works as fast as a Data frame transformation.

```
spark.sql("""select gender, sum(yes), sum(no)
        from (select case when lower(trim(gender)) in ('male','m','male-ish','maile','mal',
                                                        'male (cis)','make','male ','man','msle',
                                                        'mail','malr','cis man','cis male')
                          then 'Male'
                          when lower(trim(gender)) in ('cis female','f','female','woman',
                                                        'female','female ','cis-female/femme',
                                                        'female (cis)','femail')
                          then 'Female'
                          else 'Transgender'
                          end as gender,
                          case when treatment == 'Yes' then 1 else 0 end as yes,
                          case when treatment == 'No' then 1 else 0 end as no
              from survey_tbl)
     where gender != 'Transgender'
     group by gender""").show
```

So, instead of using a UDF and then a confusing chain of APIs, we can use an SQL statement to achieve whatever we did in the previous video. You can use API chains or an SQL, and both delivers the same performance. The choice is yours.