Spark SQL

By. Mr. Gopal Krishna

Sr.Hadoop Architect

CCA 175 – Spark and Hadoop Developer Certified Consultant

- ✓ Spark SQL is a Spark library that runs on top of Spark.
- ✓ It provides a higher-level abstraction than the Spark core API for processing structured data.
- ✓ Structured data includes data stored in a database, NoSQL data store, Parquet, ORC, Avro, JSON, CSV,
  or any other structured format

- ✓ Spark SQL is more than just about providing SQL interface to Spark.
- ✓ It was designed with the broader goals of making Spark easier to use, increasing developer productivity, and making Spark applications run faster.
- ✓ Spark SQL can be used as a library for developing data processing applications in Scala, Java, Python, or R.
- ✓ It supports multiple query languages, including SQL, HiveQL, and language integrated queries

Hive Interoperability:

- ✓ Spark SQL is compatible with Hive.
- ✓ It not only supports HiveQL, but can also access Hive metastore,
  SerDes, and UDFs. Therefore, if you have an existing Hive deployment, you can use Spark SQL alongside Hive.
- ✓ We need not to move data or make any changes to your existing Hive metastore.
- ✓ We can also replace Hive with Spark SQL to get better performance.
- ✓ Since Spark SQL supports HiveQL and Hive metastore, existing Hive workloads can be easily migrated to Spark SQL. HiveQL queries run much faster on Spark SQL than on Hive.

- ✓ Starting with version 1.4.0, Spark SQL supports multiple versions of Hive.
- ✓ It can be configured to read Hive metastores created with different versions of Hive.
- ✓ Hive is not required to use Spark SQL. You can use Spark SQL with or without Hive. It has a
- ✓ built-in HiveQL parser. In addition, if you do not have an existing Hive metastore, Spark SQL creates one

```
scala> val resultSet = sqlContext.sql("show tables")
16/09/19 00:24:25 INFO HiveMetaStore: 0: get_tables: db=default pat=.*
16/09/19 00:24:25 INFO audit: ugi=gopal	ip=unknown-ip-addr	cmd=get_tables: db=default
pat=.*
resultSet: org.apache.spark.sql.DataFrame = [tableName: string, isTemporary: boolean]
```

HiveContext:

- ✓ HiveContext is an alternative entry point into the Spark SQL library.
- ✓ It extends the SQLContext class for processing data stored in Hive.
- ✓ It also provides a HiveQL parser. A Spark SQL application must create an instance of either this class or the SQLContext class

HiveContext provides a superset of the functionality provided by SQLContext. The parser that comes with HiveContext is more powerful than the SQLContext parser. It can execute both HiveQL and SQL queries. It can read data from Hive tables. It also allows applications to access Hive UDFs (user-defined functions).

if we want to process existing Hive tables, add hive-site.xml file to Spark's classpath, since HiveContext reads Hive configuration from the hive-site.xml file.

Data Frames:

DataFrame is Spark SQL's primary data abstraction.
It represents a distributed collection of rows organized into named columns..
Conceptually, it is similar to a table in a relational database.

Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine.

1. A DataFrame is a distributed collection of data organized into named columns
2. It is conceptually equivalent to a table in a relational database or a data frame in R/Python.
3. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs

Converting "RDD" to "DataFrames [DF]"

- ✓ Spark SQL provides an implicit conversion method named toDF, which creates a DataFrame from an RDD of objects represented by a case class [To Define Schema Only we are using Case Class]
- ✓ When this technique is used, Spark SQL infers the schema of a dataset.
- ✓ The toDF method is not defined in the RDD class, but it is available through an implicit conversion.
- ✓ To convert an RDD to a DataFrame using toDF, we need to import the implicit methods defined in the implicits object

Examples

```
scala> val data = sc.parallelize(1 to 10)
data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:21

scala> val newData = data.map(l => (l , l+10) )
newData: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[1] at map at <console>:23

scala> val resultData = newData.toDF("original","transformed")
resultData: org.apache.spark.sql.DataFrame = [original: int, transformed: int]

scala> resultData.printSchema
root
 |-- original: integer (nullable = false)
 |-- transformed: integer (nullable = false)

scala> resultData.show
+--------+-----------+
|original|transformed|
+--------+-----------+
|       1|         11|
|       2|         12|
|       3|         13|
|       4|         14|
|       5|         15|
|       6|         16|
|       7|         17|
|       8|         18|
|       9|         19|
|      10|         20|
+--------+-----------+


scala> val data = List("Spark","Scala","Java","Phython")
data: List[String] = List(Spark, Scala, Java, Phython)

scala> val newData = data.map( l => (l , l.length))
newData: List[(String, Int)] = List((Spark,5), (Scala,5), (Java,4), (Phython,7))

scala> val resultData = newData.toDF("Name" , "Length")
resultData: org.apache.spark.sql.DataFrame = [Name: string, Length: int]

scala> resultData.printSchema
root
 |-- Name: string (nullable = true)
 |-- Length: integer (nullable = false)

scala> resultData.show
+-------+------+
|   Name|Length|
+-------+------+
|  Spark|     5|
|  Scala|     5|
|   Java|     4|
|Phython|     7|
+-------+------+
```

```
scala> val numList = List(1,2,3,4,5)
numList: List[Int] = List(1, 2, 3, 4, 5)

scala> val numRDD = sc.parallelize(numList)
numRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[77] at parallelize at <console>:26

scala> val numDF = numRDD.toDF
numDF: org.apache.spark.sql.DataFrame = [_1: int]

scala> numDF.show

+--+
|_1|
+--+
| 1|
| 2|
| 3|
| 4|
| 5|
+--+
```

```
scala> val data = sc.parallelize(List( ("Spark", 1),("Scala",2),("Phython",3),("R",4) ))
data: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[7] at parallelize at
<console>:21

scala> val resultData = data.toDF("Technology","Rank")
resultData: org.apache.spark.sql.DataFrame = [Technology: string, Rank: int]

scala> resultData.printSchema
root
 |-- Technology: string (nullable = true)
 |-- Rank: integer (nullable = false)


scala> resultData.show

+----------+----+
|Technology|Rank|
+----------+----+
|     Spark|   1     |
|     Scala|   2      |
|   Phython|   3    |
|         R|   4       |
+----------+----+

scala> resultData.registerTempTable("techtab")

scala>

scala> val tabList = sqlContext.sql("show tables")

scala> tabList.show

+---------+-----------+
|tableName|isTemporary|
+---------+-----------+
|  techtab|       true|
|     dept|      false|
|    dept1|      false|
|  depttab|      false|
|      emp|      false|
|   emptab|      false|
|  testtab|      false|
+---------+-----------+
scala> val bestTech = sqlContext.sql("select * from techtab")
16/09/26 00:02:56 INFO ParseDriver: Parsing command: select * from techtab
16/09/26 00:02:58 INFO ParseDriver: Parse Completed
bestTech: org.apache.spark.sql.DataFrame = [Technology: string, Rank: int]

scala> bestTech.show
+----------+----+
|Technology|Rank|
+----------+----+
|     Spark|   1|
|     Scala|   2|
|   Phython|   3|
|         R|   4|
+----------+----+
scala> val aTab = sqlContext.sql("select * from techtab where Technology like '%a%'")

scala> aTab.show

+----------+----+
|Technology|Rank|
+----------+----+
|     Spark|   1|
|     Scala|   2|
+----------+----+
```

```
scala> case class Line(x:Int,y:String)
defined class Line

scala> val data = sqlContext.createDataFrame( (1 to 100).map( l => Line( l , s"val_$l")) )
data: org.apache.spark.sql.DataFrame = [x: int, y: string]

scala> data.registerTempTable("LineTab")

scala> sqlContext.sql("select * from linetab").show()

+--+------+
| x|     y|
+--+------+
| 1| val_1|
| 2| val_2|
| 3| val_3|
| 4| val_4|
| 5| val_5|
| 6| val_6|
| 7| val_7|
| 8| val_8|
| 9| val_9|
|10|val_10|
|11|val_11|
|12|val_12|
|13|val_13|
|14|val_14|
|15|val_15|
|16|val_16|
|17|val_17|
|18|val_18|
|19|val_19|
|20|val_20|
+--+------+


scala> val data = sqlContext.createDataFrame( (1 to 100).map( l => Line( l , (l+2).toString())) )
data: org.apache.spark.sql.DataFrame = [x: int, y: string]

scala> data.registerTempTable("LineTab")

scala> sqlContext.sql("select * from linetab").show()

+--+--+
| x| y|
+--+--+
| 1| 3|
| 2| 4|
| 3| 5|
| 4| 6|
| 5| 7|
| 6| 8|
| 7| 9|
| 8|10|
| 9|11|
|10|12|
|11|13|
|12|14|
|13|15|
|14|16|
|15|17|
|16|18|
|17|19|
|18|20|
```

NOTE: show command will only display the first 20 values.

```
scala> sqlContext.sql("select * from linetab WHERE y > 15").show()

+--+--+
| x| y|
+--+--+
|14|16|
|15|17|
|16|18|
|17|19|
|18|20|
|19|21|
|20|22|
|21|23|
|22|24|
|23|25|
|24|26|
|25|27|
|26|28|
|27|29|
|28|30|
|29|31|
|30|32|
|31|33|
|32|34|
|33|35|
+--+--+

scala> sqlContext.sql("select * from linetab WHERE y > 15 AND y < 22").show()
+--+--+
| x| y|
+--+--+
|14|16|
|15|17|
|16|18|
|17|19|
|18|20|
|19|21|
+--+--+
```

To get all the values we have to use foreach()..

```
scala> sqlContext.sql("select * from linetab WHERE x >15").collect().foreach(println)
```

INPUTDATA:

```
gopal@ubuntu:~/SparkSQLInput$ cat EmpDetails.txt
   1000 Gopal      12000
1001    Krishna 14000
1002    Ravi     16000
  1002  Ramya  18000
1003    Rakesh    20000
  1004  Rajesh  22000
1005    Mounika              24000
1006    Sravya  26000
  1007  Siya     28000
1008    Dixitha    30000
   1009 Trinath 34000

scala> val empdata = sc.textFile("/home/gopal/SparkSQLInput/EmpDetails.txt")

scala> case class Employee(id:Int,name:String,sal:Int)
defined class Employee

scala> val newdata = empdata.map(_.split("\t"))
newdata: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[26] at map

scala> val resultdata = newdata.map( e => Employee( e(0).trim.toInt , e(1),e(2).trim.toInt )).toDF()
resultdata: org.apache.spark.sql.DataFrame = [id: int, name: string, sal: int]

scala> resultdata.printSchema
root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- sal: integer (nullable = false)

scala> resultdata.show

+----+-------+-----+
|  id|   name|  sal|
+----+-------+-----+
|1000|  Gopal|12000|
|1001|Krishna|14000|
|1002|   Ravi|16000|
|1002|  Ramya|18000|
|1003| Rakesh|20000|
|1004| Rajesh|22000|
|1005|Mounika|24000|
|1006| Sravya|26000|
|1007|   Siya|28000|
|1008|Dixitha|30000|
|1009|Trinath|34000|
+----+-------+-----+

scala> resultdata.registerTempTable("employeetab")
scala> val entrylevelemp = sqlContext.sql("select * from employeetab WHERE sal > 15000 AND sal <
30000").show
+----+-------+-----+
|  id|   name|  sal|
+----+-------+-----+
|1002|   Ravi|16000|
|1002|  Ramya|18000|
|1003| Rakesh|20000|
|1004| Rajesh|22000|
|1005|Mounika|24000|
|1006| Sravya|26000|
|1007|   Siya|28000|
+----+-------+-----+
```

INPUT DATA:

```
gopal@ubuntu:~/SparkSQLInput$ cat AccountDetails.txt
10101,Gopal,Savings,ICICI,50000
10201,Krishna,Current,SBI,56000
10301,Ravi,Savings,HDFC,67000
20201,Ramya,Savings,SBI,89000
30303,Radha,Fixed,ICICI,560000
50607,Sruthi,Current,AXIS,45000
70007,Nalini,Fixed,HDFC,66000
33333,Raj,Savings,AXIS,59500
20201,Ramya,Savings,SBI,89000
55303,Rakhee,Fixed,ICICI,560000
50607,Santhosh,Current,AXIS,45000
88007,Nandha,Fixed,HDFC,65900
44333,Raj,Savings,SBI,79500
gopal@ubuntu:~/SparkSQLInput$

scala> import sqlContext.implicits._
import sqlContext.implicits._

scala> val accdetails = sc.textFile("/home/gopal/SparkSQLInput/AccountDetails.txt")

scala> case class Account(acc_id:Int,acc_holder:String,acc_type:String,bankname:String,amt:Int)
defined class Account

scala> val newAccData = accdetails.map(_.split(",")).map(a =>
Account(a(0).trim.toInt,a(1),a(2),a(3),a(4).trim.toInt)).toDF()

newAccData: org.apache.spark.sql.DataFrame = [acc_id: int, acc_holder: string, acc_type: string,
bankname: string, amt: int]

scala> newAccData.registerTempTable("accounttab")

scala> newAccData.show

+------+----------+--------+--------+------+
|acc_id|acc_holder|acc_type|bankname|   amt|
+------+----------+--------+--------+------+
| 10101|     Gopal| Savings|   ICICI| 50000|
| 10201|   Krishna| Current|     SBI| 56000|
| 10301|      Ravi| Savings|    HDFC| 67000|
| 20201|     Ramya| Savings|     SBI| 89000|
| 30303|    Radha|   Fixed|   ICICI|560000|
| 50607|   Sruthi| Current|    AXIS| 45000|
| 70007|   Nalini|   Fixed|    HDFC| 66000|
| 33333|      Raj| Savings|    AXIS| 59500|
| 20201|    Ramya| Savings|     SBI| 89000|
| 55303|   Rakhee|   Fixed|   ICICI|560000|
| 50607| Santhosh| Current|    AXIS| 45000|
| 88007|   Nandha|   Fixed|    HDFC| 65900|
| 44333|      Raj| Savings|     SBI| 79500|
+------+----------+--------+--------+------+


scala> newAccData.groupBy("acc_type").count().show

+--------+-----+
|acc_type|count|
+--------+-----+
| Savings|    6   |
| Current|    3   |
|   Fixed|    4    |
+--------+-----+

scala> sqlContext.sql("select bankname , count(*) from accounttab group by bankname").show

+--------+---+
|bankname|_c1|
+--------+---+
|    AXIS|  3    |
|    HDFC|  3   |
|     SBI|  4     |
|   ICICI|  3    |
+--------+---+
```

DIFF WAYS OF "LOADING" & "SAVING" DATA USING SPARK SQL

- ✓ Spark SQL provides a unified interface for creating a DataFrame from a variety of data sources
- ✓ Same API can be used to create a DataFrame from a Parquet, JSON, ORC or CSV file on local file system, HDFS or S3
- ✓ Spark SQL provides a class named DataFrameReader, which defines the interface for reading data from a data source. It allows you to specify different options for reading data

NOTE: The default data source is Parquet , if NO Other format is specified. PFB Example for the same:

```
scala> val data = sqlContext.read.load("/home/gopal/Input.log")
java.io.IOException: Could not read footer: java.lang.RuntimeException:
file:/home/gopal/Input.log is not a Parquet file
```

Parquet Files:

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data.

Parquet is a popular column-oriented storage format that can store records with nested fields efficiently. It is often used with tools in the Hadoop ecosystem, and it supports all of the data types in Spark SQL. Spark SQL provides methods for reading data directly to and from Parquet files.

Apache Parquet is a columnar storage format for the Hadoop ecosystem. Since its inception about 2 years ago, Parquet has gotten very good adoption due to the highly efficient compression and encoding schemes used that demonstrate significant performance benefits. Its ground-up design allows it to be used regardless of any data processing framework, data model, and programming language used in Hadoop ecosystem. A variety of tools and frameworks including MapReduce, Hive, Impala, and Pig provided the ability to work with Parquet data and a number of data models such as AVRO, Protobuf, and Thrift have been expanded to be used with Parquet as storage format. Parquet is widely adopted by a number of major companies including tech giants such as Twitter and Netflix.

To Save the File as a Parquet file , use the below method.

```
people.saveAsParquetFile("people.parquet")
```

Examples On Parquet File

```
scala> val parquetFile =
sqlContext.parquetFile("/home/gopal/SparkSQLInput/users.parquet")
parquetFile: org.apache.spark.sql.DataFrame = [name: string, favorite_color:
string, favorite_numbers: array<int>]
scala> parquetFile.registerTempTable("parquetFile")


scala> parquetFile.printSchema
root
 |-- name: string (nullable = false)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = false)
 |    |-- element: integer (containsNull = false)
scala> val selectedPeople = sqlContext.sql("SELECT name FROM parquetFile")
scala> selectedPeople.map(t => "Name: " + t(0)).collect().foreach(println)
```

OUTPUT:

```
Name: Alyssa
Name: Ben
```

Alternative Way:

```
scala> val selectedPeople = sqlContext.sql("SELECT name FROM parquetFile").show
+------+
|  name|
+------+
|Alyssa|
|   Ben|
+------+
```

How to Save the Data in a “ParquetFile” format

```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@fb0ff

scala> val df = sqlContext.read.load("/home/gopal/SparkSQLInput/users.parquet")
df: org.apache.spark.sql.DataFrame = [name: string, favorite_color: string,
favorite_numbers: array<int>]

df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```