6

SQL and NoSQL Programming with Spark

Data is a precious thing and will last longer than the systems themselves.

Tim Berners-Lee, father of the World Wide Web

In This Chapter:

- Introduction to Hive and Spark SQL
- Introduction to the `SparkSession` object and DataFrame API
- Creating and accessing Spark DataFrames
- Using Spark SQL with external applications
- Introduction to NoSQL concepts and systems
- Using Spark with HBase, Cassandra, and DynamoDB

Moore’s law and the birth and explosion of mobile ubiquitous computing have permanently altered the data, computing, and database landscape. This chapter focuses on how Spark can be used in SQL applications using well-known semantics, as well as how Spark can be used in NoSQL applications where a SQL approach is not practical.

Introduction to Spark SQL

Structured Query Language (SQL) is the language most commonly and widely used to define and express questions about data. The vast majority of operational data that exists today is stored in tabular format in relational database systems. Many data analysts innately deconstruct complex problems into a series of SQL Data Manipulation Language (DML) or `SELECT` statements. A discussion of Spark SQL requires a basic understanding of the Hive project, which was born from the Hadoop ecosystem.

Introduction to Hive

Many of the SQL abstractions to Big Data processing platforms, such as Spark, are based on the Hive project. Hive and the Hive metastore remain integral components to projects such as Spark SQL.

The Apache Hive project started at Facebook in 2010 to provide a high-level SQL-like abstraction on top of Hadoop MapReduce. Hive introduced a new language called Hive Query Language (HiveQL), which implements a subset of SQL-92, an internationally accepted standard specification for the SQL language, with some extensions.

The creation of Hive was motivated by the fact that, at the time, few analysts had Java MapReduce programming skills, but most analysts were proficient in SQL. Furthermore, SQL is the common language for BI and visualization and reporting tools, which commonly use ODBC/JDBC as a standard interface.

In Hive's original implementation, HiveQL was parsed by the Hive client and mapped to a sequence of Java MapReduce operations, which were then submitted as jobs on the Hadoop cluster. The progress was monitored, and results were returned to the client or written to the desired location in HDFS. Figure 6.1 provides a high-level depiction of how Hive processes data on HDFS.

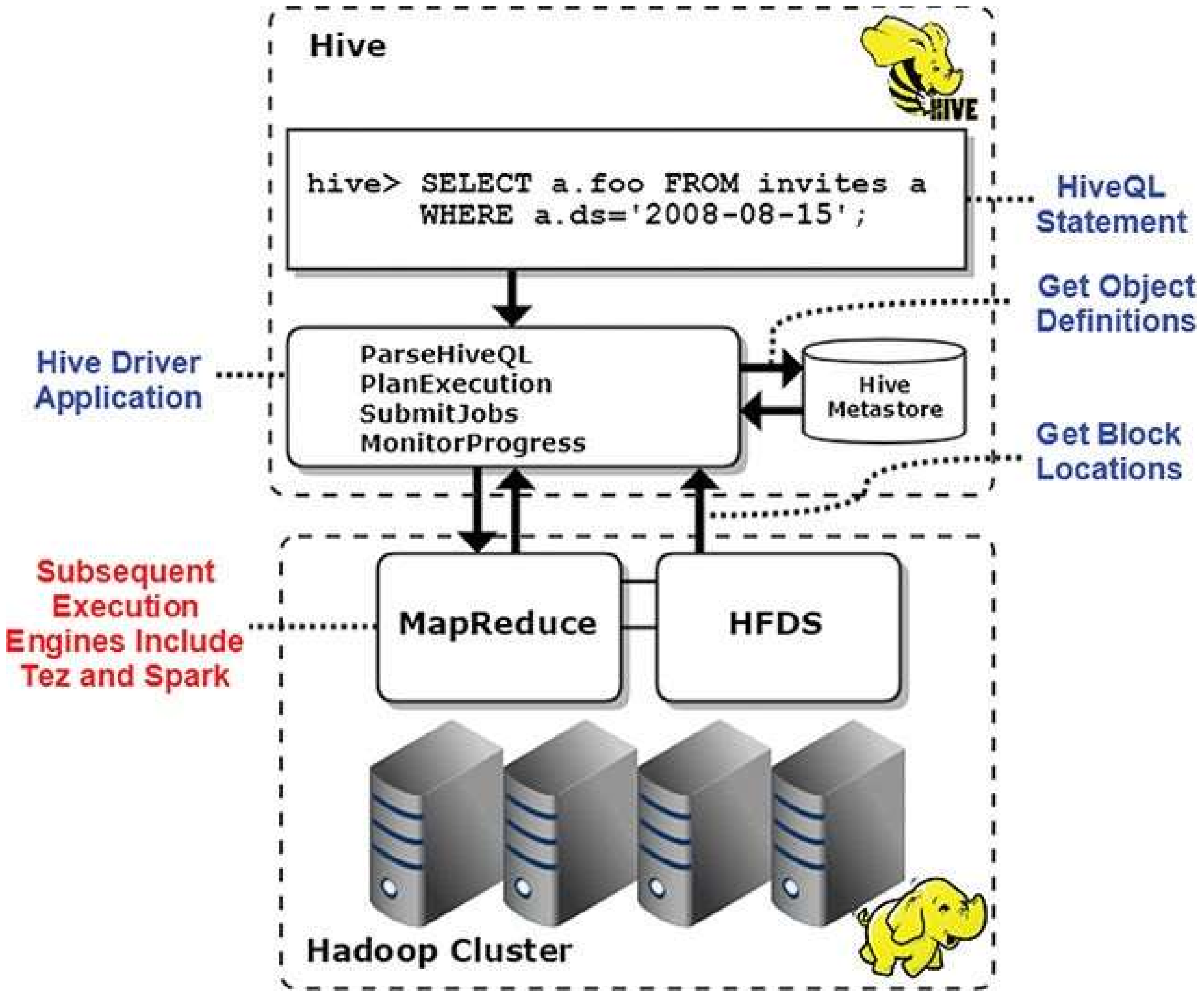


Figure 6.1 Hive high-level overview.

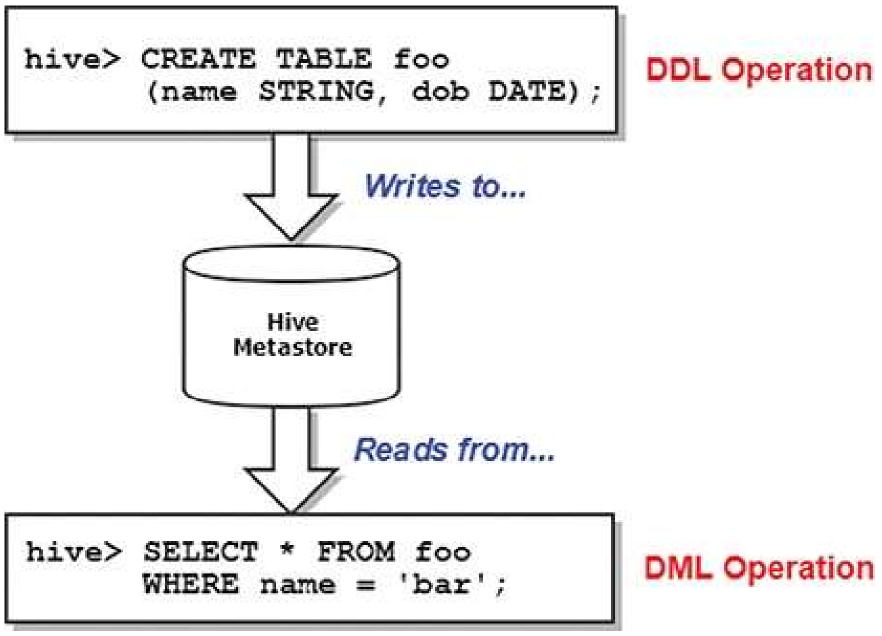
Hive Objects and the Hive Metastore

Hive implements a tabular abstraction to objects in HDFS, presenting directories and all files they contain as tables in its programming model. Just as in a conventional relational database, tables have predefined columns with designated datatypes. The data in HDFS is accessible via SQL DML statements, as with a normal database management system. This is where the similarity ends, however, as Hive is a “schema-on-read” platform, backed by an immutable filesystem, HDFS. As Hive simply implements SQL tabular abstractions over raw files in HDFS, the following key differences exist between Hive and a conventional relational database platform:

- `UPDATE` is not really supported. Although `UPDATE` was introduced into the HiveQL dialect, HDFS is still an immutable filesystem, so this abstraction involves applying coarse-grained transformations, whereas a true `UPDATE` in an RDBMS is a fine-grained operation.

- There are no transactions, journaling, rollbacks, or real transaction isolation levels.
- There is no declarative referential integrity (DRI), which means there are no definitions for primary keys or foreign keys.
- Incorrectly formatted data, such as mistyped data or malformed records, are simply represented to the client as `null` values.

The mapping of tables to their directory locations in HDFS and the columns and their definitions is maintained in the *Hive metastore*. The metastore is a relational database written to and read by the Hive client. The object definitions also include the input and output formats for the files represented by the table objects (`CSVInputFormat` and so on) and SerDes (Serialization/Deserialization), which instruct Hive on how to extract records and fields from the files. Figure 6.2 shows a high-level example of interactions between Hive and the metastore.



```
hive> CREATE TABLE foo
      (name STRING, dob DATE);
```

```
hive> SELECT * FROM foo
      WHERE name = 'bar';
```

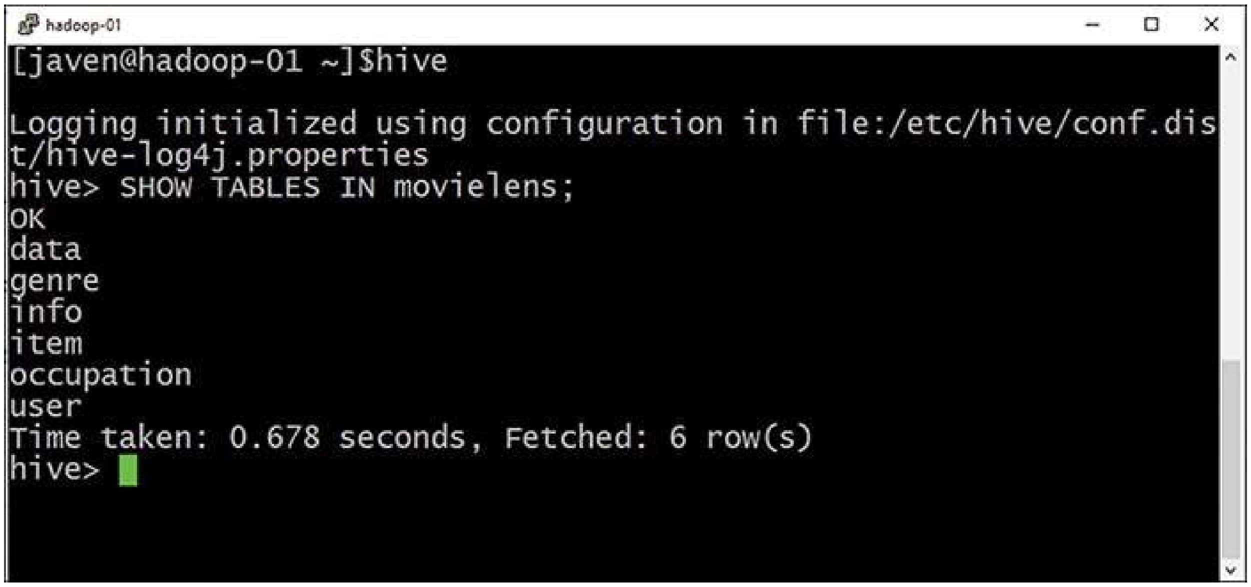
Figure 6.2 Hive metastore interaction.

The metastore can be an embedded Derby database (the default) or a local or remote database, such as MySQL or Postgres. In most cases, you want to implement a shared database, which enables developers and analysts to share object definitions.

There is also a Hive subproject called *HCatalog*, an initiative to extend objects created in Hive to other projects with a common interface, such as Apache Pig. Spark SQL leverages the Hive metastore, as we soon discuss.

Accessing Hive

Hive provides a client command line interface (CLI) that accepts and parses HiveQL input commands. This is a common method for performing ad hoc queries. Figure 6.3 shows the Hive CLI.



```
[javen@hadoop-01 ~]$hive

Logging initialized using configuration in file:/etc/hive/conf.dis
t/hive-log4j.properties
hive> SHOW TABLES IN movielens;
OK
data
genre
info
item
occupation
user
Time taken: 0.678 seconds, Fetched: 6 row(s)
hive> 
```

Figure 6.3 The Hive command line interface.

The Hive CLI is used when the Hive client or driver application deploys to the local machine, including the connection to the metastore. For large-scale implementations, a client/server approach is often more appropriate because the details about the connection to the metastore stay in one place on the server, and access can be controlled to the cluster. This approach uses a server component called *HiveServer2*.

HiveServer2 can now act as a multi-session driver application for multiple clients. HiveServer2 provides a JDBC interface that is usable by external clients, such as visualization tools, as well as a lightweight CLI called Beeline. Beeline is included and usable directly with Spark SQL. In addition, a web-based interface called *Beeswax* is used within the Hadoop User Experience (HUE) project.

Hive Datatypes and Data Definition Language (DDL)

Hive supports most common primitive datatypes, similar to those found in most database systems, as well as several complex datatypes. These types, used as the underlying types for Spark SQL, are listed in Table 6.1.

Table 6.1 **Hive Datatypes**

Datatype	Category	Description
`TINYINT`	Primitive	1-btye signed integer
`SMALLINT`	Primitive	2-byte signed integer
`INT`	Primitive	4-byte signed integer
`BIGINT`	Primitive	8-byte signed integer
`FLOAT`	Primitive	4-byte single precision floating-point number
`DOUBLE`	Primitive	8-byte double precision floating-point number
`BOOLEAN`	Primitive	True/false value
`STRING`	Primitive	Character string
`BINARY`	Primitive	Byte array
`TIMESTAMP`	Primitive	Timestamp with nanosecond precision
`DATE`	Primitive	Year/month/day, in the form YYYYMMDD
`ARRAY`	Complex	Ordered collection of fields of the same type
`MAP`	Complex	Unordered collection of key value pairs
`STRUCT`	Complex	Collection of named fields of varying types

Listing 6.1 provides an example of a typical Hive DDL statement used to create a table in Hive.

Listing 6.1 **Hive `CREATE TABLE` Statement**

Click here to view code image

```
CREATE EXTERNAL TABLE stations (
station_id INT,
name STRING,
lat DOUBLE,
long DOUBLE,
dockcount INT,
landmark STRING,
installation STRING )
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION 'hdfs:///data/bike-share/stations';
```

Internal Tables Versus External Tables in Hive

When you create tables in Hive, the default option is to create a Hive "internal" table. Hive manages directories for internal tables, and a `DROP TABLE` statement for an internal table deletes the corresponding files from HDFS. It is recommended to use external tables by specifying the keyword `EXTERNAL` in the `CREATE TABLE` statement. This provides the schema and location for the object in HDFS, but a `DROP TABLE` operation does not delete the directory and files.

Spark SQL Architecture

Spark SQL provides a mainly HiveQL-compatible SQL abstraction to its RDD-based storage, scheduling, and execution model. Many of the key characteristics of the core Spark project are in Spark SQL, including lazy evaluation and mid-query fault tolerance. Moreover, Spark SQL is usable with the Spark core API within a single application.

Spark SQL includes some key extensions to the core API that are designed to optimize typical relational access patterns. These include the following:

- **Partial DAG execution (PDE):** PDE enables DAGs to be changed and optimized on the fly as information about the data is discovered during processing. The DAG modifications include optimization for performing joins, handling skew in the data, and altering the degree of parallelism that

Spark uses.

- **Partition statistics:** Spark SQL maintains statistics about data within partitions, which can be leveraged in PDE, and provides the capability to do map pruning (pruning or filtering of partitions based on columnar statistics) and optimize normally expensive join operations.
- **The DataFrame API:** We discuss this in detail later in this chapter, in the section "Getting Started with DataFrames."
- **Columnar storage:** Spark SQL stores objects in memory using columnar storage, which organizes data by columns instead of by rows. This has a significant performance impact on SQL access patterns. Figure 6.4 shows the difference between columnar and row-oriented data storage.

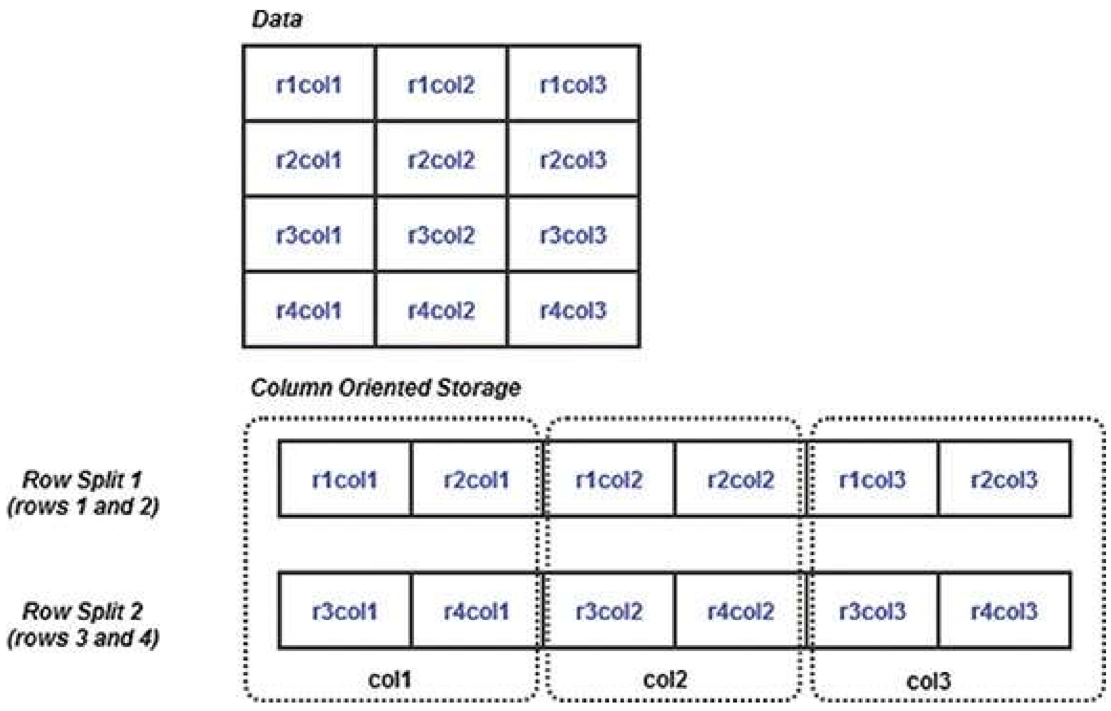


Figure 6.4 Column-oriented storage.

Spark SQL also includes native support for files in Parquet format, which is a columnar file-based storage format optimized for relational access.

Spark SQL is designed for use with environments already using Hive, with a Hive metastore and Hive (or HCatalog) object definitions for data stored in HDFS, S3, or other sources. The SQL dialect that Spark SQL supports is a subset of HiveQL and supports many HiveQL built-in functions and user-defined functions (UDFs). Spark SQL can also be used without Hive or a Hive metastore. Figure 6.5 shows a high-level overview of the Spark SQL architecture, along with the interfaces exposed by Spark SQL.

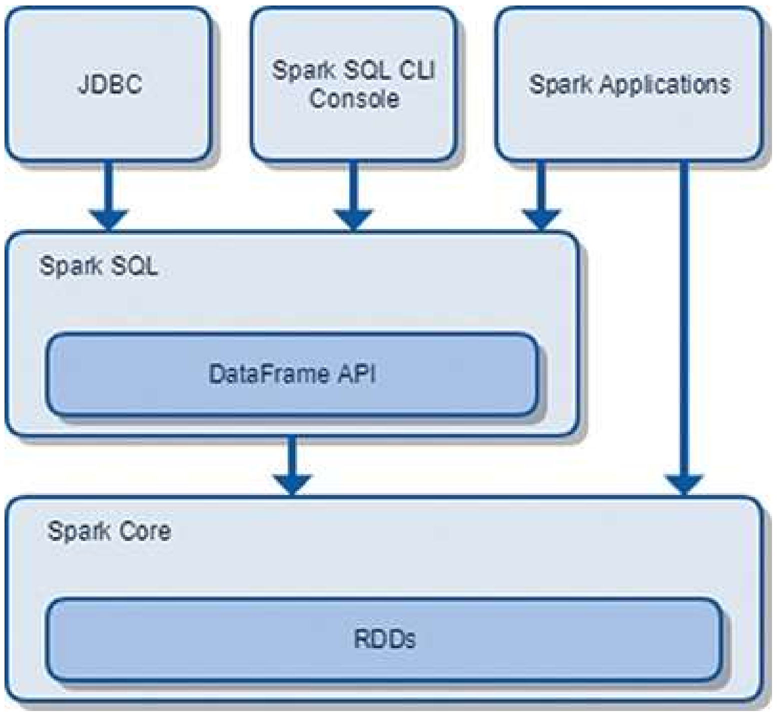


Figure 6.5 Spark SQL high-level architecture.

For more information on the Spark SQL architecture, see the whitepaper “Spark SQL: Relational Data Processing in Spark,” which is available at http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf.

The `SparkSession` Entry Point

Just as the `SparkContext` is the main entry point for an application using the Spark core API, the `SparkSession` object is the main entry point for Spark SQL applications. Prior to release 2 of Spark, special contexts called the `SQLContext` and `HiveContext` were the main entry points for Spark SQL applications. The `SparkSession` object encapsulates these contexts into one succinct entry point.

A `SparkSession` entry point, instantiated as `spark` in the interactive shells, contains a reference to a metastore to hold object (table) definitions. If Hive is available and configured, its metastore is used; otherwise, it uses its own local metastore.

Configure Hive for use with Spark can be achieved by placing your `hive-site.xml`, `core-site.xml` (for security configuration), and `hdfs-site.xml` (for HDFS configuration) files in the `conf/` directory of your `$SPARK_HOME`. The primary Hive configuration file, `hive-site.xml`, includes location and connection details for the Hive metastore.

Listing 6.2 shows instantiation of a `SparkSession` for a batch application. Note that this instantiation is not necessary when using the shells `pyspark` and `spark-shell`.

> **`SparkSession` and `SQLContext`**
>
> You may still find references to the `SQLContext` object in various examples, typically instantiated as `sqlContext`. This object is available in the interactive Spark shells and can be used interchangeably with the `spark` object. It is generally preferable to use an instantiation of the `SparkSession` object, as the `SQLContext` may be deprecated in a future release.

Listing 6.2 **Creating a `SparkSession` Object with Hive Support**

Click here to view code image

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("My Spark SQL Application") \
    .enableHiveSupport() \
    .getOrCreate()
...
```

The `SparkSession` object exposes the DataFrame API, which we discuss in the next section, and enables you to create and query table objects using SQL statements and operators. If you have Hive available and configured as shown in Listing 6.2, you can query objects referenced in the Hive metastore using HiveQL statements as shown in Listing 6.3.

Listing 6.3 **Hive Queries Using Spark SQL**

Click here to view code image

```
# SparkSession available as 'spark'
sql_cmd = """SELECT name, lat, long FROM stations WHERE landmark = 'San
Jose'"""
spark.sql(sql_cmd).show()
# returns:
# +--------------------+---------+-----------+
# |                name|      lat|       long|
# +--------------------+---------+-----------+
# |San Jose Diridon ...|37.329732|-121.901782|
# |San Jose Civic Ce...|37.330698|-121.888979|
# |Santa Clara at Al...|37.333988|-121.894902|
# |    Adobe on Almaden|37.331415|  -121.8932|
# |    San Pedro Square|37.336721|-121.894074|
# +--------------------+---------+-----------+
# only showing top 5 rows
```

Getting Started with DataFrames

Spark SQL *DataFrames* are distributed collections of records, all with the same defined schema, conceptually analogous to a sharded table from a relational database. Spark SQL DataFrames were first introduced as SchemaRDD objects; they are loosely based on the DataFrame object constructs in R, discussed in Chapter 8, "Introduction to Data Science and Machine Learning Using Spark," and Pandas (the Python library for data manipulation and analysis).

DataFrames are an abstraction for Spark RDDs. However, unlike primitive RDDs, DataFrames track their schema and provide native support for many common SQL functions and relational operators. DataFrames, like RDDs, are evaluated as DAGs, using lazy evaluation and providing lineage and fault tolerance. Also like RDDs, DataFrames support caching and persistence using methods similar to those discussed in the previous chapter.

DataFrames can be created in many different ways, including from the following:

- An existing RDD
- A JSON file
- A text file, a Parquet file, or an ORC file

- A table in Hive
- A table in an external database
- A temporary table in Spark

The following sections look at some of the common methods of constructing DataFrames if there is an existing `SparkSession` object.

Creating a DataFrame from an Existing RDD

The main function used to create DataFrames from RDDs is the `createDataFrame()` method, described next.

`createDataFrame()`

Syntax:

Click here to view code image

```
SparkSession.createDataFrame(data, schema=None, samplingRatio=None)
```

The `createDataFrame()` method creates a DataFrame object from an existing RDD. The `data` argument is a reference to a named RDD object consisting of tuples or list elements. The `schema` argument refers to the schema to be projected to the DataFrame object. The `samplingRatio` argument is for sampling the data if the schema is inferred. (You'll learn more about defining or inferring schemas for DataFrame objects shortly.) Listing 6.4 shows an example of loading a DataFrame from an existing RDD.

Listing 6.4 **Creating a DataFrame from an RDD**

Click here to view code image

```
myrdd = sc.parallelize([('Jeff', 48),('Kellie', 45)])
spark.createDataFrame(myrdd).collect()
# returns:
# [Row(_1=u'Jeff', _2=48), Row(_1=u'Kellie', _2=45)]
```

Notice that the return value from the `collect` action is a list of `Row` (`pyspark.sql.Row`) objects. In this case, because the schema, including the field names, is unspecified, the fields are referenced by `_<fieldnumber>`, where the field number starts at one.

Creating a DataFrame from a Hive Table

To load data from a Hive table into a Spark SQL DataFrame, you need to create a `HiveContext`. Recall that the `HiveContext` reads the Hive client configuration (`hive-site.xml`) to obtain connection details for the Hive metastore. This enables seamless access to Hive tables from a Spark application. You can do this in a couple different ways, including using the `sql()` method or the `table()` method, as described in the following sections.

`sql()`

Syntax:

```
SparkSession.sql(sqlQuery)
```

The `sql()` method creates a DataFrame object from a table in Hive by supplying a `sqlQuery` argument and performing a DML operation from a table in Hive. If the table is in a database other than the Hive default database, it needs to be referenced using the `<databasename>.<tablename>` format. The `sqlQuery` can be any valid HiveQL statement, including `SELECT *` or a `SELECT` statement with a `WHERE` clause or a `JOIN` predicate. Listing 6.5 shows an example of creating a DataFrame using a HiveQL query against a table in the Hive default database.

Listing 6.5 **Creating a DataFrame from a Table in Hive**

Click here to view code image

```
sql_cmd = """SELECT name, lat, long
             FROM stations
             WHERE landmark = 'San Jose'"""
df = spark.sql(sql_cmd)
df.count()
# returns: 16
df.show(5)
# returns:
# +--------------------+---------+-----------+
# |                name|      lat|       long|
# +--------------------+---------+-----------+
# |San Jose Diridon ...|37.329732|-121.901782|
# |San Jose Civic Ce...|37.330698|-121.888979|
# |Santa Clara at Al...|37.333988|-121.894902|
# |    Adobe on Almaden|37.331415|  -121.8932|
# |    San Pedro Square|37.336721|-121.894074|
# +--------------------+---------+-----------+
# only showing top 5 rows
```

`table()`

Syntax:

```
SparkSession.table(tableName)
```

The `table()` method creates a DataFrame object from a table in Hive. Unlike with the `sql()` method, there is no opportunity to prune columns with a column list or filter rows with a `WHERE` clause. The entire table loads into the DataFrame. Listing 6.6 demonstrates the `table()` method.

Listing 6.6 `table()` Method for Creating a DataFrame from a Table in Hive

Click here to view code image

```
df = spark.table('stations')
df.columns
# returns:
# ['station_id', 'name', 'lat', 'long', 'dockcount', 'landmark',
'installation']
df.count()
# returns: 70
```

There are other useful methods for interrogating the Hive system and database catalogs, such as the `tables()` method, which returns a DataFrame containing names of tables in a given database, and the `tableNames()` method, which returns a list of names of tables in a given Hive database.

Creating DataFrames from JSON Objects

JSON is a common, standard, human-readable serialization or wire transfer format often used in web service responses. Because JSON is a semi-structured source with a schema, support for JSON is included in Spark SQL.

`read.json()`

Syntax:

Click here to view code image

```
DataFrameReader.read.json(path,
                          schema=None,
                          primitivesAsString=None,
                          prefersDecimal=None,
                          allowComments=None,
                          allowUnquotedFieldNames=None,
                          allowSingleQuotes=None,
                          allowNumericLeadingZero=None,
                          allowBackslashEscapingAnyCharacter=None,
                          mode=None,
                          columnNameOfCorruptRecord=None,
                          dateFormat=None,
                          timestampFormat=None,
                          multiLine=None)
```

The `json()` method of the `DataFrameReader` creates a DataFrame object from a JSON file. Listing 6.7 demonstrates the `read.json()` method; notice that the `DataFrameReader` is accessible from the `SparkSession` object. The `path` argument refers to the fully qualified path (in a local or remote filesystem such as HDFS) of the JSON file. The `schema` argument can explicitly define a target schema for the resultant DataFrame, which we look at later in this chapter. Many additional arguments are used to specify formatting options, and the full description of them is in the Spark SQL Python API documentation at https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Data

Listing 6.7 `read.json()` Method for Creating a DataFrame from a JSON File

Click here to view code image

```
people_json_file = '/opt/spark/examples/src/main/resources/people.json'
people_df = spark.read.json(people_json_file)
```

```
people_df.show()
# returns:
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Note that each line in a JSON file must be a valid JSON object. The schemas, or keys, do not need to be uniform across all JSON objects in a file. Keys that are not present in a given JSON object are represented as `null` in the resultant DataFrame.

In addition, the `read.json()` method allows you to create a DataFrame from an existing RDD consisting of a list of one or more discrete JSON objects as strings (see Listing 6.8).

Listing 6.8 **Creating a DataFrame from a JSON RDD**

Click here to view code image

```
rdd= sc.parallelize( \
 ['{"name":"Adobe on Almaden", "lat":37.331415, "long":-121.8932}', \
  '{"name":"Japantown", "lat":37.348742, "long":-121.894715}'])
json_df = spark.read.json(rdd)
json_df.show()
# returns:
# +---------+-----------+----------------+
# |      lat|       long|            name|
# +---------+-----------+----------------+
# |37.331415|  -121.8932|Adobe on Almaden|
# |37.348742|-121.894715|       Japantown|
# +---------+-----------+----------------+
```

Creating DataFrames from Flat Files

The `DataFrameReader` can also be used to load DataFrames from other types of files, such as CSV files, as well as external SQL and NoSQL data sources. The following sections look at some examples of creating DataFrames from plaintext files and columnar storage files, including Parquet and ORC.

`text()`

Syntax:

```
DataFrameReader.read.text(path)
```

The `text()` method of the `DataFrameReader` is used to load DataFrames from text files in an external filesystem (local, NFS, HDFS, S3, or others). Its behavior is similar to its RDD equivalent, `sc.textFile()`. The `path` argument refers to a path that could be a file, directory, or file glob. ("Globbing" expressions are similar to regular expressions used to return a list of files satisfying the glob pattern.)

Listing 6.9 demonstrates the `read.text()` function.

Listing 6.9 **Creating a DataFrame from a Plaintext File or Files**

Click here to view code image

```
# read an individual file
df = spark.read.text('file:///opt/spark/data/bike-
share/stations/stations.csv')
df.take(1)
# returns:
# [Row(value=u'9,Japantown,37.348742,-121.894715,15,San Jose,8/5/2013')]
# you can also read all files from a directory...
df = spark.read.text('file:///opt/spark/data/bike-share/stations/')
df.count()
# returns: 83
```

Note that the `Row` object returned for each line in the text file or files contains one string, which is the entire line of the file.

Columnar Storage and Parquet Files

Columnar storage concepts, introduced earlier in this chapter, extend beyond in-memory structures to persistent file formats such as Parquet and

ORC (Optimized Row Columnar) files. *Apache Parquet* is a popular, generalized columnar storage format designed for integration with any Hadoop ecosystem project. Parquet is a “first-class citizen” in the Spark project and is the preferred storage format for Spark SQL processing. ORC is a successor to RCFile, a columnar storage format built to improve Hive read performance. If you need to share data structures with Hive and accommodate non-Spark access patterns, such as Tez, ORC may be an appropriate format. Parquet support is available through the Hive project as well.

`parquet()`

Syntax:

```
DataFrameReader.read.parquet(paths)
```

The `parquet()` method of the `DataFrameReader` is for loading files stored with the Parquet columnar storage format. These files are often the output of another process, such as output from a previous Spark process. The `paths` argument refers to a Parquet file or files, or a directory containing Parquet files.

The Parquet format encapsulates the schema and data in one structure, so the schema is applied and available to the resultant DataFrame.

Given an existing file in Parquet format, Listing 6.10 demonstrates the use of the `DataFrameReader.read.parquet()` method.

Listing 6.10 **Creating a DataFrame from a Parquet File or Files**

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
df.printSchema()
# returns:
# root
#  |-- station_id: integer (nullable = true)
#  |-- name: string (nullable = true)
#  |-- lat: double (nullable = true)
#  |-- long: double (nullable = true)
#  |-- dockcount: integer (nullable = true)
#  |-- landmark: string (nullable = true)
#  |-- installation: string (nullable = true)
```

```
df.take(1)
# returns:
# [Row(station_id=2, name=u'San Jose Diridon Caltrain Station',
lat=37.329732...)]
```

Parquet and Compression

By default, Spark uses the Gzip codec to compress Parquet files. If you require an alternative codec (such as Snappy) to read or write compressed Parquet files, supply the following config:

Click here to view code image

```
sqlContext.setConf("spark.sql.parquet.compression.codec.",
"snappy")
```

`orc()`

Syntax:

```
DataFrameReader.read.orc(path)
```

The `orc()` method of the `DataFrameReader` is used to load a DataFrame from a file or directory consisting of ORC format files. ORC is a format native to the Hive project. The `path` argument refers to a directory containing ORC files, typically associated with a table in ORC format in a Hive warehouse. Listing 6.11 shows the use of the `orc()` method to load the ORC files associated with a Hive table stored as ORC.

Listing 6.11 **Creating a DataFrame from Hive ORC Files**

Click here to view code image

```
df = spark.read.orc('hdfs:///user/hadoop/stations_orc/')
df.printSchema()
# returns:
# root
# |-- station_id: integer (nullable = true)
 # |-- name: string (nullable = true)
# |-- lat: double (nullable = true)
# |-- long: double (nullable = true)
# |-- dockcount: integer (nullable = true)
# |-- landmark: string (nullable = true)
# |-- installation: string (nullable = true)
df.take(1)
# returns:
# [Row(station_id=2, name=u'San Jose Diridon Caltrain Station',
lat=37.329732 ...)]
```

You can also use the `DataFrameReader` and the `spark.read.jdbc()` method to load data from external data sources such as MySQL, Oracle, or others.

Converting DataFrames to RDDs

You can easily convert DataFrames to native RDDs by using the `rdd()` method, as shown in Listing 6.12. The resultant RDD consists of `pyspark.sql.Row` objects.

Listing 6.12 **Converting a DataFrame to an RDD**

Click here to view code image

```
stationsdf = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
stationsrdd = stationsdf.rdd
stationsrdd
# returns:
# MapPartitionsRDD[4] at javaToPython at ...
stationsrdd.take(1)
# returns:
# [Row(station_id=2, name=u'San Jose Diridon Caltrain Station',
lat=37.329732 ...)]
```

DataFrame Data Model: Primitive Types

The data model for the DataFrame API is based on the Hive data model. Datatypes used with DataFrames map directly to their equivalents in Hive. This includes all common primitive types as well as complex, nested types such as the equivalents to lists, dictionaries, and tuples.

Table 6.2 lists the primitive types encapsulated by PySpark types derived from the base class `pyspark.sql.types.DataType`.

Table 6.2 **Spark SQL Primitive Types (`pyspark.sql.types`)**

Type	Hive Equivalent	Python Equivalent
`ByteType`	`TINYINT`	`int`
`ShortType`	`SMALLINT`	`int`
`IntegerType`	`INT`	`int`
`LongType`	`BIGINT`	`long`
`FloatType`	`FLOAT`	`float`
`DoubleType`	`DOUBLE`	`float`
`BooleanType`	`BOOLEAN`	`bool`
`StringType`	`STRING`	`string`
`BinaryType`	`BINARY`	`bytearray`
`TimestampType`	`TIMESTAMP`	`datetime.datetime`
`DateType`	`DATE`	`datetime.date`

DataFrame Data Model: Complex Types

Complex, nested structures are accessible in Spark SQL using native HiveQL-based operators. Table 6.3 lists the complex types in the DataFrame API, along with their Hive and Python equivalents.

Table 6.3 **Spark SQL Complex Types (`pyspark.sql.types`)**

Type	Hive Equivalent	Python Equivalent
`ArrayType`	`ARRAY`	`list`, `tuple`, or `array`
`MapType`	`MAP`	`dict`
`StructType`	`STRUCT`	`list` or `tuple`

Inferring DataFrame Schemas

The schema for a Spark SQL DataFrame can be explicitly defined or inferred. In previous examples, the schema was not explicit, so in each case, it was *inferred*. Inferring the schema is the simplest method. However, it is generally better practice to define the schema in your code.

Spark SQL uses *reflection*, a process of examining an object to determine its composition, to infer the schema of a DataFrame object. Reflection can interpret a schema for an RDD converted to a DataFrame. In this case, the process involves creating a `Row` object from each record in the RDD and assigning a datatype from each field in the RDD. The datatypes are inferred from the first record, so it is important for the first record to be representative of the dataset and to have no missing values.

Listing 6.13 shows an example of schema inference for a DataFrame created from an RDD. Note the use of the `printSchema()` DataFrame method to print the schema to the console in a tree format.

Listing 6.13 **Schema Inference for a DataFrame Created from an RDD**

Click here to view code image

```
rdd = sc.textFile('file:///home/hadoop/stations.csv') \
        .map(lambda x: x.split(',')) \
        .map(lambda x: (int(x[0]), str(x[1]),
                        float(x[2]), float(x[3]),
                        int(x[4]), str(x[5]), str(x[6])))
rdd.take(1) # returns:
# [(2, 'San Jose Diridon Caltrain Station', 37.329732, −121.901782, 27,
'San Jose',
# '8/6/2013')]
df = spark.createDataFrame(rdd)
df.printSchema()
# returns:
# root
#  |-- _1: long (nullable = true)
#  |-- _2: string (nullable = true)
#  |-- _3: double (nullable = true)
#  |-- _4: double (nullable = true)
#  |-- _5: long (nullable = true)
#  |-- _6: string (nullable = true)
#  |-- _7: string (nullable = true)
```

Note that the fields use the `_<fieldnumber>` convention for their identifiers and have a `nullable` property value set to `True`, meaning these values are not required. Also notice that the larger type variants are assumed. For instance, the `lat` and `long` fields in this RDD are cast as `float` values, yet the inferred schema in the resultant DataFrame uses `double` (actually, an instance of the `DoubleType`) for the same fields. Likewise, `long` values are inferred from `int` values.

Schema inference is performed automatically for DataFrames created from JSON documents, as shown in Listing 6.14.

Listing 6.14 **Schema Inference for DataFrames Created from JSON Objects**

Click here to view code image

```
rdd = sc.parallelize( \
      ['{"name":"Adobe on Almaden", "lat":37.331415, "long":-121.8932}',
\
       '{"name":"Japantown", "lat":37.348742, "long":-121.894715}'])
df = spark.read.json(rdd)
df.printSchema()
# returns:
# root
#  |-- lat: double (nullable = true)
#  |-- long: double (nullable = true)
#  |-- name: string (nullable = true)
```

The schema for a DataFrame created from a Hive table is automatically inherited from its Hive definition, as shown in Listing 6.15.

Listing 6.15 **Schema for a DataFrame Created from a Hive Table**

Click here to view code image

```
df = spark.table("stations")
df.printSchema()
# returns:
```

```
# root
 # |-- station_id: integer (nullable = true)
# |-- name: string (nullable = true)
# |-- lat: double (nullable = true)
# |-- long: double (nullable = true)
# |-- dockcount: integer (nullable = true)
# |-- landmark: string (nullable = true)
# |-- installation: string (nullable = true)
```

Defining DataFrame Schemas

The preferred method of defining a schema for DataFrame objects is to explicitly supply it in your code. To create a schema, you need to create a `StructType` object containing a collection of `StructField` objects. You then apply this schema to your DataFrame when it is created. Listing 6.16 shows an example of explicitly defining a schema using a previous example. Notice the difference in behavior between the inferred and defined schemas.

Listing 6.16 **Defining the Schema for a DataFrame Explicitly**

Click here to view code image

```
from pyspark.sql.types import *
myschema = StructType([ \
           StructField("station_id", IntegerType(), True), \
           StructField("name", StringType(), True), \
           StructField("lat", FloatType(), True), \
           StructField("long", FloatType(), True), \
           StructField("dockcount", IntegerType(), True), \
           StructField("landmark", StringType(), True), \
           StructField("installation", StringType(), True) \
           ])
rdd = sc.textFile('file:///home/hadoop/stations.csv') \
        .map(lambda x: x.split(',')) \
        .map(lambda x: (int(x[0]), str(x[1]),
                        float(x[2]), float(x[3]),
                        int(x[4]), str(x[5]), str(x[6])))
df = spark.createDataFrame(rdd, myschema)
df.printSchema()
# returns:
```

```
# root
#  |-- station_id: integer (nullable = true)
#  |-- name: string (nullable = true)
#  |-- lat: float (nullable = true)
#  |-- long: float (nullable = true)
#  |-- dockcount: integer (nullable = true)
#  |-- landmark: string (nullable = true)
#  |-- installation: string (nullable = true)
```

Using DataFrames

The DataFrame API is currently one of the fastest-moving areas in the Spark project. New and significant features and functions appear with every minor release. Extensions to the Spark SQL DataFrame model, such as the Datasets API, are moving equally quickly. In fact, Spark SQL, including its core component, the DataFrame API, could warrant its own book. The following sections cover the basics of the DataFrame API using Python, providing enough information to get you up and running with DataFrames. The rest is up to you!

DataFrame Metadata Operations

Several metadata functions are available with the DataFrame API. These are functions that return information about the data structure, not the data itself. You have already seen one of the available functions, `printSchema()`, which returns the schema defined for a DataFrame object in a tree format. The following sections explore some of the additional metadata functions.

`columns()`

Syntax:

```
DataFrame.columns()
```

The `columns()` method returns a list of column names for the given DataFrame. An example is provided in Listing 6.17.

Listing 6.17 **Returning a List of Columns from a DataFrame**

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
df.columns
# returns:
# ['station_id', 'name', 'lat', 'long', 'dockcount', 'landmark',
'installation']
```

`dtypes()`

Syntax:

```
DataFrame.dtypes()
```

The `dtypes()` method returns a list of tuples, with each tuple consisting of the column names and the datatypes for a column for a given DataFrame object. This may be more useful than the previously discussed `printSchema()` method because you can access it programmatically. Listing 6.18 demonstrates the `dtypes()` method.

Listing 6.18 **Returning Column Names and Datatypes from a DataFrame**

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
df.dtypes
# returns:
# [('station_id', 'int'), ('name', 'string'), ('lat', 'double'),
('long', 'double'),
# ('dockcount', 'int'), ('landmark', 'string'), ('installation',
'string')]
```

Basic DataFrame Operations

Because DataFrames are columnar abstractions of RDDs, you see many similar functions, such as transformations and actions, that are direct descendants of RDD methods, with some additional relational methods such as `select()`, `drop()`, and `where()`. Core functions such as `count()`, `collect()`, `take()`, and `foreach()` are functionally and syntactically analogous to the functions with the same names in the RDD API. As with the RDD API, each of these methods, as an action, triggers evaluation of the DataFrame and its lineage.

Much as with the `collect()` and `take()` actions, you may have noticed an alternative method, `show()`, used in previous examples. `show()` is an action that triggers evaluation of a DataFrame if the DataFrame does not exist in cache.

The `select()`, `drop()`, `filter()`, `where()`, and `distinct()` methods can prune columns or filter rows from a DataFrame. In each case, the results of these operations create a new DataFrame object.

`show()`

Syntax:

```
DataFrame.show(n=20, truncate=True)
```

The `show()` method prints the first `n` rows of a DataFrame to the console. Unlike `collect()` or `take(n)`, `show()` cannot return to a variable. It is solely intended for viewing the contents or a subset of the contents in the console or notebook. The `truncate` argument specifies whether to truncate long strings and align cells to the right.

The output of the `show()` command is “pretty printed,” meaning it is formatted as a grid result set, including column headings for readability.

`select()`

Syntax:

```
DataFrame.select(*cols)
```

The `select()` method returns a new DataFrame object from the list of columns specified by the `cols` argument. You can use an asterisk (`*`) to select all columns from the DataFrame with no manipulation. Listing 6.19 shows an example of the `select()` function.

Listing 6.19 `select()` Method in Spark SQL

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
newdf = df.select((df.name).alias("Station Name"))
newdf.show(2)
# returns:
# +--------------------+
# |        Station Name|
# +--------------------+
# |San Jose Diridon ...|
# |San Jose Civic Ce...|
# +--------------------+
# only showing top 2 rows
```

As you can see from Listing 6.19, you can also apply column aliases with `select()` by using the `alias` operator; `select()` is also the primary method for applying column-level functions in DataFrame transformation operations. You will see an example of this shortly.

`drop()`

Syntax:

```
DataFrame.drop(col)
```

The `drop()` method returns a new DataFrame with the column specified by the `col` argument removed. Listing 6.20 demonstrates the use of the `drop()` method.

Listing 6.20 **Dropping a Column from a DataFrame**

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
df.columns
# returns:
# ['station_id', 'name', 'lat', 'long', 'dockcount', 'landmark',
'installation']
newdf = df.drop(df.installation)
newdf.columns
# returns:
# ['station_id', 'name', 'lat', 'long', 'dockcount', 'landmark']
```

`filter()`

Syntax:

```
DataFrame.filter(condition)
```

The `filter()` method returns a new DataFrame that contains only rows that satisfy the given condition, an expression provided by the `condition` argument that evaluates to `True` or `False`. Listing 6.21 demonstrates the use of `filter()`.

Listing 6.21 **Filtering Rows from a DataFrame**

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
df.filter(df.name == 'St James Park') \
  .select(df.name,df.lat,df.long) \
  .show()
# returns:
# +-------------+---------+-----------+
# |         name|      lat|       long|
# +-------------+---------+-----------+
# |St James Park|37.339301|-121.889937|
# +-------------+---------+-----------+
```

The `where()` method is an alias for `filter()`, and the two can be used interchangeably.

`distinct()`

Syntax:

```
DataFrame.distinct()
```

The `distinct()` method returns a new DataFrame that contains the distinct rows in the input DataFrame, essentially filtering out duplicate rows. A duplicate row is a row where all values for all columns are the same as for another row in the same DataFrame. Listing 6.22 shows an example of the `distinct()` method.

Listing 6.22 **Filtering Duplicate Rows from a DataFrame**

Click here to view code image

```
rdd = sc.parallelize([('Jeff', 48),('Kellie', 45),('Jeff', 48)])
df = spark.createDataFrame(rdd)
```

```
df.show()
# returns:
# +------+---+
# |    _1| _2|
# +------+---+
# |  Jeff| 48|
# |Kellie| 45|
# |  Jeff| 48|
# +------+---+
df.distinct().show()
# returns:
# +------+---+
# |    _1| _2|
# +------+---+
# |Kellie| 45|
# |  Jeff| 48|
# +------+---+
```

Note that `drop_duplicates()` is a similar method that also lets you optionally consider certain columns to filter for duplicates.

In addition, `map()` and `flatMap()` are available, using `DataFrame.rdd.map()` and `DataFrame.rdd.flatMap()`, respectively. Prior to Spark's 2.0 release, you could run the `map()` and `flatMap()` methods directly on a DataFrame object; however, these were simply aliases for the `rdd.map()` and `rdd.flatMap()` methods.

Along with the `select()` method, you can use the `rdd.map()` and `rdd.flatMap()` methods to apply column-level functions to rows in Spark SQL DataFrames, as well as project-specific columns, including computed columns. However, `select()` operates on a DataFrame and returns a new DataFrame, whereas the `rdd.map()` and `rdd.flatMap()` methods operate on a DataFrame and return an RDD.

Conceptually, these methods function like their named equivalents in the RDD API. However, when dealing with DataFrames with named columns, the `lambda` functions are slightly different. Listing 6.23 uses the `rdd.map()` method to project a column from a DataFrame into a new RDD named `rdd`.

Listing 6.23 **`map()` Functions with Spark SQL DataFrames**

Click here to view code image

```
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
rdd = df.rdd.map(lambda r: r.name)
rdd
# returns:
# PythonRDD[62] at RDD at PythonRDD.scala:48
rdd.take(1)
# returns:
# [u'San Jose Diridon Caltrain Station']
```

If you want the result of a mapping operation to return a new DataFrame instead of an RDD, `select()` is a better option.

Some other operations in the Spark SQL DataFrame API are worth mentioning. The methods `sample()` and `sampleBy()`work similarly to their RDD equivalents, and the `limit()` function creates a new DataFrame with a specific number of arbitrary rows from the originating DataFrame. All these methods are helpful for working with data at scale, limiting the working set during development.

Another useful method during development is `explain()`. The `explain()` method returns a query plan, including a logical and physical plan for evaluating the DataFrame. This can be helpful in troubleshooting or optimizing Spark SQL programs.

You are encouraged to explore the documentation to learn more about all the functions available in the DataFrame API. Notably, Python *docstrings* are included with all functions in the Python Spark SQL API. You can use them to explore the syntax and usage of any function in Spark SQL, as well as any other functions in the Spark Python API. Python docstrings are accessible using the `__doc__` method of a function with the fully qualified class path, as shown in Listing 6.24.

Listing 6.24 **Getting Help for Spark SQL Functions**

Click here to view code image

```
from pyspark.sql import DataFrame
print(DataFrame.sample.__doc__)
# returns:
```

```
# Returns a sampled subset of this :class:`DataFrame`.
#.. note:: This is not guaranteed to provide exactly the fraction
specified of the
# total
#  count of the given :class:`DataFrame`.
# >>> df.sample(False, 0.5, 42).count()
# 2
#.. versionadded:: 1.3
```

DataFrame Built-in Functions

Numerous functions available in Spark SQL are present in most other common DBMS implementations of SQL. Using the Python Spark API, these built-in functions are available through the `pyspark.sql.functions` module. Functions include scalar and aggregate functions and can operate on fields, columns, or rows, depending on the function. Table 6.4 shows a sampling of the functions available in the `pyspark.sql.functions` library.

Table 6.4 **Examples of Built-in Functions Available in Spark SQL**

Type	**Available Functions**
String functions	`startswith`, `substr`, `concat`, `lower`, `upper`, `regexp_extract`, `regexp_replace`
Math functions	`abs`, `ceil`, `floor`, `log`, `round`, `sqrt`
Statistical functions	`avg`, `max`, `min`, `mean`, `stddev`
Date functions	`date_add`, `datediff`, `from_utc_timestamp`
Hashing functions	`md5`, `sha1`, `sha2`
Algorithmic functions	`soundex`, `levenshtein`
Windowing	`over`, `rank`, `dense_rank`, `lead`, `lag`, `ntile`

functions

Implementing User-Defined Functions in the DataFrame API

If you can’t find a function for what you want to do, you can create a user-defined function (UDF) in Spark SQL. You can create column-level UDFs to incorporate into Spark programs by using the `udf()` method described next.

`udf()`

Syntax:

Click here to view code image

```
pyspark.sql.functions.udf(func, returnType=StringType)
```

The `udf` method creates a column expression representing a user-defined function; `func` is a named or anonymous function, using the `lambda` syntax, that operates on a column within a DataFrame row. The `returnType` argument specifies the datatype of the object returned from the function. This type is a member of `pyspark.sql.types` or a subtype of the `pyspark.sql.types.DataType` class.

Suppose you want to define functions to convert decimal latitudinal and longitudinal coordinates to their geopositional direction with respect to the equator and the prime meridian. Listing 6.25 demonstrates creating two UDFs to take the decimal latitude and longitude coordinates and return `N`, `S`, `E`, or `W`, as appropriate.

Listing 6.25 **User-Defined Functions in Spark SQL**

Click here to view code image

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
df = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
lat2dir = udf(lambda x: 'N' if x > 0 else 'S', StringType())
lon2dir = udf(lambda x: 'E' if x > 0 else 'W', StringType())
df.select(df.lat, lat2dir(df.lat).alias('latdir'),
          df.long, lon2dir(df.lat).alias('longdir')) \
          .show(5)
# returns:
 # +---------+------+----------+-------+
# |      lat|latdir|      long|longdir|
# +---------+------+----------+-------+
# |37.329732|     N|-121.901782|      E|
# |37.330698|     N|-121.888979|      E|
# |37.333988|     N|-121.894902|      E|
# |37.331415|     N|  -121.8932|      E|
# |37.336721|     N|-121.894074|      E|
# +---------+------+----------+-------+
# only showing top 5 rows
```

Operations on Multiple DataFrames

Set operations, such as `join()` and `union()`, are common requirements for DataFrames because they are integral operations in relational SQL programming.

Joining DataFrames support all join operations supported in the RDD API and in HiveQL, including inner joins, outer joins, and left semi-joins.

`join()`

Syntax:

Click here to view code image

```
DataFrame.join(other, on=None, how=None)
```

The `join()` method creates a new DataFrame from the results of a join operation against the DataFrame referenced in the `other` argument (the right side of the argument). The `on` argument specifies a column, a list of columns, or an expression to evaluate the join operation. The `how` argument specifies the type of join to be performed. Valid values include `inner` (default), `outer`, `left_outer`, `right_outer`, and `leftsemi`.

Consider a new entity from the bike-share dataset called `trips`, which includes two fields, `start_terminal` and `end_terminal`, that correspond to `station_id` in the `stations` entity. Listing 6.26 demonstrates an inner join between these two entities, using the `join()` method.

Listing 6.26 **Joining DataFrames in Spark SQL**

Click here to view code image

```
trips = spark.table("trips")
stations = spark.table("stations")
joined = trips.join(stations, trips.startterminal ==
stations.station_id)
joined.printSchema()
# returns:
# root
#  |-- tripid: integer (nullable = true)
#  |-- duration: integer (nullable = true)
#  |-- startdate: string (nullable = true)
#  |-- startstation: string (nullable = true)
#  |-- startterminal: integer (nullable = true)
#  |-- enddate: string (nullable = true)
#  |-- endstation: string (nullable = true)
 #  |-- endterminal: integer (nullable = true)
#  |-- bikeno: integer (nullable = true)
#  |-- subscribertype: string (nullable = true)
#  |-- zipcode: string (nullable = true)
#  |-- station_id: integer (nullable = true)
#  |-- name: string (nullable = true)
#  |-- lat: double (nullable = true)
#  |-- long: double (nullable = true)
#  |-- dockcount: integer (nullable = true)
#  |-- landmark: string (nullable = true)
#  |-- installation: string (nullable = true)
joined.select(joined.startstation, joined.duration) \
      .show(2)
# returns:
# +--------------------+--------+
# |        startstation|duration|
# +--------------------+--------+
# |Harry Bridges Pla...|     765|
# |San Antonio Shopp...|    1036|
# +--------------------+--------+
# only showing top 2 rows
```

Other set operations such as `intersect()` and `subtract()` are available functions for Spark SQL DataFrames and function like the equivalent RDD functions described previously in this book. In addition, a `unionAll()` method is available for DataFrames instead of `union()`, also described previously. Note that if you need to remove duplicates, you can do so after the `unionAll()` operation by using the aforementioned `distinct()` or `drop_duplicates()` functions.

The DataFrame API also includes several standard methods for sorting or ordering, as described in the following sections.

`orderBy()`

Syntax:

Click here to view code image

```
DataFrame.orderBy(cols, ascending)
```

The `orderBy()` method creates a new DataFrame ordered by the columns specified in the `cols` argument; `ascending` is a Boolean argument that defaults to `True`, which determines the sort order for the column. Listing 6.27 shows an example of the `orderBy()` function.

Listing 6.27 **Ordering a DataFrame**

Click here to view code image

```
stations = spark.read.parquet('hdfs:///user/hadoop/stations.parquet')
stations.orderBy([stations.name], ascending=False) \
    .select(stations.name) \
    .show(2)
# returns:
 # +--------------------+
# |                name|
# +--------------------+
# |Yerba Buena Cente...|
# |Washington at Kea...|
# +--------------------+
# only showing top 2 rows
```

Note that `sort()` is a function synonymous with `orderBy()` in the DataFrame API.

Grouping is a common precursor to performing aggregations on a column or columns in a DataFrame. The DataFrame API includes the `groupBy()` method (also aliased by `groupby()`), which groups the DataFrame on specific columns. This function returns a `pyspark.sql.GroupedData` object, a special type of DataFrame that contains grouped data exposing common aggregate functions, such as `sum()` and `count()`.

`groupBy()`

Syntax:

```
DataFrame.groupBy(cols)
```

The `groupBy()` method creates a new DataFrame containing the input DataFrame grouped by the column or columns specified in the `cols` argument. Listing 6.28 demonstrates the use of `groupBy()` to average trip durations from the `trips` entity in the bike-share dataset.

Listing 6.28 **Grouping and Aggregating Data in DataFrames**

Click here to view code image

```
trips = spark.table("trips")
averaged = trips.groupBy([trips.startterminal]).avg('duration') \
                .show(2)
# returns:
# +-------------+-----------------+
# |startterminal|    avg(duration)|
# +-------------+-----------------+
# |           31|2747.6333021515434|
# |           65| 626.1329988365329|
# +-------------+-----------------+
# only showing top 2 rows
```

Caching, Persisting, and Repartitioning DataFrames

The DataFrame API supports methods for caching, persisting, and repartitioning that are similar to those in the Spark RDD API for these operations.

Methods for caching and persisting DataFrames include `cache()`, `persist()`, and `unpersist()`, which behave like the RDD functions with the same names. In addition, Spark SQL adds the `cacheTable()` method, which caches a table from Spark SQL or Hive in memory. The `clearCache()` method removes a cached table from memory. DataFrames also support the `coalesce()` and `repartition()` methods for repartitioning DataFrames.

Saving DataFrame Output

The `DataFrameWriter` is the interface used to write a DataFrame to external storage systems such as a file system or a database. The `DataFrameWriter` is accessible using `DataFrame.write()`. The following sections provide some examples.

Writing Data to a Hive Table

Earlier in this chapter, you saw how to load data into a DataFrame from a Hive table. Similarly, you may often need to write data from a DataFrame to a Hive table; you can do this by using the `saveAsTable()` function.

`saveAsTable()`

Syntax:

Click here to view code image

```
DataFrame.write.saveAsTable(name, format=None, mode=None,
partitionBy=None)
```

The `saveAsTable()` method writes the data from a DataFrame into the Hive table specified in the `name` argument. The `format` argument specifies the output format for the target table; the default is Parquet format. Likewise, `mode` is the behavior with respect to an existing object, and valid values are `append`, `overwrite`, `error`, and `ignore`. Listing 6.29 shows an example of the `saveAsTable()` method.

Listing 6.29 **Saving a DataFrame to a Hive Table**

Click here to view code image

```
stations = spark.table("stations")
stations.select([stations.station_id,stations.name]).write \
        .saveAsTable("station_names")
# load new table
```

```
station_names = spark.table("station_names")
station_names.show(2)
# returns:
# +----------+-------------------+
# |station_id|               name|
# +----------+-------------------+
# |         2|San Jose Diridon ...|
# |         3|San Jose Civic Ce...|
# +----------+-------------------+
# only showing top 2 rows
```

There is also a similar method in the DataFrame API named `insertInto()`.

Writing Data to Files

Data from DataFrames can write to files in any supported filesystem: local, network, or distributed. Output is written as a directory with files emitted for each partition, much as with the RDD output examples shown earlier in this chapter.

Comma-separated values (CSV) is a common file export format. DataFrames can export to CSV files by using the `DataFrameWriter.write.csv()` method.

Parquet is a popular columnar format that is optimized for Spark SQL. You have seen several examples so far from Parquet format files. DataFrames can write to Parquet format files by using the `DataFrameWriter.write.parquet()` method.

`write.csv()`

Syntax:

Click here to view code image

```
DataFrameWriter.write.csv(path,
                          mode=None,
                          compression=None,
                          sep=None,
                          quote=None,
                          escape=None,
                          header=None,
                          nullValue=None,
                          escapeQuotes=None,
                          quoteAll=None,
                          dateFormat=None,
                          timestampFormat=None,
                          ignoreLeadingWhiteSpace=None,
                          ignoreTrailingWhiteSpace=None)
```

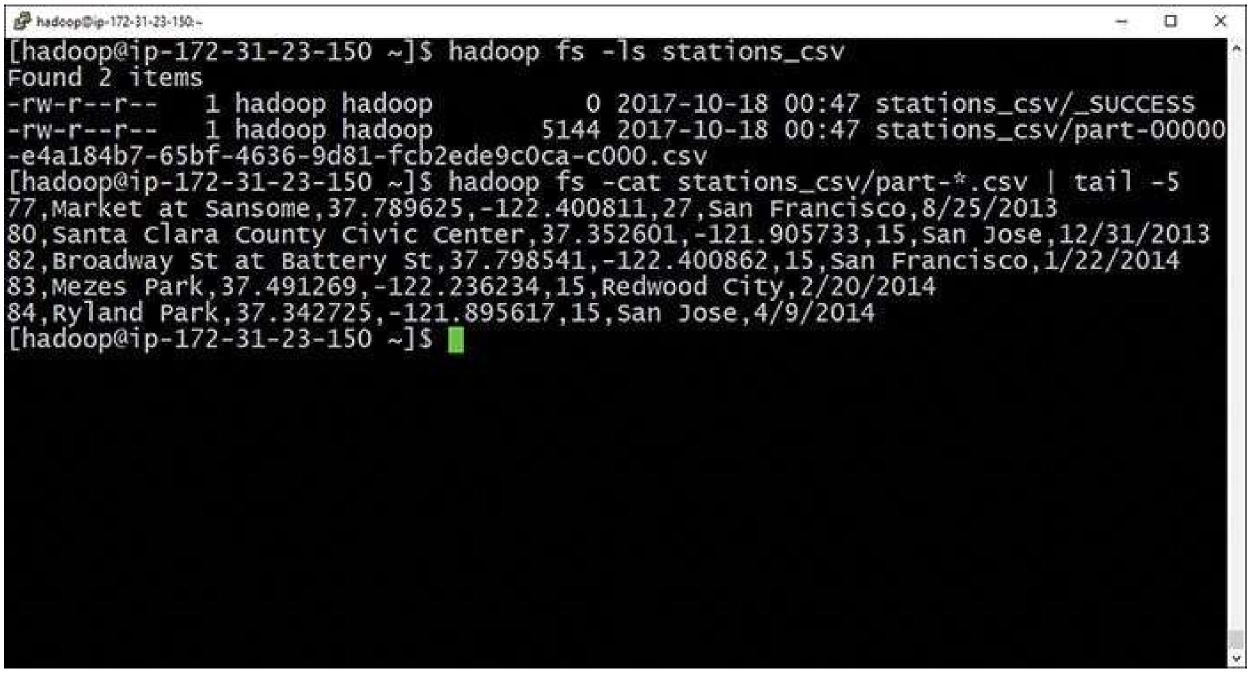
The `write.csv()` method of the `DataFrameWriter` class, accessed through the `DataFrame.write.csv()` interface, writes the content of a DataFrame to CSV files in the path specified by the `path` argument. The `mode` argument defines the behavior if a target directory already exists for the operation; valid values include `append`, `overwrite`, `ignore`, and `error` (the default). The `mode` argument is available on all `DataFrame.write()` method. Additional arguments define the desired formatting for the output CSV files. For example, the `quoteAll` argument indicates whether all values should always be enclosed in quotes. Specific information on all arguments available for the `write.csv()` method is available at https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Data
Listing 6.30 demonstrates the use of the `write.csv()` method.

Listing 6.30 **Writing a DataFrame to a CSV File or Files**

Click here to view code image

```
spark.table("stations") \
    .write.csv("stations_csv")
```

The target for a `write.csv()` operation could be a local filesystem (using the `file://` scheme), HDFS, S3, or any other filesystem available to you and configured for access from your Spark environment. In Listing 6.30, the filesystem defaults to the home directory in HDFS of the user running the command; `stations_csv` is a directory in HDFS, the contents of which are shown in Figure 6.6.



```
[hadoop@ip-172-31-23-150 ~]$ hadoop fs -ls stations_csv
Found 2 items
-rw-r--r--   1 hadoop hadoop          0 2017-10-18 00:47 stations_csv/_SUCCESS
-rw-r--r--   1 hadoop hadoop       5144 2017-10-18 00:47 stations_csv/part-00000
-e4a184b7-65bf-4636-9d81-fcb2ede9c0ca-c000.csv
[hadoop@ip-172-31-23-150 ~]$ hadoop fs -cat stations_csv/part-*.csv | tail -5
77,Market at Sansome,37.789625,-122.400811,27,San Francisco,8/25/2013
80,Santa Clara County Civic Center,37.352601,-121.905733,15,San Jose,12/31/2013
82,Broadway St at Battery St,37.798541,-122.400862,15,San Francisco,1/22/2014
83,Mezes Park,37.491269,-122.236234,15,Redwood City,2/20/2014
84,Ryland Park,37.342725,-121.895617,15,San Jose,4/9/2014
[hadoop@ip-172-31-23-150 ~]$
```

Figure 6.6 HDFS directory contents from a `write.csv()` DataFrame operation.

`parquet()`

Syntax:

Click here to view code image

```
DataFrameWriter.write.parquet(path, mode=None, partitionBy=None)
```

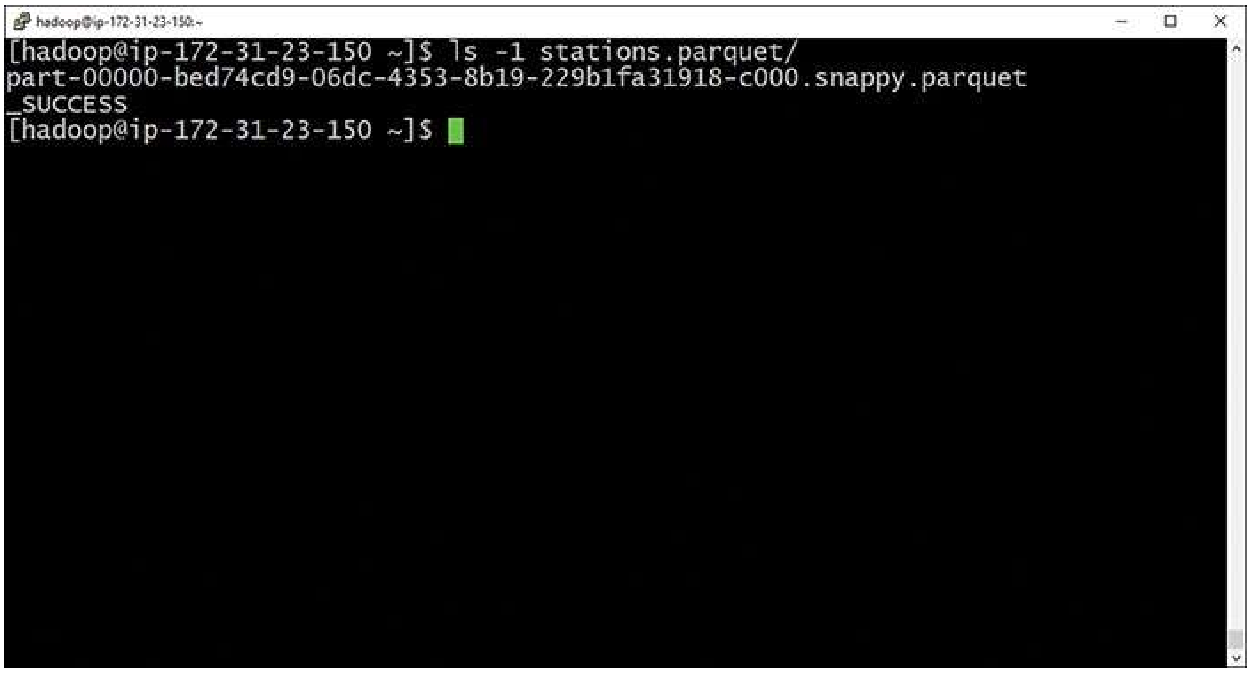
The `write.parquet()` method writes out the data from a DataFrame to a directory containing Parquet format files. Files compress according to the compression configuration settings in the current `SparkContext`. The `mode` argument specifies the behavior if the directory or files exist. Valid values for `mode` are `append`, `overwrite`, `ignore`, and `error` (the default); `partitionBy` specifies the names of columns by which to partition the output files (using the hash partitioner). Listing 6.31 demonstrates using `parquet()` to save a DataFrame to a Parquet file using Snappy compression.

Listing 6.31 **Saving a DataFrame to a Parquet File or Files**

Click here to view code image

```
spark = SparkSession.builder \
    .config("spark.sql.parquet.compression.codec.", "snappy") \
    .getOrCreate()
stations = spark.table("stations")
stations.select([stations.station_id,stations.name]).write \
        .parquet("file:///home/hadoop/stations.parquet",
mode='overwrite')
```

Figure 6.7 shows a listing of the local directory containing the Snappy-compressed Parquet-formatted output file from the operation performed in Listing 6.31.



```
[hadoop@ip-172-31-23-150 ~]$ ls -1 stations.parquet/
part-00000-bed74cd9-06dc-4353-8b19-229b1fa31918-c000.snappy.parquet
_SUCCESS
[hadoop@ip-172-31-23-150 ~]$
```

Figure 6.7 Output files created from a `write.parquet()` operation.

ORC files can be written using the `orc()` method, which is similar in usage to `parquet()`. JSON files can also be written using the `json()` method.

You can save DataFrames to external JDBC-compliant databases by using the `DataFrameWriter.write.jdbc()` method.

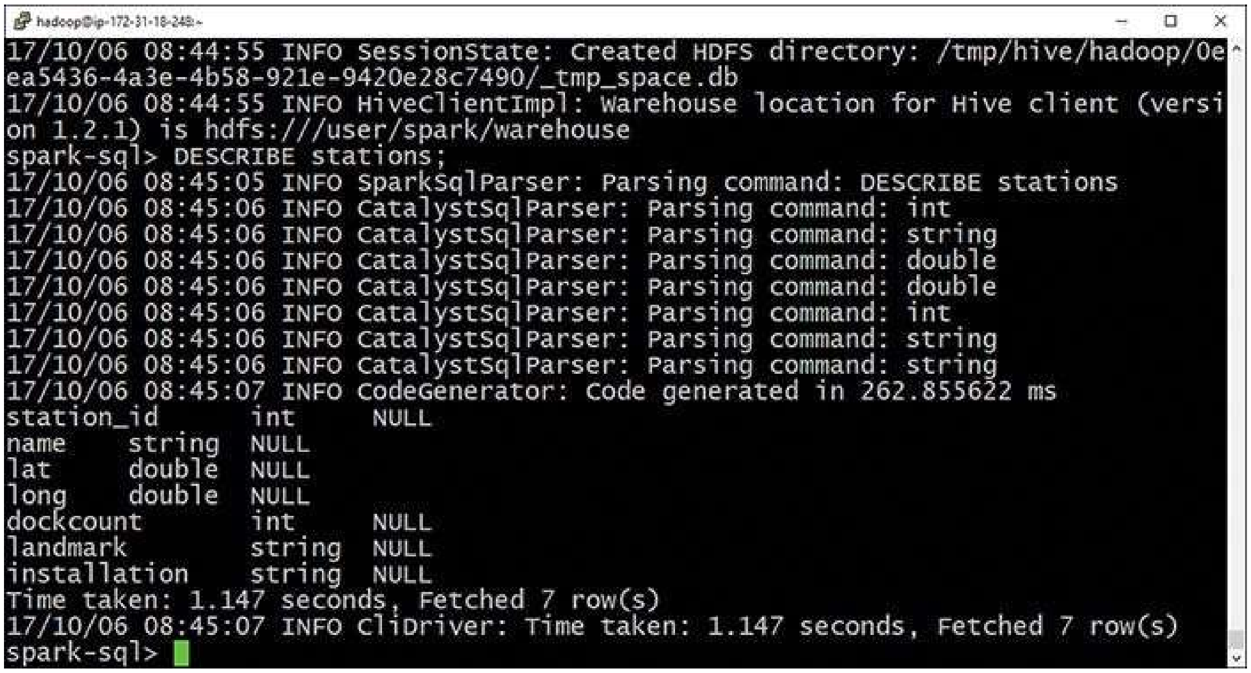
Accessing Spark SQL

So far in this chapter, the examples of Spark SQL have been within the Python (PySpark) interface. However, PySpark may not be the appropriate interface for users who are not programmers. A SQL shell or access to the Spark SQL engine from a visualization tool such as Tableau or Excel via ODBC may be more applicable.

Accessing Spark SQL Using the `spark-sql` Shell

Spark includes a SQL shell utility called `spark-sql` in the `bin` directory of your Spark installation. The `spark-sql` shell program is a lightweight REPL (read-evaluate-print loop) shell that can access Spark SQL and Hive using your local configuration and Spark Driver binaries. The shell accepts HiveQL statements, including metadata operations such as `SHOW TABLES` and `DESCRIBE`. Figure 6.8 shows an example of the `spark-sql` shell.

Note that `spark-sql` is useful for testing SQL commands locally as a developer, but it is limited because it's not a SQL engine that is accessible by other users and remote applications. This is where the Thrift JDBC/ODBC server comes into play.



```
17/10/06 08:44:55 INFO SessionState: Created HDFS directory: /tmp/hive/hadoop/0e
ea5436-4a3e-4b58-921e-9420e28c7490/_tmp_space.db
17/10/06 08:44:55 INFO HiveClientImpl: Warehouse location for Hive client (versi
on 1.2.1) is hdfs:///user/spark/warehouse
spark-sql> DESCRIBE stations;
17/10/06 08:45:05 INFO SparkSqlParser: Parsing command: DESCRIBE stations
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: int
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: string
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: double
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: double
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: int
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: string
17/10/06 08:45:06 INFO CatalystSqlParser: Parsing command: string
17/10/06 08:45:07 INFO CodeGenerator: Code generated in 262.855622 ms
station_id      int     NULL
name    string  NULL
lat     double  NULL
long    double  NULL
dockcount       int     NULL
landmark        string  NULL
installation    string  NULL
Time taken: 1.147 seconds, Fetched 7 row(s)
17/10/06 08:45:07 INFO CliDriver: Time taken: 1.147 seconds, Fetched 7 row(s)
spark-sql>
```

Figure 6.8 The `spark-sql` shell.

Running the Thrift JDBC/ODBC Server

Spark SQL is useful as a distributed query engine with a JDBC/ODBC interface. As with the `spark-sql` shell, the JDBC/ODBC server enables users to run

SQL queries without writing Python or Scala Spark code. External applications, such as visualization tools, can connect to the server and interact directly with Spark SQL.

The JDBC/ODBC interface is implemented through a Thrift JDBC/ODBC server. *Thrift* is an Apache project used for cross-language service development. The Spark SQL Thrift JDBC/ODBC server is based on the HiveServer2 project, a server interface that enables remote clients to execute queries against Hive and retrieve the results.

The Thrift JDBC/ODBC server is included with the Spark release. To run the server, execute the following command:

Click here to view code image

```
$SPARK_HOME/sbin/start-thriftserver.sh
```

All valid `spark-submit` command line arguments, such as `--master`, are accepted by the `start-thriftserver.sh` script. In addition, you can supply Hive-specific properties by using the `--hiveconf` option. The Thrift JDBC/OBDC server listens on port 10000, but you can change this by using a special environment variable, as shown here:

Click here to view code image

```
export HIVE_SERVER2_THRIFT_PORT=<customport>
```

You can use `beeline`, discussed next, to test the JDBC/ODBC server. To stop the Thrift server, simply execute the following:

Click here to view code image

```
$SPARK_HOME/sbin/stop-thriftserver.sh
```

Using `beeline`

You can use `beeline`, a command line shell, to connect to HiveServer2 or the Spark SQL Thrift JDBC/ODBC server. `beeline` is a lightweight JDBC client application that is based on the SQLLine CLI project (http://sqlline.sourceforge.net/).

Like SQLLine, `beeline` is a Java console–based utility for connecting to relational databases and executing SQL commands. It is designed to function similarly to other command line database access utilities, such as `sqlplus` for Oracle, `mysql` for MySQL, and `isql` or `osql` for Sybase/ SQL Server.

Because `beeline` is a JDBC client, you can use it to test the Spark SQL JDBC

Thrift server when you start it. Use the `beeline` CLI utility included with the Spark release as follows:

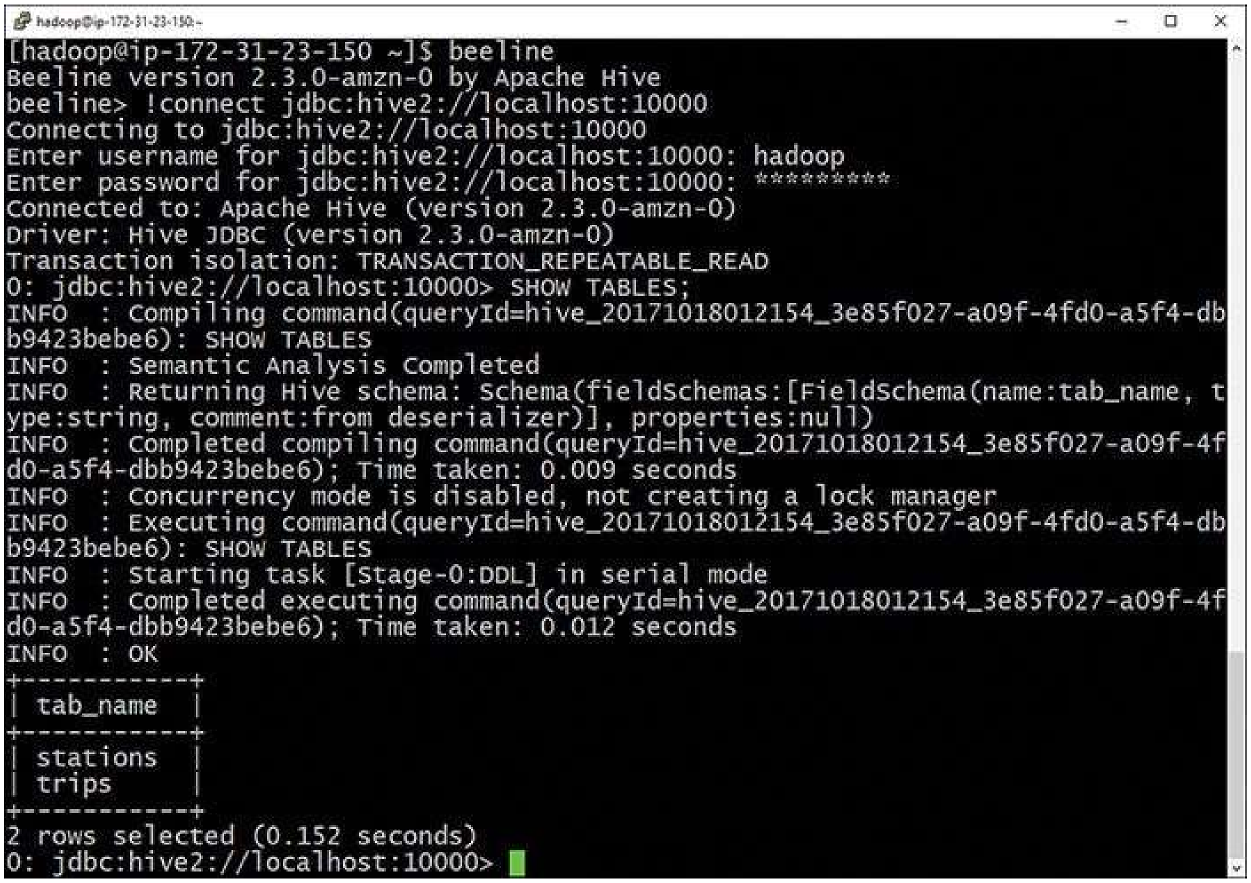
```
$SPARK_HOME/bin/beeline
```

At the `beeline` prompt, you need to connect to a JDBC server—the Spark SQL Thrift server you started previously. Do this as follows:

Click here to view code image

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

You are prompted for a username and password to connect to the server. Figure 6.9 shows an example of a `beeline` CLI session connecting to the Spark SQL Thrift server.



```
[hadoop@ip-172-31-23-150 ~]$ beeline
Beeline version 2.3.0-amzn-0 by Apache Hive
beeline> !connect jdbc:hive2://localhost:10000
Connecting to jdbc:hive2://localhost:10000
Enter username for jdbc:hive2://localhost:10000: hadoop
Enter password for jdbc:hive2://localhost:10000: *********
Connected to: Apache Hive (version 2.3.0-amzn-0)
Driver: Hive JDBC (version 2.3.0-amzn-0)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000> SHOW TABLES;
INFO  : Compiling command(queryId=hive_20171018012154_3e85f027-a09f-4fd0-a5f4-db
b9423bebe6): SHOW TABLES
INFO  : Semantic Analysis Completed
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:tab_name, t
ype:string, comment:from deserializer)], properties:null)
INFO  : Completed compiling command(queryId=hive_20171018012154_3e85f027-a09f-4f
d0-a5f4-dbb9423bebe6); Time taken: 0.009 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=hive_20171018012154_3e85f027-a09f-4fd0-a5f4-db
b9423bebe6): SHOW TABLES
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=hive_20171018012154_3e85f027-a09f-4f
d0-a5f4-dbb9423bebe6); Time taken: 0.012 seconds
INFO  : OK
+-----------+
| tab_name  |
+-----------+
| stations  |
| trips     |
+-----------+
2 rows selected (0.152 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 6.9 The `beeline` Spark SQL JDBC client.

Using External Applications via JDBC/ODBC

The Spark JDBC/ODBC Thrift server can also connect to other JDBC/ODBC client applications, such as Tableau or Excel. This usually requires that you install the relevant JDBC/ODBC drivers on your client. You can then create a data source and connect to Spark SQL to access and process data in Hive. Consult your visualization tool vendor for more information or to obtain the specific drivers required.

Exercise: Using Spark SQL

This exercise shows how to start a Spark SQL Thrift server and use the `beeline` client utility to connect to the server. You will create Hive tables based on sample data and use `beeline` and Thrift to run a SQL query against the data, executed by Spark SQL. You will use the bike-share dataset used for exercises in the previous chapter. Follow these steps:

1. Start the JDBC/ODBC Thrift server:

Click here to view code image

```
$ sudo $SPARK_HOME/sbin/start-thriftserver.sh \
--master local \
--hiveconf hive.server2.thrift.port=10001 \
--hiveconf hive.server2.thrift.bind.host=10001
```

   You can start the server in YARN mode instead by using `--master yarn-cluster` if you have a YARN cluster available to you.

2. Open a `beeline` session:

```
$SPARK_HOME/bin/beeline
```

3. At the `beeline>` prompt, create a connection to your Thrift server:

Click here to view code image

```
beeline> !connect jdbc:hive2://localhost:10001
Enter username for jdbc:hive2://localhost:10001: hadoop
Enter password for jdbc:hive2://localhost:10001: *********
```

   You are prompted for a username and password, as shown above. The username provided must exist on the Thrift server and have the appropriate permissions on the filesystem.

4. After you connect to the server, create the `trips` table from the bike-share demo by entering the following HiveQL DDL command:

Click here to view code image

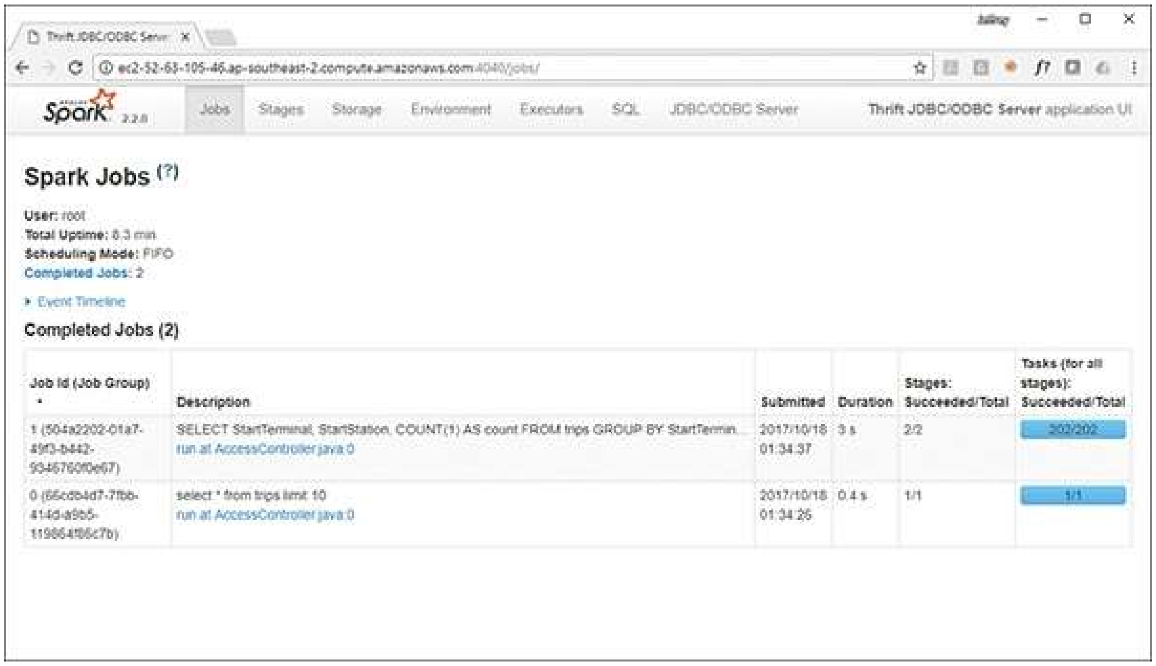
```
CREATE EXTERNAL TABLE trips (
TripID int,
Duration int,
StartDate string,
StartStation string,
StartTerminal int,
EndDate string,
EndStation string,
EndTerminal int,
BikeNo int,
SubscriberType string, ZipCode string )
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 'file:///opt/spark/data/bike-share/trips/';
```

5. Execute the following SQL query against the table you just created:

Click here to view code image

```
SELECT StartTerminal, StartStation, COUNT(1) AS count
FROM trips
GROUP BY StartTerminal, StartStation
ORDER BY count DESC
LIMIT 10;
```

6. View your Spark application web UI to confirm that your query executed using Spark SQL. Recall that this is accessible using port 4040 of your localhost if you are running Spark locally or the application master host if you are using YARN (accessible from the Resource Manager UI). Figure 6.10 shows the SQL tab in the application UI as well.


Thrift JDBC/ODBC Server
ec2-52-63-105-46.ap-southeast-2.compute.amazonaws.com:4040/jobs/
Spark 2.2.0
Jobs
Stages
Storage
Environment
Executors
SQL
JDBC/ODBC Server
Thrift JDBC/ODBC Server application UI

Spark Jobs (?)

User: root
Total Uptime: 8.3 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

Completed Jobs (2)

Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1 (504a2202-01a7-49f3-b442-9346760f0e67)	SELECT StartTerminal, StartStation, COUNT(1) AS count FROM trips GROUP BY StartTermin... run at AccessController.java:0	2017/10/18 01:34:37	3 s	2/2	202/202
0 (66cdb4d7-7fbb-414d-a9b5-119864f66c7b)	select * from trips limit 10 run at AccessController.java:0	2017/10/18 01:34:26	0.4 s	1/1	1/1

Figure 6.10 Spark application UI for a Spark SQL session.

Using Spark with NoSQL Systems

Increasingly aggressive non-functional and non-relational requirements necessitate alternative approaches to data storage, management, and processing. Enter NoSQL, a new data paradigm that allows you to look at data in terms of cells rather than just the relational paradigm of tables, rows, and columns. This is not to say that the relational database is dead—far from it—but the NoSQL approach provides a new set of capabilities to solve today’s—and tomorrow’s—problems.

This section introduces NoSQL systems and methodologies and looks at their integration with Spark-processing workflows. First, let’s look at some of the core concepts of NoSQL systems.

Introduction to NoSQL

There is some friendly disagreement about what NoSQL means; some say it means “not SQL,” others say “not only SQL,” and others have other interpretations or definitions. Regardless of the disagreement around the nomenclature, NoSQL systems have specific defining characteristics and come in different variants.

NoSQL System Characteristics

All NoSQL variants share some common properties, including the following:

- **They are schemaless at design time and "schema-on-read" at runtime:** This means they do not have predefined columns, but columns are created with each `PUT` (`INSERT`) operation, and each record, document, or data instance can have a different schema than the previous instance.
- **Data has no predefined relationship to any other object:** This means there is no concept of foreign keys or referential integrity, declarative or otherwise. Relationships may exist between data objects or instances, but they are discovered or leveraged at runtime rather than prescribed at design time.
- **Joins are typically avoided:** In most NoSQL implementations, joins are kept to an absolute minimum or avoided altogether. This is typically accomplished by denormalizing data, often with the trade-off of storing duplicate data. However, with most NoSQL implementations leveraging cost-efficient commodity or cloud infrastructure, the material cost is offset by the computation cost reduction of not having to perform excessive joins when the data is accessed.

In all cases, there is no logical or physical model that dictates how data is structured, unlike with a third normal form data warehouse or an online transaction processing system.

Moreover, NoSQL systems are typically distributed (for example, Apache Cassandra, HBase) and structured for fast lookups. Write operations are typically faster and more scalable as well, as many of the processes of traditional relational database systems that lead to overhead are not used, such as datatype or domain checks, atomic/blocking transactions, and management of transaction isolation levels.

In the majority of cases, NoSQL systems are built for scale and scalability (from petabytes of storage to queries bounded in terabytes), performance, and low friction (or having the ability to adapt to changes). NoSQL systems are often comparatively analytically friendly, as they provide a denormalized structure, which is conducive to feature extraction, machine learning, and scoring.

Types of NoSQL Systems

As mentioned earlier in this chapter, NoSQL systems come in several variants or categories: key/value stores, document stores, and graph stores (see Table 6.5).

Table 6.5 **Types of NoSQL Systems**

Type	**Description**	**Examples**
Key/value stores/ column family stores	A key/value store contains a set or sets of indexed keys and associated values. Values are typically uninterpreted byte arrays but can represent complex objects such as nested maps, structs, or lists. The schema is not defined at design time; however, some storage properties such as column families, which are effectively storage containers for values, and compression attributes can be defined at table design time.	HBase, Cassandra, and DynamoDB
Document stores	Document stores, or document databases, store complex objects, documents such as JSON or BSON objects, or other complex nested objects. Each document is assigned a key or document ID, and the contents are the semi-structured document data.	MongoDB and CouchDB
Graph stores	Graph stores are based on graph theory and used to describe relationships between objects or entities.	Neo4J and GraphBase

Using Spark with HBase

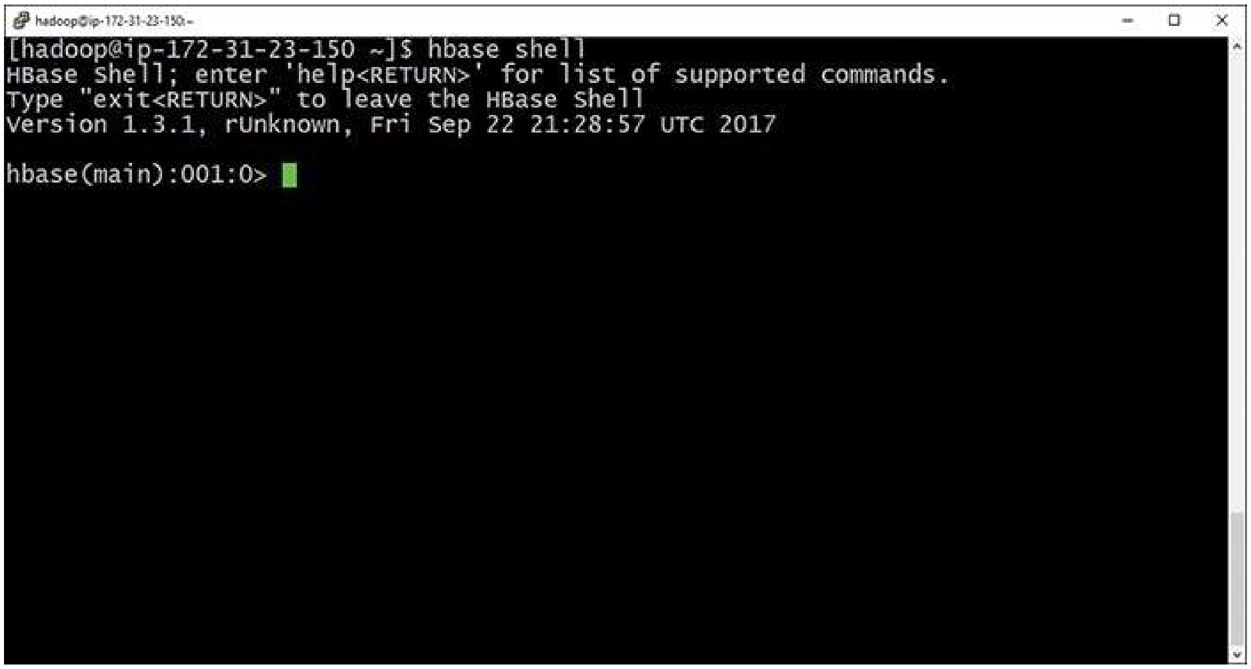
HBase is a Hadoop ecosystem project designed to deliver a distributed, massively scalable key/value store on top of HDFS. Before we discuss the use of Spark with HBase, it is important that you understand some basic HBase concepts.

Introduction to HBase

HBase stores data as a sparse, multidimensional, sorted map. The map is indexed by its key (the row key), and values are stored in cells, each consisting of a column key and a column value. The row key and column keys are strings, and the column value is an uninterpreted byte array that could represent any primitive or complex datatype. HBase is multidimensional; that is, each cell is versioned with a timestamp.

At table design time, one or more column families is defined. Column families are used as physical storage groups for columns. Different column families may have different physical storage characteristics, such as block size, compression settings, or the number of cell versions to retain.

Although there are projects such as Hive and Phoenix to provide SQL-like access to data in HBase, the natural methods for accessing and updating data in HBase are essentially `get`, `put`, `scan`, and `delete`. HBase includes a shell program as well as programmatic interfaces for multiple languages. The HBase shell is an interactive Ruby REPL shell with access to HBase API functions to create and modify tables and read and write data. The shell application is accessible only by entering `hbase shell` on a system with the HBase client binaries and configuration available (see Figure 6.11).



```
[hadoop@ip-172-31-23-150 ~]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.3.1, rUnknown, Fri Sep 22 21:28:57 UTC 2017

hbase(main):001:0>
```

Figure 6.11 HBase shell.

Listing 6.32 demonstrates the use of `hbase shell` to create a table and insert data into the table.

Listing 6.32 **Creating a Table and Inserting Data in HBase**

Click here to view code image

```
hbase> create 'my-hbase-table', \
hbase* {NAME => 'cf1', COMPRESSION => 'SNAPPY', VERSIONS => 20}, \
hbase* {NAME => 'cf2'}
hbase> put 'my-hbase-table', 'rowkey1', 'cf1:fname', 'John'
hbase> put 'my-hbase-table', 'rowkey1', 'cf1:lname', 'Doe'
hbase> put 'my-hbase-table', 'rowkey2', 'cf1:fname', 'Jeffrey'
hbase> put 'my-hbase-table', 'rowkey2', 'cf1:lname', 'Aven'
hbase> put 'my-hbase-table', 'rowkey2', 'cf1:city', 'Hayward'
hbase> put 'my-hbase-table', 'rowkey2', 'cf2:password',
'c9cb7dc02b3c0083eb70898e549'
```

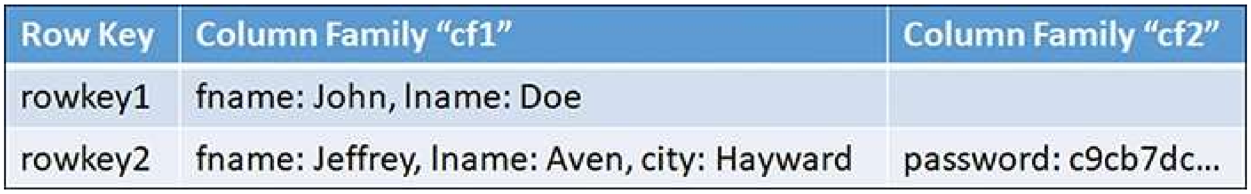
The `create` statement creates a new HBase table with two column families: `cf1` and `cf2`. One column family is configured to use compression, and the other is not. The subsequent `put` statements insert data into a cell as defined by the row key (`rowkey1` or `rowkey2`, in this case) and a column specified in the format `<column_family>:<column_name>`. Unlike with a traditional database, the columns are not defined at table design time and are not typed. (Recall that all data is an uninterpreted array of bytes.) A `scan` command of the new table is shown in Listing 6.33.

Listing 6.33 **Scanning the HBase Table**

Click here to view code image

```
hbase> scan 'my-hbase-table'
ROW                      COLUMN+CELL
 rowkey1                 column=cf1:fname, timestamp=1508291546300,
value=John
 rowkey1                 column=cf1:lname, timestamp=1508291560041,
value=Doe
 rowkey2                 column=cf1:city, timestamp=1508291579756,
value=Hayward
 rowkey2                 column=cf1:fname, timestamp=1508291566663,
value=Jeffrey rowkey2               column=cf1:lname,
timestamp=1508291572939, value=Aven
 rowkey2                 column=cf2:password, timestamp=1508291585467,
value= c9cb7dc02b3c0083eb70898e549
2 row(s) in 0.0390 seconds
```

Figure 6.12 shows a conceptual view of the data inserted in this example.



Row Key	Column Family “cf1”	Column Family “cf2”
rowkey1	fname: John, lname: Doe	
rowkey2	fname: Jeffrey, lname: Aven, city: Hayward	password: c9cb7dc...

Figure 6.12 HBase data.

As you can see in Figure 6.12, HBase supports *sparsity*. That is, not every column needs to exist in each row in a table, and nulls are not stored.

Although HBase data is stored on HDFS, an immutable file system, HBase allows in-place updates to cells in HBase tables. It does this by creating a new version of the cell with a new timestamp if the column key already exists, and then a background compaction process collapses multiple files into a smaller number of larger files.

Listing 6.34 demonstrates an update to an existing cell and the resultant new version.

Listing 6.34 **Updating a Cell in HBase**

Click here to view code image

```
hbase> put 'my-hbase-table', 'rowkey2', 'cf1:city', 'Melbourne'
hbase> get 'my-hbase-table', 'rowkey2', {COLUMNS => ['cf1:city']}
COLUMN                      CELL
 cf1:city                   timestamp=1508292292811, value=Melbourne
1 row(s) in 0.0390 seconds
hbase> get 'my-hbase-table', 'rowkey2', {COLUMNS => ['cf1:city'],
VERSIONS => 2}
COLUMN                      CELL
 cf1:city                   timestamp=1508292546999, value=Melbourne
 cf1:city                   timestamp=1508292538926, value=Hayward
1 row(s) in 0.0110 seconds
```

Notice in Listing 6.34 that HBase supports cell versioning. The number of versions retained is defined by the column family upon table creation.

HBase data is stored in *HFile* objects in HDFS. An HFile object is the intersection of a column family (storage group) and a sorted range of row keys. Ranges of sorted row keys are referred to as *regions* and are also known as *tablets* in other implementations. Regions are assigned to a *region server* by HBase; see Figure 6.13. Regions are used to provide fast row key lookups, as the regions and row keys they contain are known by HBase. HBase splits and compacts regions as necessary as part of its normal operation. Non-row key–based lookups, such as looking for a column key and value satisfying a criterion, are slower. However, HBase uses *bloom filters* to help expedite the search.

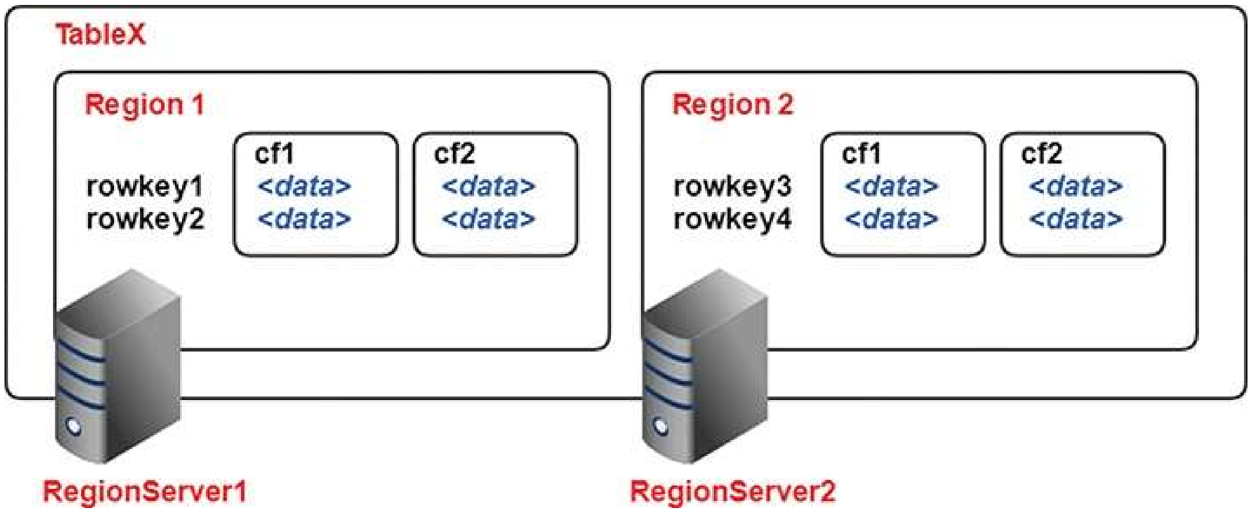


Figure 6.13 HBase regions.

HBase and Spark

The most failsafe method of reading and writing to HBase from Spark using the Python API is to use the HappyBase Python package (https://happybase.readthedocs.io/en/latest/). HappyBase is a Python library built for accessing and manipulating data in an HBase cluster. To use HappyBase, you must first install the Python package by using `pip` or `easy_install`, as shown here:

```
$ sudo pip install happybase
```

If you require more scalability, consider using either the Scala API for Spark or various third-party HBase connectors for Spark, available as Spark packages (https://spark-packages.org/).

Exercise: Using Spark with HBase

Setting up HBase is beyond the scope of this book. However, HBase is a normal component of many Hadoop vendor distributions, such as Cloudera and Hortonworks, including the sandbox VM environments provided by these vendors. You can provision HBase as an additional application in the AWS EMR-managed Hadoop service offering. For this exercise, you need a system with Hadoop, HBase, and Spark installed and running. Follow these steps:

1. Open the HBase shell:

```
$ hbase shell
```

2. From the `hbase` shell prompt, create a table named `people` with a single-column family `cf1` (using the default storage options):

Click here to view code image

```
hbase> create 'people', 'cf1'
```

3. Create several cells in two records in the table by using the `put` method:

Click here to view code image

```
hbase> put 'people', 'userid1', 'cf1:fname', 'John'
hbase> put 'people', 'userid1', 'cf1:lname', 'Doe'
hbase> put 'people', 'userid1', 'cf1:age', '41' hbase> put 'people',
'userid2', 'cf1:fname', 'Jeffrey'
hbase> put 'people', 'userid2', 'cf1:lname', 'Aven'
hbase> put 'people', 'userid2', 'cf1:age', '48'
hbase> put 'people', 'userid2', 'cf1:city', 'Hayward'
```

4. View the data in the table by using the `scan` method, as follows:

Click here to view code image

```
hbase> scan 'people'
ROW COLUMN+CELL userid1  column=cf1:age, timestamp=1461296454933,
value=41 ...
```

5. Open another terminal session and launch `pyspark` by using the arguments shown here:

```
$ pyspark --master local
```

   You can instead use YARN Client mode if you have a YARN cluster available to you.

6. Read the data from the `people` table by using `happybase` and create a Spark RDD:

Click here to view code image

```
import happybase
```

```
connection = happybase.Connection('localhost')
table = connection.table('people')
hbaserdd = sc.parallelize(table.scan())
hbaserdd.collect()
```

The output should resemble the following:

Click here to view code image

```
[('userid1', {'cf1:age': '41', 'cf1:lname': 'Doe', 'cf1:fname':
'John'}),
('userid2', {'cf1:age': '48', 'cf1:lname': 'Aven', 'cf1:fname':
'Jeffrey',
'cf1:city': 'Hayward'})]
```

7. Within your `pyspark` shell, create a new parallelized collection of users and save the contents of the Spark RDD to the `people` table in HBase:

Click here to view code image

```
newpeople = sc.parallelize([('userid3', 'cf1:fname', 'NewUser')])
for person in newpeople.collect():
    table.put(person[0], {person[1] : person[2]})
```

8. In your `hbase` shell, run the `scan` method again to confirm that the new user from the Spark RDD in step 7 exists in the HBase `people` table:

Click here to view code image

```
hbase> scan 'people' ROW COLUMN+CELL userid1 column=cf1:age,
timestamp=1461296454933, value=41 ... userid3 column=cf1:fname,
timestamp=146..., value=NewUser
```

Although this book is based on Python, there are other Spark HBase connector projects designed for the Scala API, such as `spark-hbase-connector`, at https://github.com/nerdammer/spark-hbase-connector. If you are using Spark with HBase, be sure to look at the available projects for Spark HBase connectivity.

Using Spark with Cassandra

Another notable NoSQL project is Apache Cassandra, initially developed at Facebook and later released as an open source project under the Apache software licensing scheme.

Introduction to Cassandra

Cassandra is similar to HBase in its application of the core NoSQL principles, such as not requiring a predefined schema (although Cassandra lets you define one) and not having referential integrity. However, there are differences in its physical implementation, predominantly in the fact that HBase has many Hadoop ecosystem dependencies, such as HDFS, ZooKeeper, and more, whereas Cassandra is more monolithic in its implementation, having fewer external dependencies. They also have differences in their cluster architecture: Whereas HBase is a master/slave architecture, Cassandra is a symmetric architecture that uses a "gossip" protocol to pass messages and govern cluster processes. There are many other differences, including the way the systems manage consistency, but they are beyond the scope of this discussion.

Much like HBase, Cassandra is a multidimensional, distributed map. Cassandra tables, called *keyspaces*, contain row keys and column families referred to as *tables*. Columns exist within column families but are not defined at table design time. Data is located at the intersection of a row key, column family, and column key.

In addition to row keys, Cassandra also supports *primary keys*, which can also contain a *partition key* and a *clustering key* in the case of composite primary keys. These directives are for storage and distribution of data and allow fast lookups by key.

Unlike HBase, Cassandra enables, and even encourages, you to define structure (a schema) for your data and assign datatypes. Cassandra supports *collections* within a table, which are used to store nested or complex data structures such as sets, lists, and maps. Furthermore, Cassandra enables defining *secondary indexes* to expedite lookups based on non-key values.

The *Cassandra Query Language* (*CQL*) is a SQL-like language for interacting with Cassandra. CQL supports the full set of DDL and DML operations for creating, reading, updating, and deleting objects in Cassandra. Because CQL is a SQL-like language, it supports ODBC and JDBC interfaces, enabling access from common SQL and visualization utilities. CQL is also available from an interactive shell environment, `cqlsh`.

Listing 6.35 demonstrates creating a keyspace and table in Cassandra by using the `cqlsh` utility.

Listing 6.35 **Creating a Keyspace and Table in Cassandra**

Click here to view code image

```
cqlsh> CREATE KEYSPACE mykeyspace WITH REPLICATION = { 'class' :
'SimpleStrategy',
       'replication_factor' : 1 };
cqlsh> USE mykeyspace;
cqlsh:mykeyspace> CREATE TABLE users (
              user_id int PRIMARY KEY,
          fname text,
          lname text
          );
cqlsh:mykeyspace> INSERT INTO users (user_id,  fname, lname)
          VALUES (1745, 'john', 'smith');
cqlsh:mykeyspace> INSERT INTO users (user_id,  fname, lname) VALUES
(1744, 'john', 'doe');
cqlsh:mykeyspace> INSERT INTO users (user_id,  fname, lname)
          VALUES (1746, 'jane', 'smith');
cqlsh:mykeyspace> SELECT * FROM users;
 user_id | fname | lname
---------+-------+-------
    1745 |  john | smith
    1744 |  john |   doe
    1746 |  jane | smith
```

This should look very familiar to you if your background includes relational databases such as SQL Server, Oracle, or Teradata.

Cassandra and Spark

Because the Cassandra and Spark movements are closely linked in their ties back to the Big Data/open source software community, there are several projects and libraries available to enable read/write access to Cassandra from Spark programs. The following are some of the projects providing this support:

- https://github.com/datastax/spark-cassandra-connector
- http://tuplejump.github.io/calliope/pyspark.html
- https://github.com/TargetHolding/pyspark-cassandra
- https://github.com/anguenot/pyspark-cassandra

Many of the available projects have been built and provisioned as Spark packages, available at https://spark-packages.org/.

DataStax Enterprise, which is a commercial Cassandra offering by DataStax, also ships with Spark and YARN as well.

This section uses the `pyspark-cassandra` package here, but you are encouraged to investigate all the connectivity options available—or write your own!

With many projects, classes, scripts, examples, or artifacts in the open source world, you will often find system, library, or class dependencies that you need to satisfy. Resourcefulness is a necessity when working with open source software.

For the following examples, run the `pyspark` command provided in Listing 6.36 first and note the `conf` option required to configure the Cassandra connection.

Listing 6.36 **Using the `pyspark-cassandra` Package**

Click here to view code image

```
pyspark --master local \
--packages anguenot:pyspark-cassandra:0.6.0 \
--conf spark.cassandra.connection.host=127.0.0.1
```

Listing 6.37 shows how to load the contents of the `users` table created in Listing 6.35 into an RDD.

Listing 6.37 **Reading Cassandra Data into a Spark RDD**

Click here to view code image

```
import pyspark_cassandra
spark.createDataFrame(sc.cassandraTable("mykeyspace", "users") \
    .collect()).show()
# returns:
# +-----+-------+-----+
# |lname|user_id|fname|
# +-----+-------+-----+
# |smith|   1746| jane|
# |smith|   1745| john|
# |  doe|   1744| john|
# +-----+-------+-----+
# (3 rows)
```

Listing 6.38 demonstrates writing Spark data out to a Cassandra table.

Listing 6.38 **Updating Data in a Cassandra Table Using Spark**

Click here to view code image

```
import pyspark_cassandra
rdd = sc.parallelize([{ "user_id": 1747, "fname": "Jeffrey", "lname":
"Aven" }])
rdd.saveToCassandra( "mykeyspace", "users", )
```

Running a `SELECT * FROM users` command in `cqlsh`, you can see the results of the `INSERT` from Listing 6.38 in Listing 6.39.

Listing 6.39 **Cassandra `INSERT` Results**

Click here to view code image

```
cqlsh> USE mykeyspace;
cqlsh:mykeyspace> SELECT * FROM users;
 user_id | fname   | lname
---------+---------+-------
    1745 |    john | smith
    1747 | Jeffrey |  Aven
    1744 |    john |   doe
    1746 |    jane | smith
(4 rows)
```

Using Spark with DynamoDB

DynamoDB is the AWS NoSQL PaaS offering. DynamoDB’s data model comprises *tables* containing *items*, each of which contains one or more *attributes*. Like a Cassandra table, a DynamoDB table has a primary key used for storage and fast retrieval. DynamoDB also supports secondary indexes. DynamoDB is a key/value store and a document store, as objects can be treated as documents.

Because DynamoDB originated as a web service, it has rich integration with many other language bindings and software development kits. You can implement DDL and DML statements by using Dynamo’s API endpoints and JSON-based DSL.

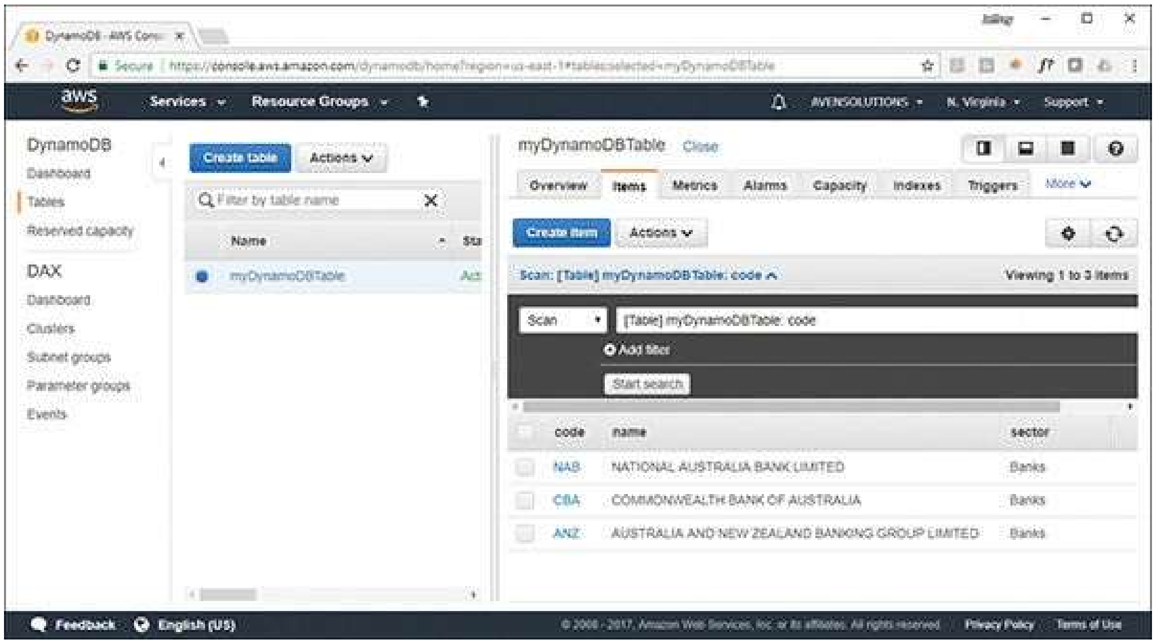
As with HBase, there are several ways to read and write data to and from DynamoDB using Spark. The simplest and most failsafe method for accessing DynamoDB from Spark using the Python API is to use the `boto3` Python library, which is designed to interact with AWS services. To use `boto3`, you need to install the package using `pip` or `easy_install`, as shown here:

```
$ sudo pip install boto3
```

You also need your AWS API credentials to connect to AWS.

Although other approaches using Spark packages or the Scala API may provide greater scalability, `boto3` always works for connecting to AWS services using Python.

Consider the DynamoDB table shown in Figure 6.14, which contains information about stocks.



code	name	sector
NAB	NATIONAL AUSTRALIA BANK LIMITED	Banks
CBA	COMMONWEALTH BANK OF AUSTRALIA	Banks
ANZ	AUSTRALIA AND NEW ZEALAND BANKING GROUP LIMITED	Banks

Figure 6.14 DynamoDB table.

Listing 6.40 demonstrates how to load the items from this DynamoDB table into a Spark RDD.

Listing 6.40 **Accessing Amazon DynamoDB from Spark**

Click here to view code image

```
import boto3
from pyspark.sql.types import *
myschema = StructType([ \
           StructField("code", StringType(), True), \
           StructField("name", StringType(), True), \
           StructField("sector", StringType(), True) \
           ])
client = boto3.client('dynamodb','us-east-1') dynamodata =
sc.parallelize(client.scan(TableName='myDynamoDBTable')['Items'])
dynamordd = dynamodata.map(lambda x: (x['code']['S'], x['name']['S'],
x['sector']['S'])).collect()
spark.createDataFrame(dynamordd, myschema).show()
# returns:
# +----+--------------------+------+
# |code|                name|sector|
# +----+--------------------+------+
# | NAB|NATIONAL AUSTRALI...| Banks|
# | CBA|COMMONWEALTH BANK...| Banks|
# | ANZ|AUSTRALIA AND NEW...| Banks|
# +----+--------------------+------+
```

Other NoSQL Platforms

In addition to the HBase, Cassandra, and DynamoDB projects, there are countless other NoSQL platforms, including document stores such as MongoDB and CouchDB, key/value stores such as Couchbase and Riak, and memory-centric key/value stores such as Memcached and Redis. There are also full text search and indexing platforms adapted to become general-purpose NoSQL platforms. These include Apache Solr and Elasticsearch, which are both based on the Lucene search engine processing project.

Many of the NoSQL platforms have available connectors or libraries that enable them to read and write RDD data in Spark. Check the project or vendor’s website or GitHub for your selected NoSQL platform’s integration. If an integration does not exist, you can always develop your own.

Summary

This chapter focuses on some of the important extensions to Spark for data manipulation and access: SQL and NoSQL.

Spark SQL is one of the most popular extensions to Spark. Spark SQL enables interactive queries, supporting business intelligence and visualization tools and making Spark accessible to a much wider audience of analysts. Spark SQL provides access to the powerful Spark runtime distributed processing framework, using SQL interfaces and a relational database-like programming approach. Spark SQL introduces many optimizations aimed specifically at relational-type access patterns using SQL. These optimizations include columnar storage, maintaining column- and partition-level statistics, and Partial DAG execution, which allows DAGs to change during processing based on statistics and skew observed in the data. Spark SQL also introduces the DataFrame, a structured, columnar abstraction of the Spark RDD. The DataFrame API enables many features and functions familiar to most SQL developers, analysts, enthusiasts, and general users. Spark SQL is evolving rapidly, and new and interesting functions and capabilities are added with every minor release. With its familiar programming interface, Spark SQL opens up a world of possibilities for a much wider community of analysts.

NoSQL databases have become a viable complement and alternative to traditional SQL systems, offering Internet-scale storage capabilities and query boundaries, as well as fast read and write access to support distributed device and mobile application interactions. NoSQL concepts and implementations have emerged in parallel with Spark, as these concepts both emanated from early Google and Yahoo! work. This chapter covers some fundamental NoSQL concepts and looks at some practical applications of key/value and document stores—Apache HBase, Apache Cassandra, and Amazon DynamoDB—to demonstrate how Spark can interact with NoSQL platforms as both a consumer and provider of data.