

BRIEF INTRODUCTION TO SPARK SQL

What is Spark SQL

“Spark SQL” is a “Module” used for “Structured Data Processing” with multiple interfaces. There are two ways to use “Spark SQL” “Module” -

- One way to use “Spark SQL” is by simply executing “SQL Queries”.
- Another way to interact with “Spark SQL” is by using the “DataFrame” API, which is available in “Python”, “Scala”, “Java” and “R”.

SQL

DataFrame API

Python, Scala, Java, R

The same “Spark SQL” “Query” can be expressed with “SQL” and “DataFrame” API.

```
%sql

select      c_customer_id Customer_ID,
            c_salutation Title,
            c_first_name First_Name,
            c_last_name Last_Name
from        retailer_db.customerTbl
where       trim(c_salutation) = 'Mr.'
order by    c_first_name
```

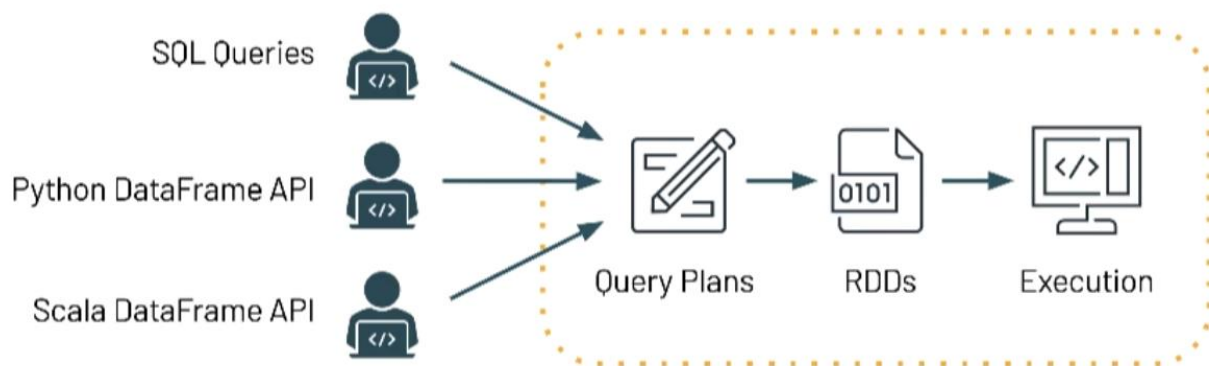
```
from pyspark.sql.functions import col, trim

customTblQueryDf = spark.table("retailer_db.customerTbl")\
    .select(\
        col("c_customer_id").alias("Customer_ID"),\
        col("c_salutation").alias("Title"),\
        col("c_first_name").alias("First_Name"),\
        col("c_last_name").alias("Last_Name")\
    )\
    .where(trim(col("c_salutation")) == "Mr.")\
    .orderBy("c_first_name")

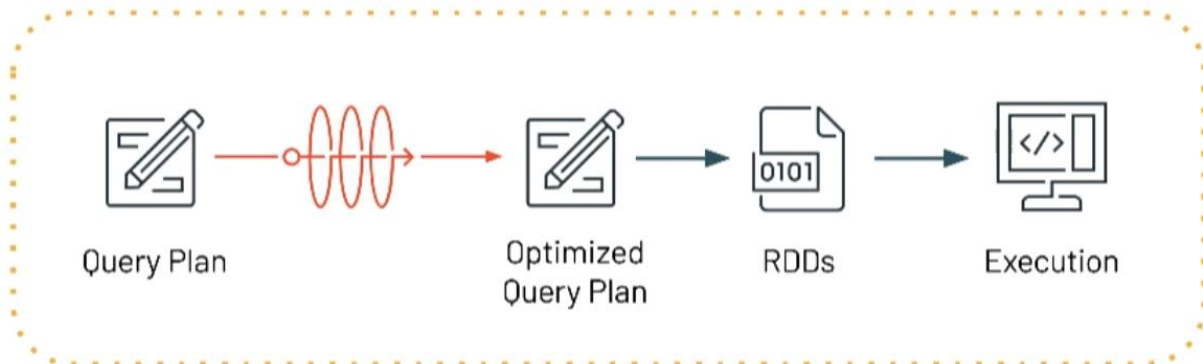
display(customTblQueryDf)
```

- “Spark SQL” executes all “Queries” on the same “Spark SQL Engine”. A “Query” can be expressed with any interface, and, the “Spark SQL Engine” will generate the same “Query Plan” to execute on the “Spark Cluster”, no matter which Programming Language is used.

With “Spark SQL”, it is possible to easily mix “SQL Queries” inside “Spark Programs”, and, use a common API across Languages.



- Apache Spark is able to perform “Optimizations” with the information “Spark SQL” provides on “Data Structure” and “Computations” being performed. Apache Spark “Optimizes” the “Query” before the commands, present in the “Query”, are executed.



BRIEF INTRODUCTION TO DATAFRAME

What is DataFrame

🌈 A “DataFrame” is a “Distributed Collection of Data” grouped into “Named Columns”.

	Customer_ID ▲	Title ▲	First_Name ▲	Last_Name ▲
1	AAAAAAAAANOBBA	Mr.	null	Carter
2	AAAAAAAAOPHCB	Mr.	null	Meyer
3	AAAAAAAAAMMOJA	Mr.	null	Henry
4	AAAAAAAAPEGC	Mr.	null	Salter
5	AAAAAAAAJMOJA	Mr.	null	Tolbert
6	AAAAAAAAJIJC	Mr.	null	Chen

What is Schema

🌈 A “Schema” defines the “Column Names” and “Data Types” of a “DataFrame”.

```
item_id: string
name: string
price: double
qty: int
```

DataFrame Transformations

🌈 “DataFrame Transformations” are “Methods” that return a new “DataFrame” and are lazily evaluated. This means that the “Transformations” are not actually run immediately.

- ✚ It is seen in the below image that “select”, “where”, and, “orderBy” are chained together to build new “DataFrame”. “select”, “where”, and, “orderBy” are examples of “Transformations”, because, each of these “Methods” return a new “DataFrame”. Multiple “Transformations” can be chained together to build new “DataFrames”.

```
from pyspark.sql.functions import col, trim

customTblQueryDf = spark.table("retailer_db.customerTbl")\
    .select(\
        col("c_customer_id").alias("Customer_ID"),\
        col("c_salutation").alias("Title"),\
        col("c_first_name").alias("First_Name"),\
        col("c_last_name").alias("Last_Name")\
    )\
    .where(trim(col("c_salutation")) == "Mr.")\
    .orderBy("c_first_name")

display(customTblQueryDf)
```

DataFrame Actions

- ✚ “DataFrame Actions” are “Methods” that trigger “Computation”. If there are some “Transformations” in a code, an “Action” is needed to “Trigger” the “Execution” of any “DataFrame Transformations”. The “Action” operations require Apache Spark to execute “Computations” to return the exact expected result.
- ✚ In the below image, there are three “Transformations”, namely “select”, “where” and “orderBy”. Those are not “Triggered” to execute until the “Action”, i.e., “show” is called.

```
spark.table("retailer_db.customerTbl")\
    .select(\
        col("c_customer_id").alias("Customer_ID"),\
        col("c_salutation").alias("Title"),\
        col("c_first_name").alias("First_Name"),\
        col("c_last_name").alias("Last_Name")\
    )\
    .where(trim(col("c_salutation")) == "Mr.")\
    .orderBy("c_first_name")\
    .show()
```

- ✚ Example of some “Actions” are -
- count() - “count ()” returns the number of “Records” in a “DataFrame”.
 - collect() - “collect ()” returns an “Array of All the Rows” in a “DataFrame”.
 - show() - “show ()” displays the top few “Rows” of a “DataFrame”.

BRIEF INTRODUCTION TO SPARKSESSION

What is SparkSession

- ✚ The first step of any Spark Application is creating a “SparkSession”, which enables to run Spark codes.
- ✚ The “SparkSession” class provides the “Single Entry Point” to all functionalities in Spark using the “DataFrame” API.
- ✚ “SparkSession” is automatically created for the Users in “Databricks Notebook” as the variable “spark”.
- ✚ Some of the useful “SparkSession” methods are -
 - sql - Returns a “DataFrame” representing the “Result” of the given “Query”.
 - table - Returns the specified “Table” as a “DataFrame”.
 - read - Returns a “DataFrameReader” that can be used to read “Data” as a “DataFrame”.
 - range - Creates a “DataFrame” with “Column” containing elements in a “Range” from “Start” to “End” (exclusive) with “Step” value, and, “Number of Partitions”.
 - createDataFrame - Creates a “DataFrame” from a “List of Tuples”, primarily used for “Testing”.

READ AND WRITE USING DATAFRAME

Different Types of Data Source

✚ **Comma Separated Values (CSV)** - A “CSV File” is a “Text File” that uses comma “,” or, other delimiters to separate “Values”.

In the below image, it is seen that the “Optional Header” is the “First Line”, and, the “Data Records” in each “Line” below that.

```
item_id, name, price, qty
M_PREM_Q, Premium Queen Mattress, 1795, 35
M_STAN_F, Standard Full Mattress, 945, 24
M_PREM_F, Premium Full Mattress, 1695, 45
M_PREM_T, Premium Twin Mattress, 1095, 18
```

✚ **Parquet** - Apache Parquet is a “Columnar Storage Format” that provides “Compressed Efficient Columnar Data Representation”. Unlike “CSV”, the “Parquet” allows to load only the needed “Columns”. So, the “Values” for a “Single Record” are not stored together.

Name	Score	ID

Row-Oriented data on disk

Kit	4.2	1	Alex	4.5	2	Terry	4.1	3
-----	-----	---	------	-----	---	-------	-----	---

Column-Oriented data on disk

Kit	Alex	Terry	4.2	4.5	4.1	1	2	3
-----	------	-------	-----	-----	-----	---	---	---

This file format is available to any Project in “Hadoop Ecosystem” regardless of the choice of “Data Processing Framework”, “Data Model”, or, “Programming Language”. The “Schema” is stored in the “Footer” of the “Parquet File”. So, there is no need to “Infer the Schema”.

The “Parquet Files” does not waste space storing “Missing Values”.

“Parquet File” has “Predicate Push Down”, where it pushes the “Filters” down to the “Source”.

There is also “Data Skipping” feature available, where “Parquet File” stores only the “Minimum” and “Maximum” values of each “Segment”, so that the entire file can be skipped.

It is harder to corrupt a “Parquet File”, because, people can’t just easily open and modify a “Parquet File”.

“Schema” needs to be defined upfront for a Parquet File” for working with “Streaming Data”.

✚ **Delta Lake** - “Delta Lake” is an “Open-Source Technology” designed to work with Apache Spark to bring “Reliability” to “Data Lake”. “Delta Lake” runs on top of an existing “Data Lake” to provide -

- ACID Transactions
- Scalable Metadata Handling
- Unified Streaming and Batch Processing



Read Files Using DataFrameReader

- ✚ “DataFrameReader” is accessible through the “read ()” method of “SparkSession”.
- ✚ “DataFrameReader” class includes methods to load “DataFrames” from different “External Storage Systems”.
- ✚ “DataFrameReader” class includes several methods for reading the “Data” from different file formats, as well as some additional methods for configurations.

```
spark.read.parquet("/mnt/training/ecommerce/events.parquet")
```

Write Files Using DataFrameWriter

- ✚ “DataFrameWriter” is accessible through the “write ()” method of “SparkSession”.
- ✚ “DataFrameReader” class includes several methods for writing out “Data” to different file formats, as well as some additional methods for configurations.

```
customerDf.write\  
    .format("csv")\  
    .mode("Overwrite")\  
    .options(  
        path = "dbfs:/tmp/output_csv",\  
        header = "true",\  
        sep = "|"\  
    )\  
    .save()
```


Reading a CSV File Using DataFrameReader

- ✚ A “CSV File” can be read using “DataFrameReader” class in two ways -
 - Using “csv ()” Method - The “DataFrameReader” class has a special method “csv” to read “Data” from a “CSV File”. Pass the “Path” of the “CSV File” to this method.

```
customerDfReadCsvWithHeader = spark.read\  
    .option("header", "true")\  
    .csv("dbfs:/FileStore/tables/retailer/data/customer.csv")
```

Output -

► (1) Spark Jobs

▼  customerDfReadCsvWithHeader: pyspark.sql.dataframe.DataFrame


```
c_customer_sk: string
c_customer_id: string
c_current_cdemo_sk: string
c_current_hdemo_sk: string
c_current_addr_sk: string
c_first_ship_to_date_sk: string
c_first_sales_date_sk: string
c_salutation: string
c_first_name: string
c_last_name: string
c_preferred_cust_flag: string
c_birth_day: string
c_birth_month: string
c_birth_year: string
c_birth_country: string
c_login: string
c_email_address: string
c_last_review_date: string
```

- Using “format ()” and “load ()” Method - The “DataFrameReader” class has a method called “format ()”, where the type of file to read needs to be passed. In this case “csv” needs to be passed. Then pass the “Path” of the “CSV File” to another method of the “DataFrameReader” class, i.e., “load ()”.

```
customerDatWithHeader = spark.read\
    .option("header", "true")\
    .option("sep", "|")\
    .format("csv")\
    .load("dbfs:/FileStore/tables/retailer/data/customer.dat")
```


Output -

▶ (1) Spark Jobs

▼  customerDatWithHeader: pyspark.sql.dataframe.DataFrame

```
c_customer_sk: string
c_customer_id: string
c_current_demo_sk: string
c_current_hdemo_sk: string
c_current_addr_sk: string
c_first_ship_to_date_sk: string
c_first_sales_date_sk: string
c_salutation: string
c_first_name: string
c_last_name: string
c_preferred_cust_flag: string
c_birth_day: string
c_birth_month: string
c_birth_year: string
c_birth_country: string
c_login: string
c_email_address: string
c_last_review_date: string
```

Handling Schema of DataFrame

 **inferSchema Option** - While reading a “File” using Apache Spark, the “inferSchema” option tells Spark to infer the “Schema” of the “File” to read. This means that Spark has to analyze the entire “File” to figure out the “Data Type” of each “Column”. Hence, a “Spark Job” would be “Triggered” for this. To do so, the Value of the “inferSchema” option needs to be passed as “true” inside “option” method.

```
customerDf = spark.read\
    .options(
        header = "true",\
        sep = "|",\
        inferSchema = "true"\
    )\
    .csv("dbfs:/FileStore/tables/retailer/data/customer.dat")
```

The “Schema” of a “DataFrame” can be viewed using the “printSchema ()” method, which is called on the “DataFrame”.

```
customerDf.printSchema()
```

Output -

```
▶ (1) Spark Jobs
▶ customerDf: pyspark.sql.dataframe.DataFrame = [c_customer_sk: integer, c_customer_id: string ... 16 more fields]
root
|-- c_customer_sk: integer (nullable = true)
|-- c_customer_id: string (nullable = true)
|-- c_current_demo_sk: integer (nullable = true)
|-- c_current_hdemo_sk: integer (nullable = true)
|-- c_current_addr_sk: integer (nullable = true)
|-- c_first_ship_to_date_sk: integer (nullable = true)
|-- c_first_sales_date_sk: integer (nullable = true)
|-- c_salutation: string (nullable = true)
|-- c_first_name: string (nullable = true)
|-- c_last_name: string (nullable = true)
|-- c_preferred_cust_flag: string (nullable = true)
|-- c_birth_day: integer (nullable = true)
|-- c_birth_month: integer (nullable = true)
|-- c_birth_year: integer (nullable = true)
|-- c_birth_country: string (nullable = true)
|-- c_login: string (nullable = true)
|-- c_email_address: string (nullable = true)
|-- c_last_review_date: double (nullable = true)
```

🚦 **User Defined Schema** - It is possible to manually define the “Schema” by creating a “StructType” with “Column Names” and “Data Types”. In this case, The “Data Types” needed to be imported from the “pyspark.sql.types” “Module”.

First, the “StructType” needs to be specified, then, inside the “StructType”, a “List” of “StructFields”. Inside each “StructField”, specify the “Column Name”, followed by the “Data Type” of the “Column”, and, finally whether “Nullable” is “True”. “Nullable” as “True” is not a “Data Constraint”.

```

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, LongType

customerSchema = StructType([
    StructField("c_customer_sk", IntegerType(), True),
    StructField("c_customer_id", StringType(), True),
    StructField("c_current_cdemo_sk", LongType(), True),
    StructField("c_current_hdemo_sk", IntegerType(), True),
    StructField("c_current_addr_sk", IntegerType(), True),
    StructField("c_first_shipto_date_sk", LongType(), True),
    StructField("c_first_sales_date_sk", LongType(), True),
    StructField("c_salutation", StringType(), True),
    StructField("c_first_name", StringType(), True),
    StructField("c_last_name", StringType(), True),
    StructField("c_preferred_cust_flag", StringType(), True),
    StructField("c_birth_day", IntegerType(), True),
    StructField("c_birth_month", IntegerType(), True),
    StructField("c_birth_year", IntegerType(), True),
    StructField("c_birth_country", StringType(), True),
    StructField("c_login", StringType(), True),
    StructField("c_email_address", StringType(), True),
    StructField("c_last_review_date", LongType(), True)
])

```

Now, read from the "CSV File" using the "User Defined Schema".

```

customerUserDefinedSchemaDf = spark.read.option("header", "true")\
    .schema(customerSchema)\
    .csv("dbfs:/FileStore/tables/retailer/data/customer.csv")

customerUserDefinedSchemaDf.printSchema()

```

Output -

```

▶ customerUserDefinedSchemaDf: pyspark.sql.dataframe.DataFrame = [c_customer_sk: integer, c_customer_id: string ... 16 more fields]
root
 |-- c_customer_sk: integer (nullable = true)
 |-- c_customer_id: string (nullable = true)
 |-- c_current_cdemo_sk: long (nullable = true)
 |-- c_current_hdemo_sk: integer (nullable = true)
 |-- c_current_addr_sk: integer (nullable = true)
 |-- c_first_shipto_date_sk: long (nullable = true)
 |-- c_first_sales_date_sk: long (nullable = true)
 |-- c_salutation: string (nullable = true)
 |-- c_first_name: string (nullable = true)
 |-- c_last_name: string (nullable = true)
 |-- c_preferred_cust_flag: string (nullable = true)
 |-- c_birth_day: integer (nullable = true)
 |-- c_birth_month: integer (nullable = true)
 |-- c_birth_year: integer (nullable = true)
 |-- c_birth_country: string (nullable = true)
 |-- c_login: string (nullable = true)
 |-- c_email_address: string (nullable = true)
 |-- c_last_review_date: long (nullable = true)

```

🌈 **User Defined Schema Using DDL Formatted String** - Alternatively, it is possible to define the “Schema” using a “DDL-Formatted String”. The “Schema” is created as “DDL Formatted String” and passed to the “schema ()” method.

```
customerDdlSchema = """
c_customer_sk int,
c_customer_id string,
c_current_cdemo_sk long,
c_current_hdemo_sk int,
c_current_addr_sk int,
c_first_shipto_date_sk long,
c_first_sales_date_sk long,
c_salutation string,
c_first_name string,
c_last_name string,
c_preferred_cust_flag string,
c_birth_day int,
c_birth_month int,
c_birth_year int,
c_birth_country string,
c_login string,
c_email_address string,
c_last_review_date long
"""
```

Now, read from the “CSV File” using the “DDL-String User Defined Schema”.

```
customerDdlSchemaDf = spark.read.option("header", "true")\
    .schema(customerDdlSchema)\
    .csv("dbfs:/FileStore/tables/retailer/data/customer.csv")

customerDdlSchemaDf.printSchema()
```

Output -

```

▶ customerDdlSchemaDf: pyspark.sql.dataframe.DataFrame = [c_customer_sk: integer, c_customer_id: string ... 16 more fields]
root
|-- c_customer_sk: integer (nullable = true)
|-- c_customer_id: string (nullable = true)
|-- c_current_cdemo_sk: long (nullable = true)
|-- c_current_hdemo_sk: integer (nullable = true)
|-- c_current_addr_sk: integer (nullable = true)
|-- c_first_shipto_date_sk: long (nullable = true)
|-- c_first_sales_date_sk: long (nullable = true)
|-- c_salutation: string (nullable = true)
|-- c_first_name: string (nullable = true)
|-- c_last_name: string (nullable = true)
|-- c_preferred_cust_flag: string (nullable = true)
|-- c_birth_day: integer (nullable = true)
|-- c_birth_month: integer (nullable = true)
|-- c_birth_year: integer (nullable = true)
|-- c_birth_country: string (nullable = true)
|-- c_login: string (nullable = true)
|-- c_email_address: string (nullable = true)
|-- c_last_review_date: long (nullable = true)

```

- One thing to notice is that, when the “CSV Files” are read using “User Defined Schemas”, no “Spark Jobs” were “Triggered”, because, Apache Spark did not analyze the “CSV Files” to get the “Schema”. “Schemas” were already passed in. In this case, when the “Data” is actually being read, like using “show ()” method, then only “Spark Jobs” would be “Triggered”.

Reading a JSON File Using DataFrameReader

- A “JSON File” can be read using “DataFrameReader” class in two ways -
 - **Using “json ()” Method** - The “DataFrameReader” class has a special method “json” to read “Data” from a “JSON File”. Pass the “Path” of the “JSON File” to this method.

```

jsonDf = spark.read\
    .option("inferSchema", "true")\
    .json("dbfs:/FileStore/tables/retailer/data/single_line.json")

```

Output -

► (1) Spark Jobs

```
▼ jsonDf: pyspark.sql.dataframe.DataFrame
  ▼ address: struct
    |   city: string
    |   country: string
    |   state: string
    birthday: string
    email: string
    first_name: string
    id: long
    last_name: string
  ▼ skills: array
    |   element: string
```

- Using “format ()” and “load ()” Method - The “DataFrameReader” class has a method called “format ()”, where the type of file to read needs to be passed. In this case “json” needs to be passed. Then pass the “Path” of the “JSON File” to another method of the “DataFrameReader” class, i.e., “load ()”.

```
jsonFormatLoadDf = spark.read\
    .option("inferSchema", "true")\
    .format("json")\
    .load("dbfs:/FileStore/tables/retailer/data/single_line.json")
```

Output -

► (1) Spark Jobs

```
▼ jsonFormatLoadDf: pyspark.sql.dataframe.DataFrame
  ▼ address: struct
    |   city: string
    |   country: string
    |   state: string
    birthday: string
    email: string
    first_name: string
    id: long
    last_name: string
  ▼ skills: array
    |   element: string
```

Handling Schema of DataFrame Created from a JSON File

- 🚦 **inferSchema Option** - Like the “CSV File”, while reading a “JSON File”, if the Value of “inferSchema” option is provided as “true” in the “option” method, the “inferSchema” option tells Spark to analyze the entire “JSON File” to figure out the “Data Type” of each “Column”. Hence, a “Spark Job” would be “Triggered” for this.

```
jsonDf.printSchema()
```

Output -

```
▶ (1) Spark Jobs
▶ jsonDf: pyspark.sql.dataframe.DataFrame = [address: struct, birthday: string ... 5 more fields]
root
|-- address: struct (nullable = true)
|   |-- city: string (nullable = true)
|   |-- country: string (nullable = true)
|   |-- state: string (nullable = true)
|-- birthday: string (nullable = true)
|-- email: string (nullable = true)
|-- first_name: string (nullable = true)
|-- id: long (nullable = true)
|-- last_name: string (nullable = true)
|-- skills: array (nullable = true)
|   |-- element: string (containsNull = true)
```


- 🚦 **User Defined Schema** - It is possible to read “Data” faster from a “JSON File” by defining a “StructType” with “Schema Names” and “Data Types”. In this case, The “Data Types” needed to be imported from the “pyspark.sql.types” “Module”. This can be a bit complex as the “JSON Files” might have “Nested Structure”.


```

from pyspark.sql.types import ArrayType, DoubleType, IntegerType, LongType, StringType, StructType, StructField

userDefinedSchema = StructType([
    StructField("device", StringType(), True),
    StructField("ecommerce", StructType([
        StructField("purchaseRevenue", DoubleType(), True),
        StructField("total_item_quantity", LongType(), True),
        StructField("unique_items", LongType(), True)
    ]), True),
    StructField("event_name", StringType(), True),
    StructField("event_previous_timestamp", LongType(), True),
    StructField("event_timestamp", LongType(), True),
    StructField("geo", StructType([
        StructField("city", StringType(), True),
        StructField("state", StringType(), True)
    ]), True),
    StructField("items", ArrayType(
        StructType([
            StructField("coupon", StringType(), True),
            StructField("item_id", StringType(), True),
            StructField("item_name", StringType(), True),
            StructField("item_revenue_in_usd", DoubleType(), True),
            StructField("price_in_usd", DoubleType(), True),
            StructField("quantity", LongType(), True)
        ])
    ), True),
    StructField("traffic_source", StringType(), True),
    StructField("user_first_touch_timestamp", LongType(), True),
    StructField("user_id", StringType(), True)
])

```

-  **User Defined Schema Using DDL Formatted String** - Alternatively, it is possible to define the “Schema” using a “DDL-Formatted String” to read a “JSON File” and load the data into a “DataFrame”. Using a “Scala Method”, i.e., “toDDL ()” on a “Schema” the “DDL String” can be retrieved, which can be passed later in the “schema ()” method. The method “toDDL ()” is only available in “Scala”.

```

%scala

spark.read
    .option("inferSchema", "true")
    .json("dbfs:/FileStore/tables/retailer/data/single_line.json")
    .schema
    .toDDL

```

Output -

```

▶ (1) Spark Jobs
res0: String = 'address' STRUCT<'city': STRING, 'country': STRING, 'state': STRING>, 'birthday' STRING, 'email' STRING, 'first_name' STRING, 'id' BIGINT, 'last_name' STRING, 'skills' ARRAY<STRING>

```

Reading a JSON File in Single-Line and Multi-Line Using DataFrameReader

🚦 Reading a JSON File in Single-Line Mode - In a “Single-Line JSON File”, there is one Object per Line.

```
{"string": "string1", "int": 1, "array": [1, 2, 3], "dict": {"key": "value1"}}
```

In “Single-Line” mode, a “JSON File” can be split into many parts and read in “Parallel”. No special option needs to be provided to read a “Single Line JSON File”.

```
jsonDf = spark.read\  
    .option("inferSchema", "true")\  
    .json("dbfs:/FileStore/tables/retailer/data/single_line.json")
```

Output -

```
▶ (1) Spark Jobs  
▼ jsonDf: pyspark.sql.dataframe.DataFrame  
  ▼ address: struct  
    |   city: string  
    |   country: string  
    |   state: string  
    | birthday: string  
    | email: string  
    | first_name: string  
    | id: long  
    | last_name: string  
  ▼ skills: array  
    |   element: string
```

🚦 Reading a JSON File in Multi-Line Mode - In a “Multi-Line JSON File”, one Object occupies multiple Lines.

```
[
  {"string": "string1", "int": 1, "array": [1, 2, 3], "dict": {"key": "value1"}},
  {"string": "string2", "int": 2, "array": [2, 4, 6], "dict": {"key": "value2"}},
  {
    "string": "string3",
    "int": 3,
    "array": [
      3,
      6,
      9
    ],
    "dict": {
      "key": "value3",
      "extra_key": "extra_value3"
    }
  }
]
```

In “Multi-Line” mode, a “JSON File” is loaded as a whole entity and cannot be split. To read a “Multi Line JSON File”, enable the “Multi Line” mode by passing the option “multiline” as “true” to the “option” method.

```
multiLineJsonDf = spark.read\
    .option("multiline", "true")\
    .json("dbfs:/FileStore/tables/retailer/data/multi_line.json")
```

Output -

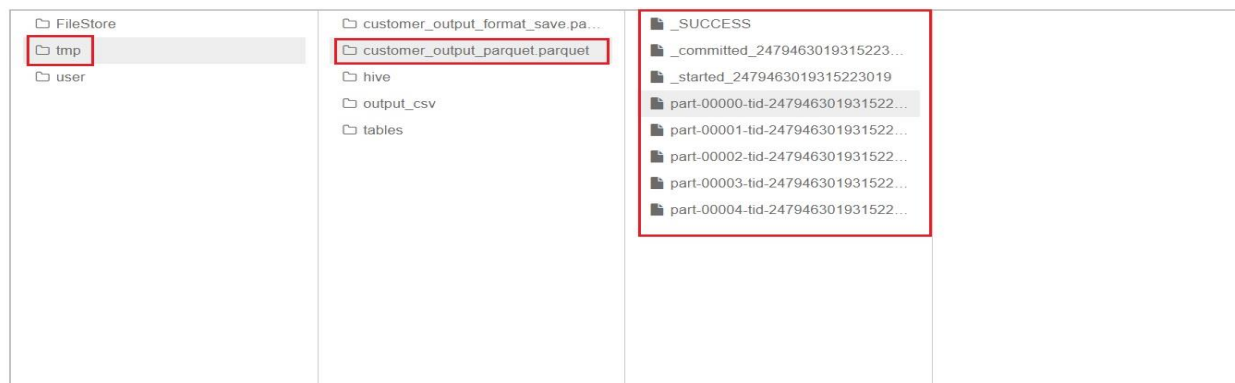
```
▶ (1) Spark Jobs
▼ multiLineJsonDf: pyspark.sql.dataframe.DataFrame
  ▼ address: struct
    |   city: string
    |   country: string
    |   state: string
    birthday: string
    email: string
    first_name: string
    id: long
    last_name: string
  ▼ skills: array
    |   element: string
```

Writing a DataFrame to File Using DataFrameWriter

- ✚ **“compression” Option** - The correct “Compression” option needs to be provided. The “compression” option can be - “snappy”, “zip” etc.
- ✚ **“mode ()” Method** - The “mode ()” method defines the behavior of “save ()”, or, “saveAsTable ()” method when the “File”, or, “Table” to write to already exists. There are four types of “SaveMode” options available -
 - **Append** - Records are appended to the existing “Data” in the “File”, or, “Table” to write to.
 - **ErrorIfExists** - If the “File”, or, “Table” to write to already exists, this mode throws a “Runtime Exception”. This is the default “SaveMode” option of “save ()”, or, “saveAsTable ()” method.
 - **Ignore** - If the “File”, or, “Table” to write to already exists, this mode does not save the “Records” and does not change the existing “Data” in any way.
 - **Overwrite** - If the “File”, or, “Table” to write to already exists, this mode overwrites the existing “Data” in the “File”, or, “Table”.
- ✚ A “DataFrame” can be written to any type of “File” using “DataFrameWriter” class in two ways -
 - **Using the Method Specified for a Particular File Type** - To write the “DataFrame” “customerUserDefinedSchemaDf” to a “Parquet File”, the “parquet” method of “DataFrameWriter” class is used, along with some configuration.

```
customerUserDefinedSchemaDf.write\  
    .mode("Overwrite")\  
    .option("compression", "snappy")\  
    .parquet("dbfs:/tmp/customer_output_parquet.parquet")
```

In the provided path, i.e., “dbfs:/tmp/customer_output_parquet.parquet”, the “Part Files”, along with “Metadata Files” are saved.



[/tmp/customer_output_parquet.parquet/part-00000-tid-2479463019315223019-67364ae3-56a7-4830-b89d-936d32ea41b5-0-1-c000.snappy.parquet](#)

- Using “format ()” and “save ()” Method - The “DataFrameWriter” class has a method called “format ()”, where the type of file to write to, needs to be passed. To write the “DataFrame” “customerUserDefinedSchemaDf” to a “Parquet File”, “parquet” needs to be passed. The “Path” of the “File”, where the “Data” from “DataFrame” needs to be written, is passed as a Value to the option “path” in “Option ()” method. Finally, the “save ()” method of the “DataFrameWriter” class is called.

```
customerUserDefinedSchemaDf.write\  
    .format("parquet")\  
    .mode("Overwrite")\  
    .options\  
        path = "dbfs:/tmp/customer_output_format_save.parquet",\  
        compression = "snappy"\  
    )\  
    .save()
```

In the provided path, i.e., “dbfs:/tmp/customer_output_format_save.parquet”, the “Part Files”, along with “Metadata Files” are saved.

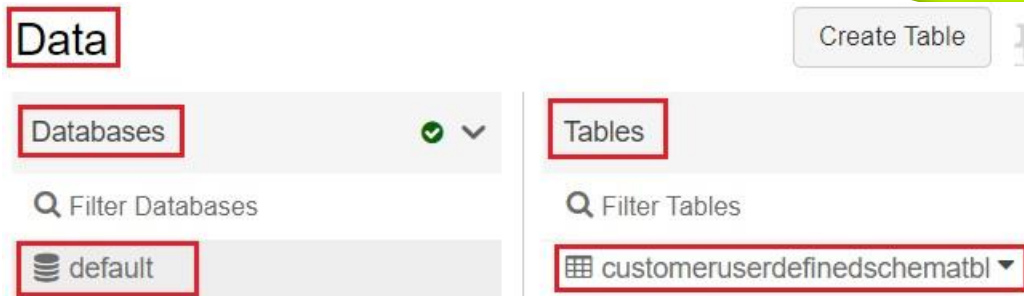


Writing a DataFrame to Table Using DataFrameWriter

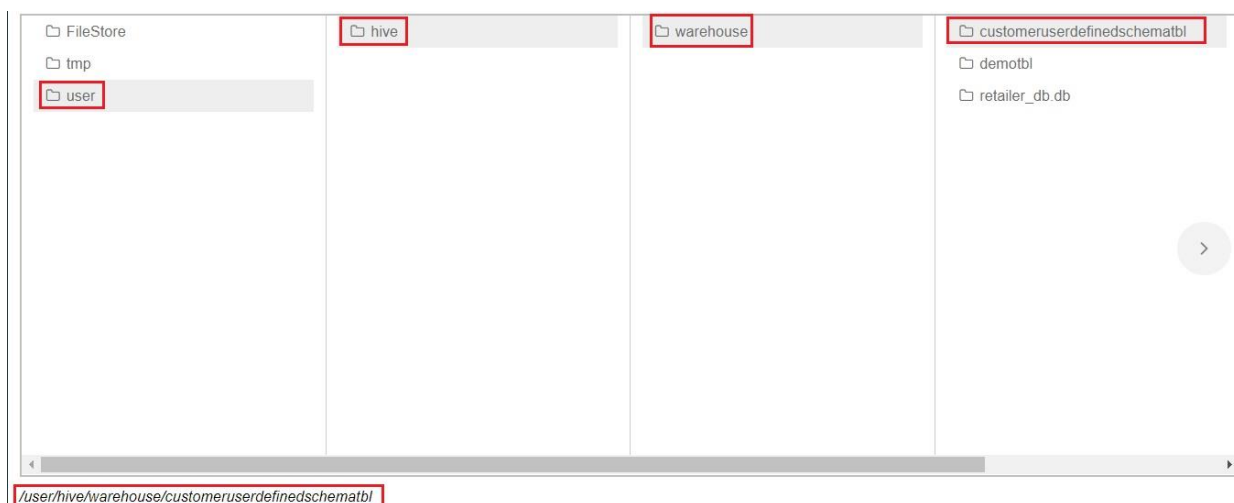
- ✚ Writing a “DataFrame” to a “Table” using the “saveAsTable” method of the “DataFrameWriter” class creates a “Global Table”.

```
customerUserDefinedSchemaDf.write\  
    .mode("overwrite")\  
    .saveAsTable("customerUserDefinedSchemaTbl")
```

- ✚ If no “Database” is specified, under which the “Table” needs to be saved, then the “Table” is saved inside the “Database” named “default”, which is created by Databricks.



- The “Underlying Files” for the created “Table” resides in the path - user/hive/warehouse/<database_name.db>/<table_name>. Since, the created “Table” is under no specific “Database”, instead under the “Database” named “default”, the part “<database_name.db>” is not appended in the “Path” of the “Directory” for the created “Table”.



- Unlike the “Local View”, also known as “Local Table”, which is created using “createTempView”, or, “createOrReplaceTempView” method of the “DataFrameWriter” class, the “Global Table” is available from outside of the “Notebook”.

Best Practice: Writing a DataFrame to Delta Table Using DataFrameWriter

- In almost all cases, the “Best Practice” is to save “DataFrames” to “Delta Lake”, especially whenever the “Data” will be referenced from a Databricks “Workspace”. Data in “Delta Table” is stored in “Parquet” format. The “save ()” method of “DataFrameWriter” class is used to store the “Data” in a “DataFrame” to a “File” in “Delta Lake”.
In this case “delta” must be mentioned in the “format ()” method.

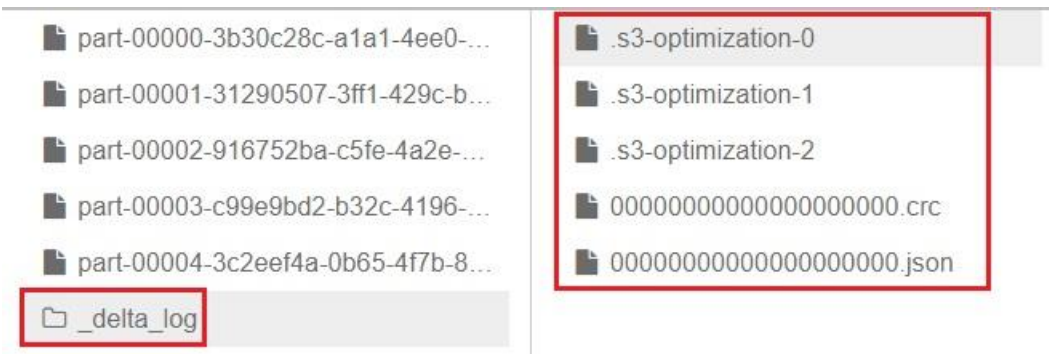
```
customerUserDefinedSchemaDf.write\
    .format("delta")\
    .mode("overwrite")\
    .save("dbfs:/tmp/delta/customer_output_delta_parquet.parquet")
```

The “Underlying Part Files” are present in the specified “Path” along with the “_delta_log” folder.



/tmp/delta/customer_output_delta_parquet.parquet

The “_delta_log” folder contains the following “Files” -



COLUMN HANDLING IN DATAFRAME

What is a Column

- ✚ A “Column” is a “Logical Construction” that will be “Computed” based on the “Data” in a “DataFrame” using a particular “Expression”.
- ✚ It is possible to “Select”, “Manipulate” and “Remove” one, or, more “Columns” from a “DataFrame”. These “Operations” are represented as “Expressions”.
- ✚ To Apache Spark, a “Column” is a “Logical Construction” that simply represents a “Value”, which is “Computed” on a “Per-Record Basis” by means of an “Expression”.
- ✚ It is not possible to “Manipulate” an individual “Column” outside the “Context” of a “DataFrame”. “Columns” can only be “Transformed” within the “Context” of a “DataFrame”.

Access a Column

- ✚ A “Column” can be accessed using a “col”, or, “column” Function in “Python” -
 - `col("columnName")`
 - `column("columnName")`
 - `col("columnName.field")`
 - `column("columnName.field")`
- ✚ A “Column” can also be accessed using the “Syntax” specific to the “Language API” being used. The following “Expressions” reference “Columns” in a specific “DataFrame” using “Python API” -
 - `df["columnName"]`
 - `df.columnName`

```
from pyspark.sql.functions import col

customerDf.select(\
    col("c_salutation"),\
    customerDf.c_first_name,\
    customerDf["c_last_name"]\
).show()
```

Create Columns from Expression

- ✚ All “Columns” are just “Expressions”, or, a set of “Transformations” on one or more “Values” in a “Record” in a “DataFrame”.
- ✚ Following is some “Python Expressions” to create “Columns” -
 - `col("a") + col("b")`

- `col("a").desc()`
- `col("a").cast(int) * 100`

```
customerDf.select(\
    (col("c_salutation") + col("c_first_name") + col("c_last_name")).alias("Customer_Name"),\
    (col("c_last_review_date") * 10).cast("string")\
)\
.show()
```

Create Columns from Expression

- ✚ In addition to accessing an existing “Column” in a “DataFrame”, the “Operators” and “Methods” of the “Column” can be used to build “Complex Expressions”.
- ✚ The Apache Spark “Column” class API contains a number of “Operators” and “Methods” that can be used to create or modify “Columns”.
 - Some of these evaluate “Boolean Expression” that can be used to “Filter Rows” of a “DataFrame”.
 - Others can change the “Values” in a “Column”, such as “Casting to Different Data Types”, or, “Performing a Sort Operation”.


✚ Column Operators -

&	Boolean AND
	Boolean OR
+	Math Addition Operator
-	Math Subtraction Operator
*	Math Multiplication Operator
/	Math Division Operator
%	Math Modulus Operator
<	Less Than Comparison Operator
<=	Less Than and Equals To Comparison Operator
>	Greater Than Comparison Operator
>=	Greater Than and Equals To Comparison Operator
==	Equality Test Operator
!=	Inequality Test Operator

Column Methods -

alias	Gives the “Column” an “Alias”.
cast, astype	Casts the “Value” of the “Column” to a different Data Type.
isNull	An “Expression” that returns “true” if the “Column” is “NULL”.
isNotNull	An “Expression” that returns “true” if the “Column” is not “NULL”.
isNan	An “Expression” that returns “true” if the “Column” is “NaN”.
asc	Returns a “Sort Expression” based on “Ascending” order of the “Column”.
desc	Returns a “Sort Expression” based on “Descending” order of the “Column”.

DataFrame Transformation Methods

 Many “Transformation Methods” have “Alias” to provide familiar terms to SQL users. Some of the “Transformation Methods” are -

select	Returns a new “DataFrame” by computing given “Expression” for each “Element”.
drop	Returns a new “DataFrame” with a “Column” dropped.
withColumnRenamed	Returns a new “DataFrame” with a “Column” renamed.
withColumn	Returns a new “DataFrame” by adding “Column”, or, replacing the existing “Column” that has the same name.
Filter, where	Filters “Rows” using the given “Condition”.
Sort, orderBy	Returns a new “DataFrame” sorted by the given “Expression”.
dropDuplicates, distinct	Returns a new “DataFrame” with the “Duplicate Rows” removed.
limit	Returns a new “DataFrame” by taking the first “n Rows”.
groupBy	Groups the “DataFrame” using the specified “Columns”, so that “Aggregation” can be run on the specified “Columns”.

DataFrame Action Methods

🔗 “DataFrame Action Methods” are “Eagerly Evaluated”, which means that these methods “Trigger Apache Spark” to “Execute a Job”. Some of the “Action Methods” are -

show	Displays the “Top n Rows” of a “DataFrame” in a “Tabular Form”.
count	Returns the “Number of Rows” in a “DataFrame”.
Describe, summary	Computes “Basic Statistics” for the “Numeric Columns” and “String Columns” of a “DataFrame”.
First, head	Returns the “First Row”.
collect	Returns an “Array” that contains “All Rows” in a “DataFrame”.
take	Returns an “Array” of the “First n Rows” in the “DataFrame”.

DataFrame Row Methods

🔗 Some of the “Row Methods” are -

index	Returns the “First Index” of the given “Value”.
count	Returns the “Number of Occurrences” of the given “Value”.
asDict	Returns as a “Dictionary”.
row.key	Access “Fields” like “Attributes”.
row[“key”]	Access “Fields” like “Dictionary Values”.
key in row	Search through “Row Keys”.

Subset Columns

🔗 “DataFrame Transform Methods” can be used to “Subset Columns”.

- **select()** - The “select ()” method selects a “Set of Columns”, or, “Column Based Expressions”.

It is possible to pass “Strings” as “Column Names” to the “select ()” method.

```
customerDf.select(\
    "c_salutation",\
    "c_first_name",\
    "c_last_name",\
    "c_last_review_date"\
)\
.show()
```

It is also possible to specify “Column Objects” in the “select ()” method. This approach is helpful to apply “Column Methods”.

```
customerDf.select(\
    col("c_salutation").alias("Salutation"),\
    col("c_first_name").alias("First_Name"),\
    col("c_last_name").alias("Last_Name"),\
    col("c_last_review_date")\
)\
.show()
```

- **selectExpr ()** - The “selectExpr ()” method selects a “Set of SQL Expressions”, or, “Column Based Expressions”.

```
selectExprDf = customerDf.selectExpr(\
    "trim(c_salutation) in ('Mr.', 'Miss') as Salutation",\
    "c_first_name as First_Name",\
    "c_last_name as Last_Name",\
    "c_birth_country in ('Japan', 'India')"\
)\
.show()
```

- **drop ()** - The “drop ()” method returns a new “DataFrame” after dropping the given “Column”, specified as a “String”, or, “Column Object”.

```
singleColumnStrDropDf = customerDf.drop("c_salutation").show()
```

```
singleColumnClnObjDropDf = customerDf.drop(col("c_salutation")).show()
```

To drop “Multiple Columns” from a “DataFrame”, the names of the “Multiple Columns” need to be specified in “Strings”.

```
multiColumnsDropDf = customerDf.drop("c_salutation", "c_last_name", "c_review_date").show()
```

Add or Replace Columns

🧩 “DataFrame Transform Methods” can be used to “Add or Replace Columns”.

- **withColumn ()** - The “withColumn ()” method returns a new “DataFrame” by adding a “Column”.

```
addNewColDf = customerDf.withColumn("birth_country_india_japan", col("c_birth_country").isin('Japan', 'India'))\
.show()
```

It is also possible to replace the existing “Column” that has the same name using the “withColumn ()” method.

```
replaceExtColDf = customerDf.withColumn("c_last_review_date", trim(col("c_salutation")).isin("Mr.", "Miss"))\
    .show()
```

- **withColumnRenamed ()** - The “withColumnRenamed ()” method returns a new “DataFrame” with the “Column” renamed.

```
renameColDf = customerDf.withColumnRenamed("c_salutation", "Salutation")\
    .show()
```

Subset Rows

🔗 “DataFrame Transform Methods” can be used to “Subset Rows”.

- **filter ()** - The “filter ()” method filters “Rows” using the given “SQL Expression” or “Column Based Condition”.

```
filterUsingSqlExprDf = customerDf.filter("trim(c_first_name) = 'Ron'")\
    .show()
```

```
filterUsingColCondDf = customerDf.filter(trim(col("c_first_name")).isin("Ron", "Cindy", "Daniel"))\
    .show()
```

```
filterUsingColCondAndOpDf = customerDf.filter(trim(col("c_first_name")).isin("Ron", "Cindy", "Daniel")\
    & trim(col("c_birth_country")).isin("USA", "Australia"))\
    .show()
```

- **dropDuplicate ()** - The “dropDuplicate ()” method returns a new “DataFrame” with “Duplicate Rows Removed” from all the “Columns”.

```
dropDupAllColDf = customerDf.dropDuplicates().show()
```

With “dropDuplicate ()” method, it is also possible to specify the “Specific Columns” to look for “Duplicate Records” in.

```
dropDupSomeColDf = customerDf.dropDuplicates(["c_first_name", "c_salutation"]).show()
```

An “Alias” for the “dropDuplicate ()” method is “distinct ()” method.

```
dropDupAllColDf = customerDf.distinct().show()
```

- **limit ()** - The “limit ()” method returns a new “DataFrame” by taking the “First n Rows”.

```
customerDf.limit(45).show()
```

```
display(customerDf.limit(45))
```

Sort Rows

🔗 “DataFrame Transform Methods” can be used to “Sort Rows”.

- **`sort ()`** - To sort “Rows” in a “DataFrame”, the “`sort ()`” method, or, its “Alias”, i.e., the “`orderBy`” method is used. The “`sort ()`” method returns a new “DataFrame” sorted by the given “Columns” or “Expressions”.

```
customerDf.sort("c_first_name").show()
```

The “`sort ()`” method, by default, sorts in “Ascending Order”. To sort in the “Descending Order”, the “`desc ()`” method of “Column” class is used on a “Column Object”.

```
customerDf.sort(col("c_first_name").desc()).show()
```

It is also possible to sort by “Multiple Columns” at a time.

```
customerDf.sort("c_salutation", "c_first_name", "c_last_name").show()
```

It is also possible to sort by “Multiple Columns”, where each “Column” can have different “Sorting Order”, i.e., “Ascending Order” and “Descending Order”.

```
customerDf.sort(col("c_salutation").desc(), "c_first_name", col("c_last_name").desc()).show()
```