This is the second article of the "Big Data Processing with Apache Spark" series. Please see also: Part 1: Introduction, Part 3: Spark Streaming, Part 4: Spark Machine Learning, Part 5: Spark ML Data Pipelines, Part 6: Graph Data Analytics with Spark GraphX.

In the previous article of the Apache Spark article series, we learned what Apache Spark framework is and how it helps with big data processing analytics needs in the organizations.

Spark SQL, part of Apache Spark big data framework, is used for structured data processing and allows running SQL like queries on Spark data. We can perform ETL on the data from different formats like JSON, Parquet, Database) and then run ad-hoc querying.

In this second installment of the article series, we'll look at the Spark SQL library, how it can be used for executing SQL queries against the data stored in batch files, JSON data sets, or Hive tables.

Spark 1.3 is the latest version of the big data framework which was released last month. Prior to this version, Spark SQL module has been in an “Alpha” status but now the team has removed that label from the library. This release includes several new features some of which are listed below:

- **DataFrame:** The new release provides a programming abstraction called DataFrames which can act as distributed SQL query engine.
- **Data Sources:** With the addition of the data sources API, Spark SQL now makes it easier to compute over structured data stored in a wide variety of formats, including Parquet, JSON, and Apache Avro library.
- **JDBC Server:** The built-in JDBC server makes it easy to connect to the structured data stored in relational database tables and perform big data analytics using the traditional BI tools.

Spark SQL Components

The two main components when using Spark SQL are DataFrame and SQLContext.

Let’s look at DataFrame first.

DataFrame

A DataFrame is a distributed collection of data organized into named columns. It is based on the data frame concept in R language and is similar to a database table in a relational database.

SchemaRDD in prior versions of Spark SQL API, has been renamed to DataFrame.

DataFrames can be converted to RDDs by calling the rdd method which returns the content of the DataFrame as an RDD of Rows.

DataFrames can be created from different data sources such as:

- Existing RDDs

- Structured data files
- JSON datasets
- Hive tables
- External databases

Spark SQL and DataFrame API are available in the following programming languages:

- Scala (https://spark.apache.org/docs/1.3.0/api/scala/index.html#org.apache.spark.sql.package
- Java (https://spark.apache.org/docs/1.3.0/api/java/index.html?org/apache/spark/sql/api/java/package-summary.html)
- Python (https://spark.apache.org/docs/1.3.0/api/python/pyspark.sql.html)

Spark SQL code examples we discuss in this article use the Spark Scala Shell program.

SQLContext

Spark SQL provides SQLContext to encapsulate all relational functionality in Spark. You create the SQLContext from the existing SparkContext that we have seen in the previous examples. Following code snippet shows how to create a SQLContext object.

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

There is also HiveContext which provides a superset of the functionality provided by SQLContext. It can be used to write queries using the HiveQL parser and read data from Hive tables.

Note that you don't need an existing Hive environment to use the HiveContext in Spark programs.

JDBC Datasource

Other features in Spark SQL library include the data sources including the JDBC data source.

JDBC data source can be used to read data from relational databases using JDBC API. This approach is preferred over using the JdbcRDD because the data source returns the results as a DataFrame which can be processed in Spark SQL or joined with other data sources.

Sample Spark SQL Application

In the previous article, we learned how to install the Spark framework on the local machine, how to launch it and interact with it using Spark Scala Shell program. To install the latest version of Spark, download the software from their website.

For the code examples in this article, we will use the same Spark Shell to execute the Spark SQL programs. These code examples are for Windows environment. If you are using

To make sure Spark Shell program has enough memory, use the driver-memory command line argument when running spark-shell, as shown in the following command.

```
spark-shell.cmd --driver-memory 1G
```

Spark SQL Application

Once you have Spark Shell launched, you can run the data analytics queries using Spark SQL API.

In the first example, we'll load the customer data from a text file and create a DataFrame object from the dataset. Then we can run DataFrame functions as specific queries to select the data.

Let's look at the contents of the text file called customers.txt shown below.

```
100, John Smith, Austin, TX, 78727
200, Joe Johnson, Dallas, TX, 75201
300, Bob Jones, Houston, TX, 77028
400, Andy Davis, San Antonio, TX, 78227
500, James Williams, Austin, TX, 78727
```

Following code snippet shows the Spark SQL commands you can run on the Spark Shell console.

```
// Create the SQLContext first from the existing Spark Context
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Import statement to implicitly convert an RDD to a DataFrame
import sqlContext.implicits._

// Create a custom class to represent the Customer
case class Customer(customer_id: Int, name: String, city: String, state: String

// Create a DataFrame of Customer objects from the dataset text file.
val dfCustomers = sc.textFile("data/customers.txt").map(_.split(",")).map(p =>

// Register DataFrame as a table.
dfCustomers.registerTempTable("customers")

// Display the content of DataFrame
dfCustomers.show()

// Print the DF schema
dfCustomers.printSchema()

// Select customer name column
dfCustomers.select("name").show()

// Select customer name and city columns
dfCustomers.select("name", "city").show()

// Select a customer by id
dfCustomers.filter(dfCustomers("customer_id").equalTo(500)).show()

// Count the customers by zip code
dfCustomers.groupBy("zip_code").count().show()
```

In the above example, the schema is inferred using the reflection. We can also programmatically specify the schema of the dataset. This is useful when the custom classes cannot be defined

ahead of time because the structure of data is encoded in a string.

Following code example shows how to specify the schema using the new data type classes StructType, StringType, and StructField.

```
//
// Programmatically Specifying the Schema
//

// Create SQLContext from the existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Create an RDD
val rddCustomers = sc.textFile("data/customers.txt")

// The schema is encoded in a string
val schemaString = "customer_id name city state zip_code"

// Import Spark SQL data types and Row.
import org.apache.spark.sql._

import org.apache.spark.sql.types._;

// Generate the schema based on the string of schema
val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fi

// Convert records of the RDD (rddCustomers) to Rows.
val rowRDD = rddCustomers.map(_.split(",")).map(p => Row(p(0).trim,p(1),p(2),p(

// Apply the schema to the RDD.
val dfCustomers = sqlContext.createDataFrame(rowRDD, schema)

// Register the DataFrames as a table.
dfCustomers.registerTempTable("customers")

// SQL statements can be run by using the sql methods provided by sqlContext.
val custNames = sqlContext.sql("SELECT name FROM customers")

// The results of SQL queries are DataFrames and support all the normal RDD ope
// The columns of a row in the result can be accessed by ordinal.
custNames.map(t => "Name: " + t(0)).collect().foreach(println)

// SQL statements can be run by using the sql methods provided by sqlContext.
val customersByCity = sqlContext.sql("SELECT name,zip_code FROM customers ORDER

// The results of SQL queries are DataFrames and support all the normal RDD ope
```

```
// The columns of a row in the result can be accessed by ordinal.
customersByCity.map(t => t(0) + "," + t(1)).collect().foreach(println)
```

You can also load the data from other data sources like JSON data files, Hive tables, or even relational database tables using the JDBC data source.

As you can see, Spark SQL provides a nice SQL interface to interact with data that's loaded from diverse data sources, using the SQL query syntax which is familiar to the teams. This is especially useful for non-technical project members like data analysts as well as DBAs.

Conclusions

In this article, we looked at how Apache Spark SQL works to provide an SQL interface to interact with Spark data using the familiar SQL query syntax. Spark SQL is a powerful library that non-technical team members like Business and Data Analysts can use to run data analytics in their organizations.

In the next article, we'll look at the Spark Streaming library which can be used for processing real-time data or streaming data. This library is another important part of the overall data processing and management lifecycle in any organization because the streaming data processing gives us the real-time insights into the systems. This is critical for use cases like fraud detection, online trading systems, event processing solutions etc.

References

- Spark Main Website
- Spark SQL web site
- Spark SQL Programming Guide
- Big Data Processing using Apache Spark - Part 1: Introduction

About the Author

Srini Penchikala currently works as Software Architect at a financial services organization in Austin, Texas. He has over 20 years of experience in software architecture, design and development. Srini is currently authoring a book on NoSQL Database Patterns topic. He is also the co-author of "Spring Roo in Action" book from Manning Publications. He has presented at conferences like JavaOne, SEI Architecture Technology Conference (SATURN), IT Architect Conference (ITARC), No Fluff Just Stuff, NoSQL Now and Project World Conference. Srini also published several articles on software architecture, security and risk management, and NoSQL databases on websites like InfoQ, The ServerSide, OReilly Network (ONJava), DevX Java, java.net and JavaWorld. He is a Lead Editor for NoSQL Databases community at InfoQ.

This is the second article of the "Big Data Processing with Apache Spark" series. Please see also: Part 1: Introduction, Part 2: Spark SQL, Part 3: Spark Streaming, Part 4: Spark Machine Learning, Part 5: Spark ML Data Pipelines, Part 6: Graph Data Analytics with Spark GraphX.