Spark SQL

Introduction

- Spark SQL is a module for structured data processing. It provides Spark with more information about the both the data and the computation being performed.
- We can interact with Spark SQL with *SQL*, *`Dataset`* (and *`DataFrames`*) API and other ways.
  - We can interact with the SQL interface via a programming language and the data set will be returned as a `Dataset`/`DataFrame`. We can also interact with the SQL interface using the *command-line* or over JDBC/ODBC.
- When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth.

Creating DataFrames

- With a SparkSession, we can create DataFrames from an existing RDD, from a Hive table or from other Spark data sources.
- JavaSparkSQLExample1.java in the folder `Spark/java-sql-example`
- DataFrames provide a domain-specific language for structured data manipulation. DataFrames are just Dataset of Rows in Scala and Java API. These operations are also referred as "untyped transformations" in contrast to "typed transformations" that come with strongly typed Scala/Java Datasets.
- See same example as above for some basic examples of structured data processing using Datasets. Here are some code snippets:

```
df.printSchema();

// Select only the "name" column
df.select("name").show();

// Select everybody, but increment the age by 1
df.select(col("name"), col("age").plus(1)).show();

// Select people older than 21
df.filter(col("age").gt(21)).show();

// Count people by age
df.groupBy("age").count().show();
```

Running SQL Queries Programmatically

- The `sql` function on a `SparkSession` enables applications to run SQL queries programmatically and returns the result as a `Dataset<Row>`. See same example as before. Here is a code snippet.

```
df.createOrReplaceTempView("people");
Dataset<Row> sqlDF = spark.sql("SELECT * FROM people"
    );
sqlDF.show();
```

- Create a global temporary view to have one that is shared among all sessions while a Spark application runs. Temporary views (like above) are session-scoped and will disappear if the session that creates it terminates.

```
df.createGlobalTempView("people");
spark.sql("SELECT * FROM global_temp.people").show();
spark.newSession().sql("SELECT * FROM global_temp.
    people").show();
```

Creating `Datasets`

- `Datasets` are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized `Encoder` to serialize the objects for processing or transmitting over the network
- Encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.
- See example: JavaSQLExample2.java

Interoperability with RDDs (1)

- Spark SQL supports two different methods for converting existing RDDs into `Dataset`s.
  - Use reflection to infer the schema of an RDD that contains specific types of objects. This reflection-based approach leads to more concise code and works well when we already know the schema.
  - Use a programmatic interface that allows us to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows us to construct `Dataset`s when the columns and their types are not known until run time.

Interoperability with RDDs (2)

- Spark SQL supports automatically converting an RDD of `JavaBean`s into a `DataFrame`.
- The `BeanInfo`, obtained using reflection, defines the schema of the table. Currently, Spark SQL does not support `JavaBean`s that contain Map field(s). Nested `JavaBean`s and `List` or `Array` fields are supported though.
- See the method `runInferSchemaExample` in JavaSQLExample3.java

What is a JavaBean?

- It is a reusable software component written in Java.
- JavaBeans are classes that encapsulate many objects into a single object (the bean) by following a standard. The name "Bean" was given to encompass this standard, which is defined below:
  - Must implement Serializable
  - It should have a public no-arg constructor
  - All attributes must be private with public getters and setter methods
  - Rules for setter methods:
    - It should be public
    - The return-type should be void
    - The method name should be prefixed with set
    - It should take some argument i.e. it should not be no-arg method
  - Rules for getter methods:
    - It should be public
    - The return-type should not be void
    - The method name should be prefixed with get
    - For boolean attributes, the name can be prefixed with “get" or “is" but preferred prefix would be “is"
    - It should not take any argument

Interoperability with RDDs (3)

- When `JavaBean` classes cannot be defined ahead of time, a `Dataset<Row>` can be created programmatically with three steps.
  - **Step 1**: Create an RDD of Rows from the original RDD
  - **Step 2**: Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1
  - **Step 3**: Apply the schema to the RDD of Rows via `createDataFrame` method provided by SparkSession
- See the method `runProgrammaticSchemaExample` in JavaSQLExample3.java

User Defined Aggregations

- Untyped user-defined aggregate functions. Users have to extend the `UserDefinedAggregateFunction` abstract class to implement a custom untyped aggregate function.
- See example UserDefinedUntypedAggregation.java in `spark-sql-example` folder.
- User-defined aggregations for strongly typed Datasets revolve around the `Aggregator` abstract class.
- See example UserDefinedTypedAggregation.java in `spark-sql-example` folder.

Performance Tuning

- Caching data in memory. Spark SQL can cache tables using an in-memory columnar format by calling `spark.catalog.cacheTable("tableName")` or `dataFrame.cache()`.
- Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure.
- We can call `spark.catalog.uncacheTable("tableName")` to remove the table from memory.
- Configuration of in-memory caching can be done using the `setConf` method on `SparkSession` or by running `SET key=value` commands using SQL.

```
spark.sql.inMemoryColumnarStorage.compressed
    default value: true
spark.sql.inMemoryColumnarStorage.batchSize
    default value: 10000
```

- See Spark SQL documentation for more tuning options.

Hive

- Apache Hive is a SQL layer on top of Hadoop. Hive uses a SQL-like HiveQL query language to execute queries over the large volume of data stored in HDFS.
- HiveQL queries are executed using Hadoop MapReduce, but Hive can also use other distributed computation engines like Apache Spark.
- Since HiveQL is very similar to SQL and many data analysts know SQL already, Hive has always been a viable choice for data queries with Hadoop for storage (HDFS) and processing (MapReduce).

Distributed SQL Engine (1)

- Spark SQL can also act as a distributed query engine using its JDBC/ODBC or command-line interface. In this mode, end-users or applications can interact with Spark SQL directly to run SQL queries, without the need to write any code.
- **Thrift JDBC/ODBC server**: The Thrift JDBC/ODBC server implemented here corresponds to the HiveServer2 in Hive 1.2.1. We can test the JDBC server with the `beeline` script that comes with either Spark or Hive 1.2.1.
- To start the JDBC/ODBC server, run the following script. The script accepts all of the spark-submit command line options plus a `-hiveconf` option to specify Hive properties. By default, the server listens on `localhost:10000`. You may override this behaviour via either environment variables or system properties.

```
start-thriftserver.sh
```

Distributed SQL Engine (2)

- Then start the `beeline` and connect to the JDBC/ODBC server.

```
beeline
beeline> !connect jdbc:hive2://localhost:10000
```

- Then explore the database. Use the `!quit` command to end the session with `beeline`. Beeline will ask you for a username and password. In non-secure mode, simply enter the username on your machine and a blank password. Here is the beeline documentation.
- The Spark SQL CLI (`spqrk-sql`) is a convenient tool to run the Hive metastore service in local mode and execute queries input from the command line. Note that the Spark SQL CLI cannot talk to the Thrift JDBC server. And we cnanot have both running at the same time.

HTTP Mode with Thrift

- Thrift JDBC server also supports sending thrift RPC messages over HTTP transport. Use the following setting to enable HTTP mode as system property or in hive-site.xml file in conf/:

```
hive.server2.transport.mode - Set this to: http
hive.server2.thrift.http.port - HTTP port to listen on;
    default is 10001
hive.server2.http.endpoint - HTTP endpoint; default is
    cliservice
```

- To test, use beeline to connect to the JDBC/ODBC server in http mode with:

```
beeline> !connect jdbc:hive2://<host>:<port>/<database>?
    hive.server2.transport.mode=http;hive.server2.thrift
    .http.path=<http_endpoint>
```