Spark SQL

- Spark’s interface for working with **structured and semistructured data.**
- Structured data is any data **that has a schema — that is, a known set of fields for each record.**
- When you have **this type of data, Spark SQL makes it both easier and more efficient to load and query**

In particular, Spark SQL provides three main capabilities

1. It can **load data from a variety of structured sources** (e.g., JSON, Hive, and Parquet).

2. It lets you **query the data using SQL, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors** (JDBC/ODBC), such as business intelligence tools like Tableau.
3. **When used within a Spark program, Spark SQL provides rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables**, expose custom **functions in SQL, and more**. Many jobs are easier to write using this combination.

- To implement these capabilities, **Spark SQL provides a special type of RDD called SchemaRDD.**

- A **SchemaRDD is an RDD of Row objects, each representing a record**.

- A SchemaRDD **also knows the schema (i.e., data fields) of its rows.**

- While **SchemaRDDs look like regular RDDs, internally they store data in a more efficient manner, taking advantage of their schema.**

- In addition, **they provide new operations not available on RDDs, such as the ability to run SQL querie**s. SchemaRDDs can be created from external data sources, from the results of queries, or from regular RDDs

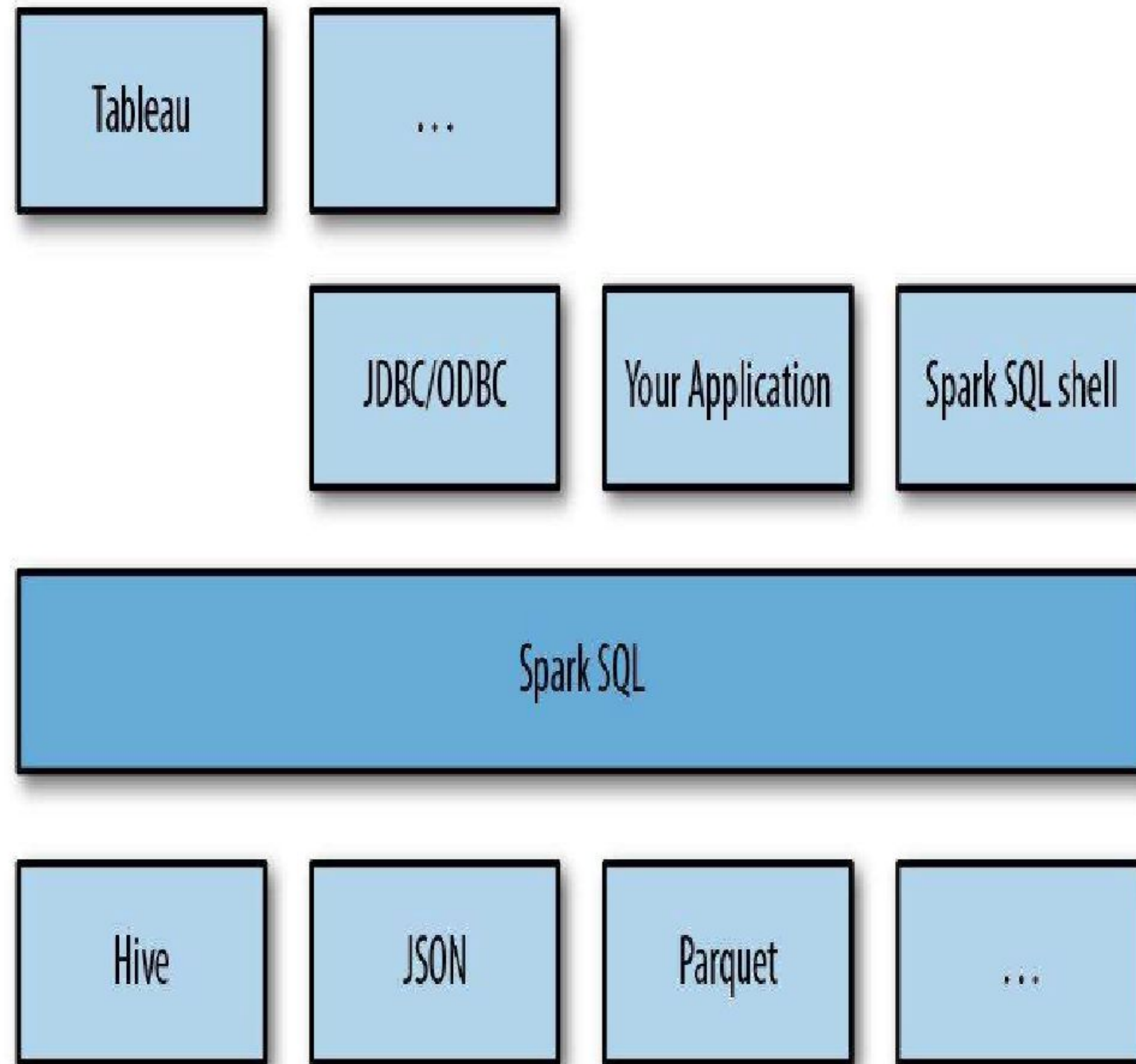


Figure 9-1. Spark SQL usage

Linking with Spark SQL

Spark SQL can be built **with or without Apache Hive, the Hadoop SQL engine.**

Spark SQL **with Hive support allows us to access Hive tables, UDFs** (user-defined functions), SerDes (serialization and deserialization formats), **and the Hive query language (HiveQL).**

It is important to **note that including the Hive libraries does not require an existing Hive installation.**

In general, it is **best to build Spark SQL with Hive support to access these features**

If you have dependency conflicts with Hive, you can also build and link to Spark SQL without Hive

When programming against Spark SQL we have **two entry points depending on whether we need Hive support.**

The recommended entry point is the HiveContext to provide access to HiveQL and other Hive-dependent functionality.

The more **basic SQLContext provides a subset of the Spark SQL support that does not depend on Hive.**

The separation exists for users who might have conflicts with including all of the Hive dependencies.

If you don’t have an existing Hive installation, Spark SQL will still run.

Using Spark SQL in Applications

The **most powerful way to use Spark SQL is inside a Spark application.**

This gives us the **power to easily load data and query it with SQL while simultaneously combining it with "regular" program code in Python, Java, or Scala.**

To use Spark SQL this way, **we construct a HiveContext** (or SQLContext for those wanting a stripped-down version) **based on our SparkContext.**

This context **provides additional functions for querying and interacting with Spark SQL data**.

Using the **HiveContext, we can build SchemaRDDs, which represent our structure data, and operate on them with SQL or with normal RDD operations like map()**

Initializing Spark SQL To get started with Spark SQL we need to add a few imports to our programs, as shown in Example.
Scala SQL imports //

Import Spark SQL import org.apache.spark.sql.hive.HiveContext

// Or if you can’t have the hive dependencies I
import org.apache.spark.sql.SQLContext

Example 9-4. Java SQL imports

// Import Spark SQL import org.apache.spark.sql.hive.HiveContext; /

/ Or if you can't have the hive dependencies import org.apache.spark.sql.SQLContext; //

Example 9-5. Python SQL imports

Import Spark SQL from pyspark.sql import HiveContext, Row

Or if you can't include the hive requirements from pyspark.sql

import SQLContext, Row

Once we’ve added our imports, we need to create a HiveContext, or a SQLContext if we cannot bring in the Hive dependencies

Both of these classes take a SparkContext to run on.

Example 9-6. Constructing a SQL context in Scala

```
val sc = new SparkContext(…) val hiveCtx = new HiveContext(sc)
```

Example 9-7. Constructing a SQL context in Java

```
JavaSparkContext ctx = new JavaSparkContext(…);
SQLContext sqlCtx = new HiveContext(ctx);
```

Example 9-8. Constructing a SQL context in Python

```
hiveCtx = HiveContext(sc)
```

Now that we have a HiveContext or SQLContext, we are ready to load our data and query

Basic Query Example

To make a query against a table, we call the sql() method on the HiveContext or SQLContext.

The first thing we need to do is tell Spark SQL about some data to query.

In this case we will load some Twitter data from JSON, and give it a name by registering it as a “temporary table” so we can query it with SQL

Example 9-9. Loading and quering tweets in Scala

```
val input = hiveCtx.jsonFile(inputFile)
// Register the input schema RDD
input.registerTempTable(“tweets”)
// Select tweets based on the retweetCount
val topTweets = hiveCtx.sql(“SELECT text, retweetCount FROM
tweets ORDER BY retweetCount LIMIT 10”)
```

Example 9-11. Loading and quering tweets in Python

```
input = hiveCtx.jsonFile(inputFile)
# Register the input schema RDD
input.registerTempTable(“tweets”)
# Select tweets based on the retweetCount
topTweets = hiveCtx.sql(“““SELECT text, retweetCount FROM
tweets ORDER BY retweetCount LIMIT 10”””)
```

SchemaRDDs

Both loading data and executing queries return SchemaRDDs.

SchemaRDDs are similar to tables in a traditional database. Under the hood, a SchemaRDD is an RDD composed of Row objects with additional schema information of the types in each column.

Row objects are just wrappers around arrays of basic types (e.g., integers and strings)

SchemaRDDs are also regular RDDs, so you can operate on them using existing RDD transformations like map() and filter().

However, they provide several additional capabilities.

Most importantly, you can register any SchemaRDD as a temporary table to query it via HiveContext.sql or SQLContext.sql.
You do so using the SchemaRDD’s registerTempTable() method

SchemaRDDs can store several basic types, as well as structures and arrays of these types.

Table 9-1. Types stored by SchemaRDDs

Spark SQL/HiveQL type	Scala type	Java type	Python
TINYINT	Byte	Byte/byte	int/long (in range of –128 to 127)
SMALLINT	Short	Short/short	int/long (in range of –32768 to 32767)
INT	Int	Int/int	int or long
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	Java.math.BigDecimal	decimal.Decimal

BINARY	Array[Byte]	byte[]	bytearray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	java.sql.TimeStamp	java.sql.TimeStamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list, tuple, or array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict
STRUCT<COL1: COL1_TYPE, …>	Row	Row	Row

Working with Row objects

Row objects represent **records inside SchemaRDDs, and are simply fixed-length arrays of fields.**

In Scala/Java, **Row objects have a number of getter functions to obtain the value of each field given its index.**

The standard getter, get , takes a column number and returns an Object type that we are responsible for casting to the correct type.

For Boolean, Byte, Double, Float, Int, Long, Short, and String, there is a getType() method, which returns that type.

For example, getString(0) would return field 0 as a string

Accessing the text column (also first column) in the topTweetsSchemaRDD in Scala

```
val topTweetText = topTweets.map(row => row.getString(0))
```

Accessing the text column in the topTweets SchemaRDD in Python

```
topTweetText = topTweets.map(lambda row: row.text)
```

Caching

Caching in Spark SQL works a bit differently.

Since we know the types of each column,Spark is able to more efficiently store the data.

To make sure that we cache using the memory efficient representation, rather than the full objects, we should use the special hiveCtx.cacheTable("tableName") method.

When caching a table Spark SQL represents the data in an in-memory columnar format.

This cached table will remain in memory only for the life of our driver program, so if it exits we will need to recache our data.

Loading and Saving Data

Spark SQL **supports a number of structured data sources out of the box, letting you get Row objects from them** without any complicated loading process.

These sources include **Hive tables, JSON, and Parquet files.**

In addition, **if you query these sources using SQL and select only a subset of the fields, Spark SQL can smartly scan only the subset of the data for those fields, instead of scanning all the data**

Apart from **these data sources, you can also convert regular RDDs in your program to SchemaRDDs** by assigning them a schema.

This makes it easy to write SQL queries even when your underlying data is Python or Java objects.

Hive load in Python

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT key, value FROM mytable")
keys = rows.map(lambda row: row[0])
```

Hive load in Scala

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new HiveContext(sc)
val rows = hiveCtx.sql("SELECT key, value FROM mytable")
val keys = rows.map(row => row.getInt(0))
```

To load our JSON data, all we need to do is call the jsonFile() function on our hiveCtx,
as shown in Examples

Input records

{“**name**”: “Holden”} {“**name**”:“Sparky The Bear”, “**lovesPandas**”:**true**, “**knows**”:{“**friends**”: [“holden”]}}

Loading JSON with Spark SQL in Python

```
input = hiveCtx.jsonFile(inputFile)
```

Loading JSON with Spark SQL in Scala

```
val input = hiveCtx.jsonFile(inputFile)
```

From RDDs

In addition to loading data, we can also create a SchemaRDD from an RDD.

In Scala, RDDs with case classes are implicitly converted into SchemaRDDs.

For Python we create an RDD of Row objects and then call inferSchema()

Example - Creating a SchemaRDD using Row and named tuple in Python

```
happyPeopleRDD = sc.parallelize([Row(name=“holden”,
favouriteBeverage=“coffee”)])
happyPeopleSchemaRDD =
hiveCtx.inferSchema(happyPeopleRDD)
happyPeopleSchemaRDD.registerTempTable(“happy_people”)
```

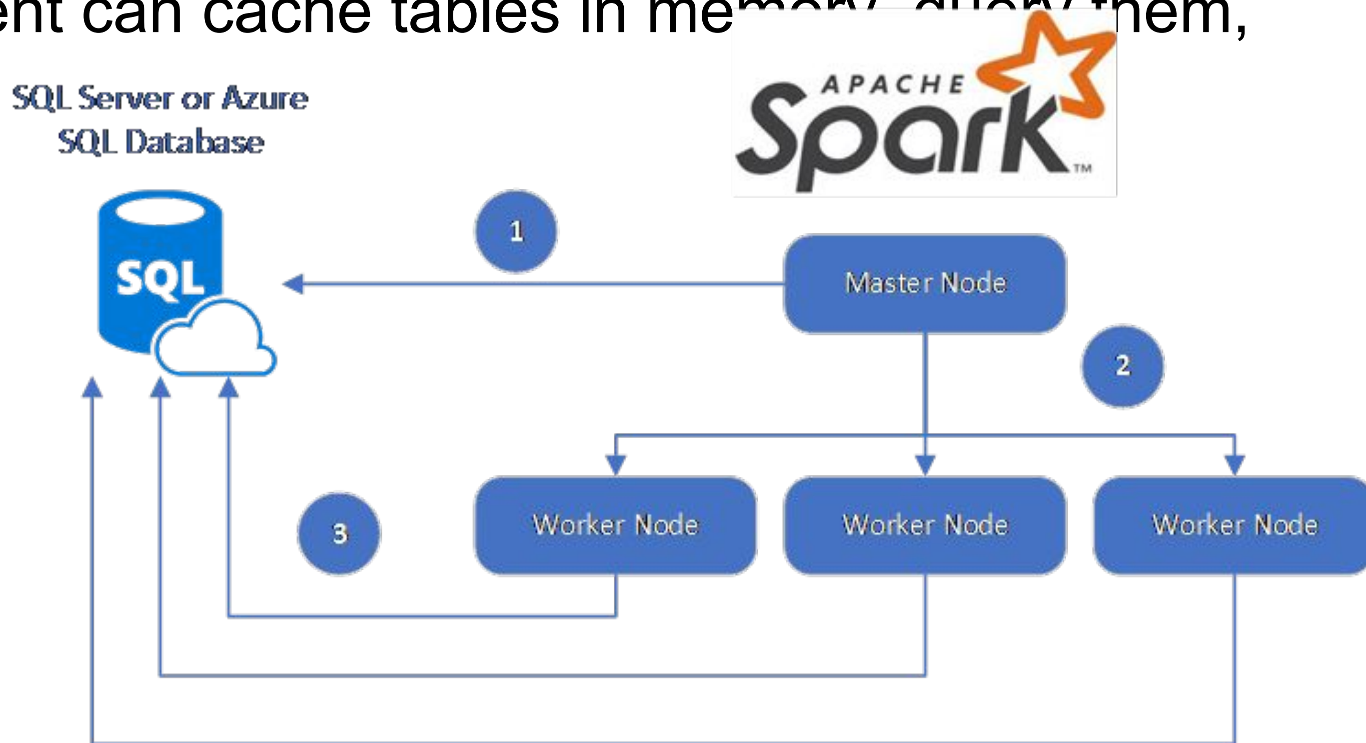
Example. Creating a SchemaRDD from case class in Scala

```
case class HappyPerson(handle: String, favouriteBeverage: String)
… // Create a person and turn it into a Schema RDD
val happyPeopleRDD = sc.parallelize(List(HappyPerson(“holden”, “coffee”)))
// Note: there is an implicit conversion
// that is equivalent to sqlCtx.createSchemaRDD(happyPeopleRDD)
happyPeopleRDD.registerTempTable(“happy_people”)
```

JDBC/ODBC Server

Spark SQL also provides JDBC connectivity, which is useful for connecting business intelligence (BI) tools to a Spark cluster and for sharing a cluster across multiple users.

The JDBC server runs as a standalone Spark driver program that can be shared by multiple
clients. Any client can cache tables in memory, query them,



User-Defined Functions

User-defined functions, or UDFs, allow you to register custom functions in Python, Java, and Scala to call within SQL.

Spark SQL UDFs

Spark SQL offers a built-in method to easily register UDFs by passing in a function in your programming language.

In Scala and Python, we can use the native function and lambda syntax of the language, and in Java we need only extend the appropriate UDF class.

Example - Python string length UDF

```
# Make a UDF to tell us how long some text is
hiveCtx.registerFunction(“strLenPython”, lambda x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql(“SELECT strLenPython(‘text’) FROM tweets
LIMIT 10”)
```

Example - Scala string length UDF

```
registerFunction(“strLenScala”, (_: String).length)
val tweetLength = hiveCtx.sql(“SELECT strLenScala(‘tweet’) FROM tweets LIMIT
10”)
```

Spark SQL Performance

Spark SQL's higher-level query language and additional type information allows Spark SQL to be more efficient.

Spark SQL makes it very easy to perform conditional aggregate operations, like counting the sum of multiple columns

Example 9-40. Spark SQL multiple sums

```
SELECT SUM(user.favouritesCount), SUM(retweetCount), user.id
FROM tweets
GROUP BY user.id
```

Spark SQL is able to use the knowledge of types to more efficiently represent our data.

When caching data, Spark SQL uses an in-memory columnar storage.