5. SPARK SQL – DATAFRAMES

A DataFrame is a distributed collection of data, which is organized into named columns. Conceptually, it is equivalent to relational tables with good optimization techniques.

A DataFrame can be constructed from an array of different sources such as Hive tables, Structured Data files, external databases, or existing RDDs. This API was designed for modern Big Data and data science applications taking inspiration from **DataFrame in R Programming** and **Pandas in Python**.

Features of DataFrame

Here is a set of few characteristic features of DataFrame:

- Ability to process the data in the size of Kilobytes to Petabytes on a single node cluster to large cluster.
- Supports different data formats (Avro, csv, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, mysql, etc).
- State of art optimization and code generation through the Spark SQL Catalyst optimizer (tree transformation framework).
- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.
- Provides API for Python, Java, Scala, and R Programming.

SQLContext

SQLContext is a class and is used for initializing the functionalities of Spark SQL. SparkContext class object (sc) is required for initializing SQLContext class object.

The following command is used for initializing the SparkContext through spark-shell.

```
$ spark-shell
```

By default, the SparkContext object is initialized with the name **sc** when the spark-shell starts.

Use the following command to create SQLContext.

```
scala> val sqlcontext = new org.apache.spark.sql.SQLContext(sc)
```

Example

Let us consider an example of employee records in a JSON file named **employee.json**. Use the following commands to create a DataFrame (df) and read a JSON document named **employee.json** with the following content.

employee.json – Place this file in the directory where the current **scala>** pointer is located.

```
{

   {"id" : "1201", "name" : "satish", "age" : "25"}
   {"id" : "1202", "name" : "krishna", "age" : "28"}
   {"id" : "1203", "name" : "amith", "age" : "39"}
   {"id" : "1204", "name" : "javed", "age" : "23"}
   {"id" : "1205", "name" : "prudvi", "age" : "23"}

}
```

DataFrame Operations

DataFrame provides a domain-specific language for structured data manipulation. Here, we include some basic examples of structured data processing using DataFrames.

Follow the steps given below to perform DataFrame operations:

Read the JSON Document

First, we have to read the JSON document. Based on this, generate a DataFrame named (dfs).

Use the following command to read the JSON document named **employee.json.** The data is shown as a table with the fields – id, name, and age.

```
scala> val dfs = sqlContext.read.json("employee.json")
```

Output: The field names are taken automatically from **employee.json**.

```
dfs: org.apache.spark.sql.DataFrame = [age: string, id: string, name: string]
```

Show the Data

If you want to see the data in the DataFrame, then use the following command.

```
scala> dfs.show()
```

Output: You can see the employee data in a tabular format.

```
<console>:22, took 0.052610 s
+---+----+-------+
|age|  id|   name|
+---+----+-------+
| 25|1201| satish|
| 28|1202|krishna|
| 39|1203|  amith|
| 23|1204|  javed|
| 23|1205| prudvi|
+---+----+-------+
```

Use printSchema Method

If you want to see the Structure (Schema) of the DataFrame, then use the following command.

```
scala> dfs.printSchema()
```

Output

```
root
 |-- age: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
```

Use Select Method

Use the following command to fetch **name-**column among three columns from the DataFrame.

```
scala> dfs.select("name").show()
```

Output: You can see the values of the **name** column.

```
<console>:22, took 0.044023 s
+-------+
|   name|
+-------+
| satish|
|krishna|
|  amith|
|  javed|
| prudvi|
+-------+
```

Use Age Filter

Use the following command for finding the employees whose age is greater than 23 (age > 23).

```
scala> dfs.filter(dfs("age") > 23).show()
```

Output

```
<console>:22, took 0.078670 s
+---+----+-------+
|age|  id|   name|
+---+----+-------+
| 25|1201| satish|
| 28|1202|krishna|
| 39|1203|  amith|
+---+----+-------+
```

Use groupBy Method

Use the following command for counting the number of employees who are of the same age.

```
scala> dfs.groupBy("age").count().show()
```

Output: two employees are having age 23.

```
<console>:22, took 5.196091 s
+---+-----+
|age|count|
+---+-----+
| 23 |    2  |
| 25 |    1  |
| 28 |    1  |
| 39 |    1  |
+---+-----+
```

Running SQL Queries Programmatically

An SQLContext enables applications to run SQL queries programmatically while running SQL functions and returns the result as a DataFrame.

Generally, in the background, SparkSQL supports two different methods for converting existing RDDs into DataFrames:

1. Inferring the Schema using Reflection

2. Programmatically specifying the Schema

Inferring the Schema using Reflection

This method uses reflection to generate the schema of an RDD that contains specific types of objects. The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame. The **case class** defines the schema of the table. The names of the arguments to the case class are read using reflection and they become the names of the columns.

Case classes can also be nested or contain complex types such as Sequences or Arrays. This RDD can be implicitly be converted to a DataFrame and then registered as a table. Tables can be used in subsequent SQL statements.

Example

Let us consider an example of employee records in a text file named **employee.txt**. Create an RDD by reading the data from text file and convert it into DataFrame using Default SQL functions.

Given Data: Take a look into the following data of a file named **employee.txt** placed it in the current respective directory where the spark shell point is running.

```
1201, satish, 25
1202, krishna, 28
1203, amith, 39
1204, javed, 23
1205, prudvi, 23
```

The following examples explain how to generate a schema using Reflections.

Start the Spark Shell

Start the Spark Shell using following command.

```
$ spark-shell
```

Create SQLContext

Generate SQLContext using the following command. Here, **sc** means SparkContext object.

```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Import SQL Functions

Use the following command to import all the SQL functions used to implicitly convert an RDD to a DataFrame.

```
scala> import sqlContext.implicts._
```

Create Case Class

Next, we have to define a schema for employee record data using a case class. The following command is used to declare the case class based on the given data (id, name, age).

```
scala> case class Employee(id: Int, name: String, age: Int)


defined class Employee
```

Create RDD and Apply Transformations

Use the following command to generate an RDD named **empl** by reading the data from **employee.txt** and converting it into DataFrame, using the Map functions.

Here, two map functions are defined. One is for splitting the text record into fields (**.map(_.split(“,”))**) and the second map function for converting individual fields (id, name, age) into one case class object (**.map(e(0).trim.toInt, e(1), e(2).trim.toInt)**).

At last, **toDF()** method is used for converting the case class object with schema into a DataFrame.

```
scala> val empl=sc.textFile("employee.txt")
.map(_.split(","))
.map(e=> employee(e(0).trim.toInt,e(1), e(2).trim.toInt))
.toDF()
```

Output

```
empl: org.apache.spark.sql.DataFrame = [id: int, name: string, age: int]
```

Store the DataFrame Data in a Table

Use the following command to store the DataFrame data into a table named **employee**. After this command, we can apply all types of SQL statements into it.

```
scala> empl.registerTempTable("employee")
```

The employee table is ready. Let us now pass some sql queries on the table using **SQLContext.sql()** method.

Select Query on DataFrame

Use the following command to select all the records from the **employee** table. Here, we use the variable **allrecords** for capturing all records data. To display those records, call **show()** method on it.

```
scala> val allrecords = sqlContext.sql("SELeCT * FROM employee")
```

To see the result data of **allrecords** DataFrame, use the following command.

```
scala> allrecords.show()
```

Output

```
+----+-------+---+
|  id|   name|age|
+----+-------+---+
|1201| satish| 25|
|1202|krishna| 28|
|1203|  amith| 39|
|1204|  javed| 23|
|1205| prudvi| 23|
+----+-------+---+
```

Where Clause SQL Query on DataFrame

Use the following command for applying **where** statement in a table. Here, the variable **agefilter** stores the records of employees whose age are between 20 and 35.

```
scala> val agefilter = sqlContext.sql("SELeCT * FROM employee WHERE age>=20 AND
age <= 35")
```

To see the result data of **agefilter** DataFrame, use the following command.

```
scala> agefilter.show()
```

Output

```
<console>:25, took 0.112757 s
+----+-------+---+
|  id|   name|age|
+----+-------+---+
|1201| satish| 25|
|1202|krishna| 28|
|1204|  javed| 23|
|1205| prudvi| 23|
+----+-------+---+
```

The previous two queries were passed against the whole table DataFrame. Now let us try to fetch data from the result DataFrame by applying **Transformations** on it.

Fetch ID values from agefilter DataFrame using column index-

The following statement is used for fetching the ID values from **agefilter** RDD result, using field index.

```
scala> agefilter.map(t=>"ID: "+t(0)).collect().foreach(println)
```

Output

```
<console>:25, took 0.093844 s

ID: 1201

ID: 1202

ID: 1204

ID: 1205
```

This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.

Programmatically Specifying the Schema

The second method for creating DataFrame is through programmatic interface that allows you to construct a schema and then apply it to an existing RDD. We can create a DataFrame programmatically using the following three steps.

- Create an RDD of Rows from an Original RDD.
- Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
- Apply the schema to the RDD of Rows via createDataFrame method provided bySQLContext.

Example

Let us consider an example of employee records in a text file named **employee.txt**. Create a Schema using DataFrame directly by reading the data from text file.

Given Data: Look at the following data of a file named **employee.txt** placed in the current respective directory where the spark shell point is running.

```
1201, satish, 25

1202, krishna, 28

1203, amith, 39

1204, javed, 23
```

```
1205, prudvi, 23
```

Follow the steps given below to generate a schema programmatically.

Open Spark Shell

Start the Spark shell using following example.

```
$ spark-shell
```

Create SQLContext Object

Generate SQLContext using the following command. Here, **sc** means SparkContext object.

```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Read Input from Text File

Create an RDD DataFrame by reading a data from the text file named **employee.txt** using the following command.

```
scala> val employee = sc.textFile("employee.txt")
```

Create an Encoded Schema in a String Format

Use the following command for creating an encoded schema in a string format. That means, assume the field structure of a table and pass the field names using some delimiter.

```
scala> val schemaString = "id name age"
```

Output

```
schemaString: String = id name age
```

Import Respective APIs

Use the following command to import Row capabilities and SQL DataTypes.

```
scala> import org.apache.spark.sql.Row;

scala> import org.apache.spark.sql.types.{StructType, StructField, StringType};
```

Generate Schema

The following command is used to generate a schema by reading the **schemaString** variable. It means you need to read each field by splitting the whole string with space as a delimiter and take each field type is String type, by default.

```
scala> val schema = StructType(schemaString.split(" ").map(fieldName =>
StructField(fieldName, StringType, true)))
```

Apply Transformation for Reading Data from Text File

Use the following command to convert an RDD (employee) to Rows. It means, here we are specifying the logic for reading the RDD data and store it into rowRDD. Here we are using two map functions: one is a delimiter for splitting the record string (**.map(_.split(","))**) and the second map function for defining a Row with the field index value (**.map(e => Row(e(0).trim.toInt, e(1), e(2).trim.toInt))**).

```
scala> val rowRDD = employee.map(_.split(",")).map(e => Row(e(0).trim.toInt,
e(1), e(2).trim.toInt))
```

Apply RowRDD in Row Data based on Schema

Use the following statement for creating a DataFrame using **rowRDD** data and **schema** (SCHEMA) variable.

```
scala> val employeeDF = sqlContext.createDataFrame(rowRDD, schema)
```

Output

```
employeeDF: org.apache.spark.sql.DataFrame = [id: string, name: string, age:
string]
```

Store DataFrame Data into Table

Use the following command to store the DataFrame into a table named **employee**.

```
scala> employeeDF.registerTempTable("employee")
```

The **employee** table is now ready. Let us pass some SQL queries into the table using the method **SQLContext.sql()**.

Select Query on DataFrame

Use the following statement for selecting all records from the **employee** table. Here we use the variable **allrecords** for capturing all records data. To display those records, call **show()** method on it.

```
scala> val allrecords = sqlContext.sql("SELECT * FROM employee")
```

To see the result data of **allrecords** DataFrame, use the following command.

```
scala> allrecords.show()
```

Output

```
+----+--------+----+

|  id|    name|age|
+----+-------+---+
|1201|  satish| 25|
|1202| krishna| 28|
|1203|   amith| 39|
|1204|   javed| 23|
|1205|  prudvi| 23|
+----+-------+---+
```

The method **sqlContext.sql** allows you to construct DataFrames when the columns and their types are not known until runtime. Now you can run different SQL queries into it.