4
Spark DataFrame and Dataset

In the previous chapter, we learned about RDD concepts and APIs. In this chapter, we will explore DataFrame APIs, which are abstractions over RDDs, and also discuss the dataset APIs that come with Spark 2.0 to provide various optimizations over DataFrames.

The following topics will be covered in this chapter:

- DataFrames
- Datasets

DataFrames

As we already mentioned, DataFrame APIs are abstractions of RDD APIs. DataFrames are distributed collections of data that are organized in the form of rows and columns. In other words, DataFrames provide APIs to efficiently process structured data that's available in different sources. The sources could be an RDD, different types of files in a filesystem, any RDBMS, or Hive tables.

The features of DataFrames are as follows:

- DataFrames can process data that's available in different formats, such as CSV, AVRO, and JSON, or stored in any storage media, such as Hive, HDFS, and RDBMS
- DataFrames can process data volumes from kilobytes to petabytes
- Use the Spark-SQL query optimizer to process data in a distributed and optimized manner
- Support for APIs in multiple languages, including Java, Scala, Python, and R

Creating DataFrames

To start with, we need a **Spark session object**, which will be used to convert RDDs into DataFrames, or to load data directly from a file into a DataFrame.

We are using the sales dataset in this chapter. You can get the dataset file, along with the code files for this chapter, from the following link: `https://github.com/PacktPublishing/Apache-Spark-Quick-Start-Guide`:

```
//Scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Spark DataFrame
example").config("spark.some.config.option", "value").getOrCreate()
// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._

//Java
import org.apache.spark.sql.SparkSession;
SparkSession spark = SparkSession.builder().appName("Java Spark DataFrame
example").config("spark.some.config.option", "value").getOrCreate();

#Python
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Python Spark DataFrame
example").config("spark.some.config.option", "value").getOrCreate()
```

Source: `https://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sparksession`.

Once the spark session has been created in the language of your choice, you can either convert an RDD into a DataFrame or load data from any file storage in to a DataFrame:

```
//Scala
val sales_df = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt")
// Displays the content of the DataFrame to stdout
sales_df.show()

//Java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
Dataset<Row> df_sales = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt");
// Displays the content of the DataFrame to stdout
sales_df.show()
```

```
#Python
sales_df = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt")
# Displays the content of the DataFrame to stdout
sales_df.show()
```

For files in HDFS and S3, the filepath format will have `hdfs://` or `S3://` instead of `file://`.

If files do not have header information in them, you can skip the (`header`, `true`) option.

Data sources

Spark SQL allows users to query a wide variety of data sources. These sources could be files, such as **Java Database Connectivity** (**JDBC**).

There are a couple of ways to load data. Let's take a look at both methods:

- Load data from `parquet`:

```
//Scala
val sales_df = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt")

sales_df.write.parquet("sales.parquet")

val parquet_sales_DF = spark.read.parquet("sales.parquet")
parquet_sales_DF.createOrReplaceTempView("parquetSales")

val ipDF = spark.sql("SELECT ip FROM parquetSales WHERE id BETWEEN
10 AND 19")
ipDF.map(attributes => "IPS: " + attributes(0)).show()

//Java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
Dataset<Row> df_sales = spark.read.option("sep",
"\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt");

// Write data to parquet file
df_sales.write().parquet("sales.parquet");

// Parquet preserve the schema of file
```

```
Dataset<Row> parquetSalesDF =
spark.read().parquet("sales.parquet");

parquetSalesDF.createOrReplaceTempView("parquetSales");
Dataset<Row> ipDF = spark.sql("SELECT ip FROM parquetSales WHERE id
BETWEEN 10 AND 19");
Dataset<String> ipDS = ipDF.map(
    (MapFunction<Row, String>) row -> "IP: " + row.getString(0),
    Encoders.STRING());
ipDS.show();

#Python
sales_df = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt")

sales_df.write.parquet("sales.parquet")

parquetSales = spark.read.parquet("sales.parquet")

parquetSales.createOrReplaceTempView("parquetSales")
ip = spark.sql("SELECT ip FROM parquetsales WHERE id >= 10 AND id
<= 19")
ip.show()
```

- Load data from JSON:

```
//Scala
val sales_df = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt")

sales_df.write.json("sales.json")

val json_sales_DF = spark.read.json("sales.json")
json_sales_DF.createOrReplaceTempView("jsonSales")

var ipDF = spark.sql("SELECT ip FROM jsonSales WHERE id BETWEEN 10
AND 19")
ipDF.map(attributes => "IPS: " + attributes(0)).show()
```

DataFrame operations and associated functions

DataFrames support *untyped transformations* with the following operations:

- `printSchema`: This prints out the mapping for a Spark DataFrame in a tree structure. The following code will give you a clear idea of how this operation works:

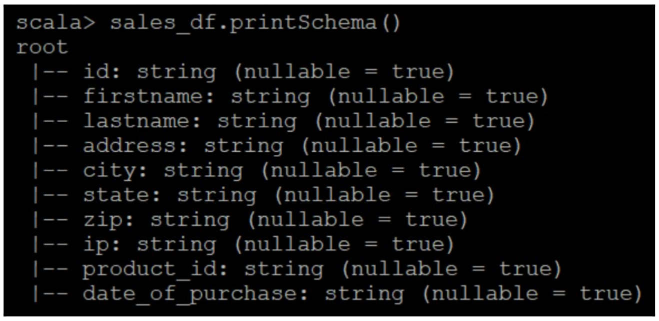
```
//Scala
import spark.implicits._
// Print the schema in a tree format
sales_df.printSchema()

//Java
import static org.apache.spark.sql.functions.col;

// Print the schema in a tree format
sales_df.printSchema();

#Python
# Print the schema in a tree format
sales_df.printSchema()
```

The output you get should look like this:



```
scala> sales_df.printSchema()
root
 |-- id: string (nullable = true)
 |-- firstname: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- address: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- zip: string (nullable = true)
 |-- ip: string (nullable = true)
 |-- product_id: string (nullable = true)
 |-- date_of_purchase: string (nullable = true)
```

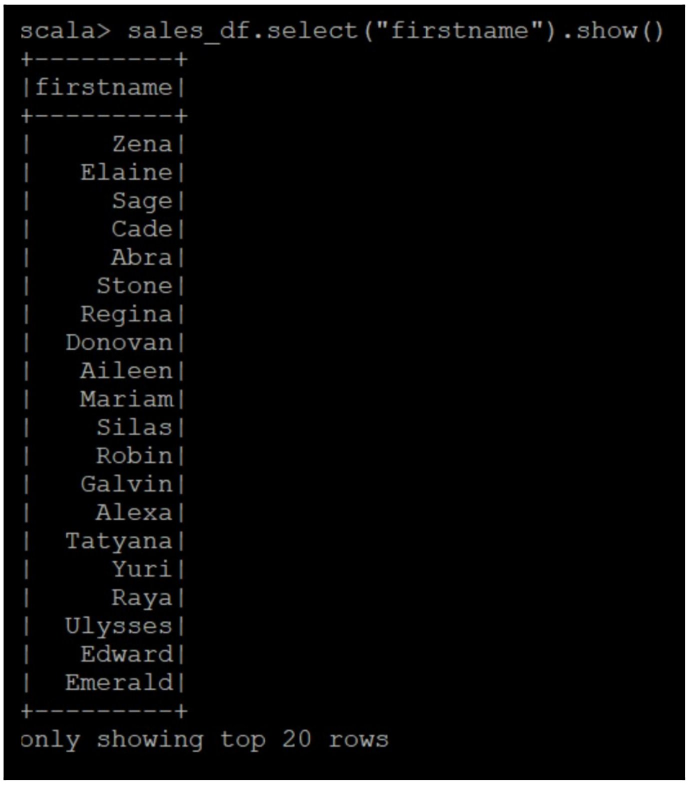
- `select`: This allows you to select a set of columns from a DataFrame. The following code will give you a clear idea of how this operation works:

```
//Scala
import spark.implicits._
sales_df.select("firstname").show()

//Java
import static org.apache.spark.sql.functions.col;
sales_df.select("firstname").show()
```

```
#Python
sales_df.select("firstname").show()
```

The output you get should look like this:



```
scala> sales_df.select("firstname").show()
+---------+
|firstname|
+---------+
|     Zena|
|   Elaine|
|     Sage|
|     Cade|
|     Abra|
|    Stone|
|   Regina|
|  Donovan|
|   Aileen|
|   Mariam|
|    Silas|
|    Robin|
|   Galvin|
|    Alexa|
|  Tatyana|
|     Yuri|
|     Raya|
|  Ulysses|
|   Edward|
|  Emerald|
+---------+
only showing top 20 rows
```

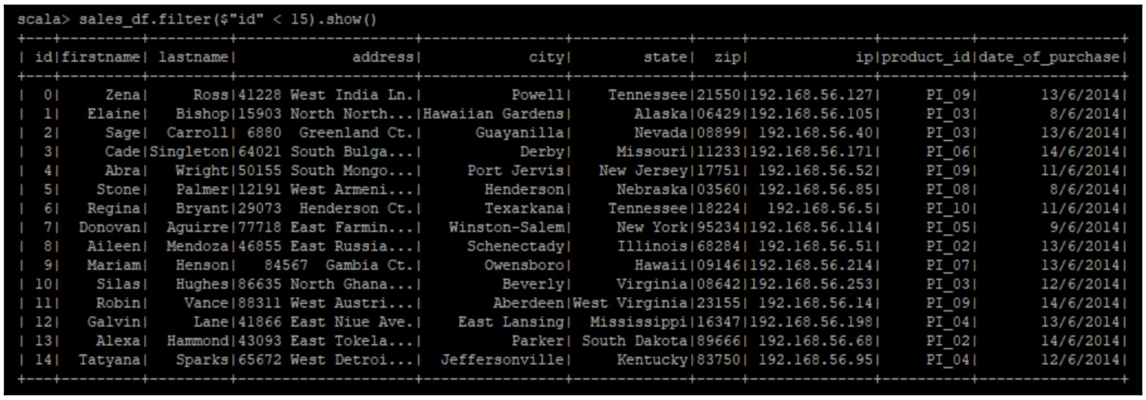
- `filter`: This allows you to filter rows from a DataFrame based on certain conditions. The following code will give you a clear idea of how this operation works:

```
//Scala
import spark.implicits._
sales_df.filter($"id" < 50).show()

//Java
import static org.apache.spark.sql.functions.col;
sales_df.filter(col("id").gt(9990)).show();

#Python
sales_df.filter(sales_df['id'] < 50).show()
```

The output you get should look like this:



```
scala> sales_df.filter($"id" < 15).show()
+---+---------+---------+--------------------+----------------+-------------+-----+--------------+----------+----------------+
| id|firstname| lastname|             address|            city|        state|  zip|            ip|product_id|date_of_purchase|
+---+---------+---------+--------------------+----------------+-------------+-----+--------------+----------+----------------+
|  0|     Zena|     Ross|41228 West India Ln.|          Powell|    Tennessee|21550|192.168.56.127|     PI_09|       13/6/2014|
|  1|   Elaine|   Bishop|15903 North North...|Hawaiian Gardens|       Alaska|06429|192.168.56.105|     PI_03|        8/6/2014|
|  2|     Sage|  Carroll| 6880  Greenland Ct.|      Guayanilla|       Nevada|08899| 192.168.56.40|     PI_03|       13/6/2014|
|  3|     Cade|Singleton|64021 South Bulga...|           Derby|     Missouri|11233|192.168.56.171|     PI_06|       14/6/2014|
|  4|     Abra|   Wright|50155 South Mongo...|     Port Jervis|   New Jersey|17751| 192.168.56.52|     PI_09|       11/6/2014|
|  5|    Stone|   Palmer|12191 West Armeni...|       Henderson|     Nebraska|03560| 192.168.56.85|     PI_08|        8/6/2014|
|  6|   Regina|   Bryant|29073  Henderson Ct.|       Texarkana|    Tennessee|18224|  192.168.56.5|     PI_10|       11/6/2014|
|  7|  Donovan|  Aguirre|77718 East Farmin...|   Winston-Salem|     New York|95234|192.168.56.114|     PI_05|        9/6/2014|
|  8|   Aileen|  Mendoza|46855 East Russia...|     Schenectady|     Illinois|68284| 192.168.56.51|     PI_02|       13/6/2014|
|  9|   Mariam|   Henson|   84567  Gambia Ct.|       Owensboro|       Hawaii|09146|192.168.56.214|     PI_07|       13/6/2014|
| 10|    Silas|   Hughes|86635 North Ghana...|         Beverly|     Virginia|08642|192.168.56.253|     PI_03|       12/6/2014|
| 11|    Robin|    Vance|88311 West Austri...|        Aberdeen|West Virginia|23155| 192.168.56.14|     PI_09|       14/6/2014|
| 12|   Galvin|     Lane|41866 East Niue Ave.|    East Lansing|  Mississippi|16347|192.168.56.198|     PI_04|       13/6/2014|
| 13|    Alexa|  Hammond|43093 East Tokela...|          Parker| South Dakota|89666| 192.168.56.68|     PI_02|       14/6/2014|
| 14|  Tatyana|   Sparks|65672 West Detroi...|  Jeffersonville|     Kentucky|83750| 192.168.56.95|     PI_04|       12/6/2014|
+---+---------+---------+--------------------+----------------+-------------+-----+--------------+----------+----------------+
```

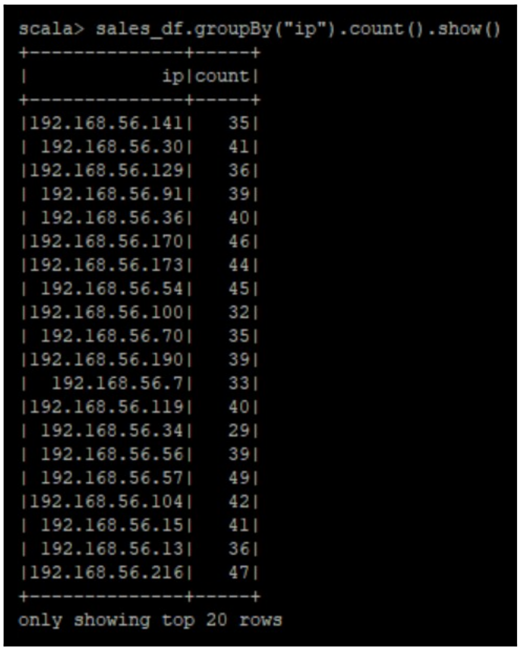
- `groupBy`: This allows you to group rows in a DataFrame based on a set of columns, and apply aggregated functions such as `count()`, `avg()` , and so on on the grouped dataset. The following code will give you a clear idea of how this operation works:

```
//Scala
import spark.implicits._
sales_df.groupBy("ip").count().show()

//Java
import static org.apache.spark.sql.functions.col;
sales_df.groupBy("ip").count().show();

#Python
sales_df.groupBy("ip").count().show()
```

The output you get should look like this:



```
scala> sales_df.groupBy("ip").count().show()
+--------------+-----+
|            ip|count|
+--------------+-----+
|192.168.56.141|   35|
| 192.168.56.30|   41|
|192.168.56.129|   36|
| 192.168.56.91|   39|
| 192.168.56.36|   40|
|192.168.56.170|   46|
|192.168.56.173|   44|
| 192.168.56.54|   45|
|192.168.56.100|   32|
| 192.168.56.70|   35|
|192.168.56.190|   39|
|  192.168.56.7|   33|
|192.168.56.119|   40|
| 192.168.56.34|   29|
| 192.168.56.56|   39|
| 192.168.56.57|   49|
|192.168.56.104|   42|
| 192.168.56.15|   41|
| 192.168.56.13|   36|
|192.168.56.216|   47|
+--------------+-----+
only showing top 20 rows
```

A complete list of DataFrame functions that can be used with these operations is available here:

`https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$.`

Running SQL on DataFrames

Other than DataFrame operations and functions, DataFrames also allow you to run SQL directly on data. For this, all we need to do is create temporary views on DataFrames. These views are categorized as local or global views.

Temporary views on DataFrames

This feature enables developers to run SQL queries in a program, and get the result as a DataFrame:

```
//Scala
sales_df.createOrReplaceTempView("sales")
```

```
val sqlDF = spark.sql("SELECT * FROM sales")
sqlDF.show()

//Java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
sales_df.createOrReplaceTempView("sales");
Dataset<Row> sqlDF = spark.sql("SELECT * FROM sales");
sqlDF.show();

#Python
sales_df.createOrReplaceTempView("sales")
sqlDF = spark.sql("SELECT * FROM sales")
sqlDF.show()
```

Global temporary views on DataFrames

Temporary views only last for the session in which they are created. If we want to have views available across various sessions, we need to create **Global Temporary Views**. The view definition is stored in the default database, `global_temp`. Once a view is created, we need to use the fully qualified name to access it in a query:

```
//Scala
sales_df.createGlobalTempView("sales")
// Global temporary view is tied to a system database `global_temp`
spark.sql("SELECT * FROM global_temp.sales").show()
spark.newSession().sql("SELECT * FROM global_temp.sales").show()

//Java
sales_df.createGlobalTempView("sales");
spark.sql("SELECT * FROM global_temp.sales").show();
spark.newSession().sql("SELECT * FROM global_temp.sales").show();

#Python
sales_df.createGlobalTempView("sales")
# Global temporary view is tied to a system database `global_temp`
spark.sql("SELECT * FROM global_temp.sales").show()
spark.newSession().sql("SELECT * FROM global_temp.sales").show()
```

Datasets

Datasets are strongly typed collections of objects. These objects are usually domain-specific and can be transformed in parallel using relational or functional operations.

These operations are further categorized into actions and transformations. Transformations are functions that generate new datasets, while actions compute datasets and return the transformed results. Transformation functions include Map, FlatMap, Filter, Select, and Aggregate, while Action functions include `count`, `show`, and `save` to any filesystem.

The following instructions will help you create a dataset from a CSV file:

1. Initialize `SparkSession`:

```
//Scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Spark DataSet
example").config("spark.config.option", "value").getOrCreate()
// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._

//Java
import org.apache.spark.sql.SparkSession;
SparkSession spark = SparkSession.builder().appName("Java Spark
DataFrame example").config("spark.config.option",
"value").getOrCreate();

#Python
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Python Spark DataFrame
example").config("spark.config.option", "value").getOrCreate()
```

2. Define an encoder for this CSV:

```
case class Sales (id: Int, firstname: String,lastname:
String,address: String,city: String,state: String,zip: String,ip:
String,product_id: String,date_of_purchase: String)
```

3. Load the dataset from the CSV with type sales:

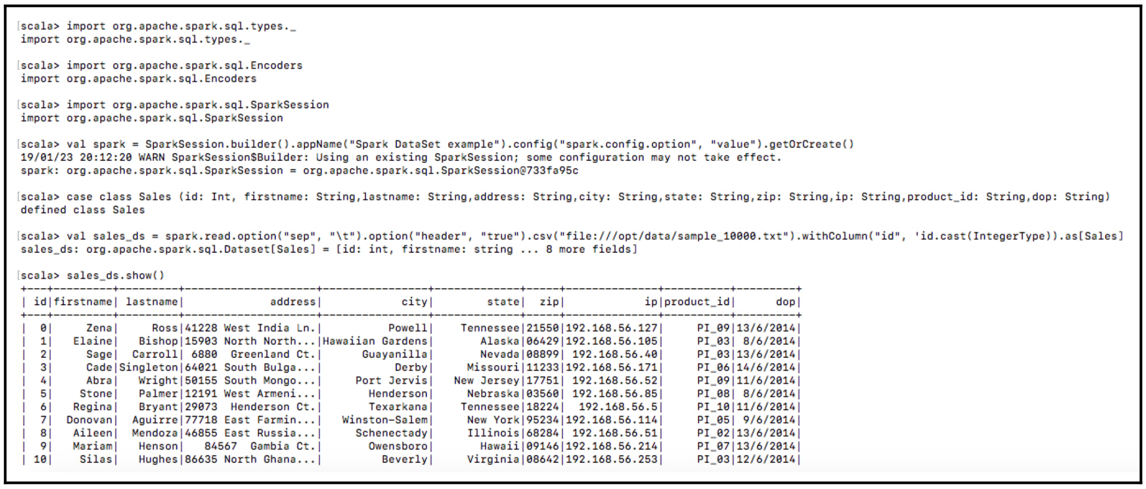
```
//Scala
import org.apache.spark.sql.types._
import org.apache.spark.sql.Encoders
val sales_ds = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt").withColumn("
id", 'id.cast(IntegerType)).as[Sales]
```

```
// Displays the content of the Dataset to stdout
sales_ds.show()

//Java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
Dataset<Row> sales_ds = spark.read.option("sep",
"\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt");
// Displays the content of the Dataset to stdout
sales_ds.show()

#Python
sales_ds = spark.read.option("sep", "\t").option("header",
"true").csv("file:///opt/data/sales/sample_10000.txt")
# Displays the content of the Dataset to stdout
sales_ds.show()
```

The following image shows how we can create a **Dataset** of **Sales** CSV data, along with a **Sales** encoder defined for a dataset:



```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._

scala> import org.apache.spark.sql.Encoders
import org.apache.spark.sql.Encoders

scala> import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SparkSession

scala> val spark = SparkSession.builder().appName("Spark DataSet example").config("spark.config.option", "value").getOrCreate()
19/01/23 20:12:20 WARN SparkSession$Builder: Using an existing SparkSession; some configuration may not take effect.
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@733fa95c

scala> case class Sales (id: Int, firstname: String,lastname: String,address: String,city: String,state: String,zip: String,ip: String,product_id: String,dop: String)
defined class Sales

scala> val sales_ds = spark.read.option("sep", "\t").option("header", "true").csv("file:///opt/data/sample_10000.txt").withColumn("id", 'id.cast(IntegerType)).as[Sales]
sales_ds: org.apache.spark.sql.Dataset[Sales] = [id: int, firstname: string ... 8 more fields]

scala> sales_ds.show()
+---+---------+---------+--------------------+----------------+----------+-----+--------------+----------+---------+
| id|firstname| lastname|             address|            city|     state|  zip|            ip|product_id|      dop|
+---+---------+---------+--------------------+----------------+----------+-----+--------------+----------+---------+
|  0|     Zena|     Ross|41228 West India Ln.|          Powell| Tennessee|21550|192.168.56.127|     PI_09|13/6/2014|
|  1|   Elaine|   Bishop|15903 North North...|Hawaiian Gardens|    Alaska|06429|192.168.56.105|     PI_03| 8/6/2014|
|  2|     Sage|  Carroll| 6880  Greenland Ct.|      Guayanilla|    Nevada|08899| 192.168.56.40|     PI_03|13/6/2014|
|  3|     Cade|Singleton|64021 South Bulga...|           Derby|  Missouri|11233|192.168.56.171|     PI_06|14/6/2014|
|  4|     Abra|   Wright|50155 South Mongo...|     Port Jervis|New Jersey|17751| 192.168.56.52|     PI_09|11/6/2014|
|  5|    Stone|   Palmer|12191 West Armeni...|       Henderson|  Nebraska|03560| 192.168.56.85|     PI_08| 8/6/2014|
|  6|   Regina|   Bryant|29073  Henderson Ct.|       Texarkana| Tennessee|18224|  192.168.56.5|     PI_10|11/6/2014|
|  7|  Donovan|  Aguirre|77718 East Farmin...|   Winston-Salem|  New York|95234|192.168.56.114|     PI_05| 9/6/2014|
|  8|   Aileen|  Mendoza|46855 East Russia...|     Schenectady|  Illinois|68284| 192.168.56.51|     PI_02|13/6/2014|
|  9|   Mariam|   Henson|   84567  Gambia Ct.|       Owensboro|    Hawaii|09146|192.168.56.214|     PI_07|13/6/2014|
| 10|    Silas|   Hughes|86635 North Ghana...|         Beverly|  Virginia|08642|192.168.56.253|     PI_03|12/6/2014|
```

Important differences in a dataset compared to DataFrames are as follows:

- Defining a case class to define types of columns in CSV
- If the interpreter takes up a different type by inference, we need to cast to the exact type using the `withColumn` property
- The output is a dataset of `Type Sales`, not a DataFrame

We need to check the correctness of the data, as it is possible that the actual data does not match with the `Type` defined. There are three options to deal with this situation:

1. `Permissive`: This is the default `mode` in which, if the data type is not matched with the schema type, the data fields are replaced with null:

```
 val sales_ds = spark.read.option("sep", "\t").option("header",
"true").option("mode",
"PERMISSIVE").csv("file:///opt/data/sales/sample_10000.txt").withCo
lumn("id", 'id.cast(IntegerType)).as[Sales]
```

2. `DROPMALFORMED`: As the name suggests, this `mode` will drop records where the parser finds a mismatch between the data type and schema type:

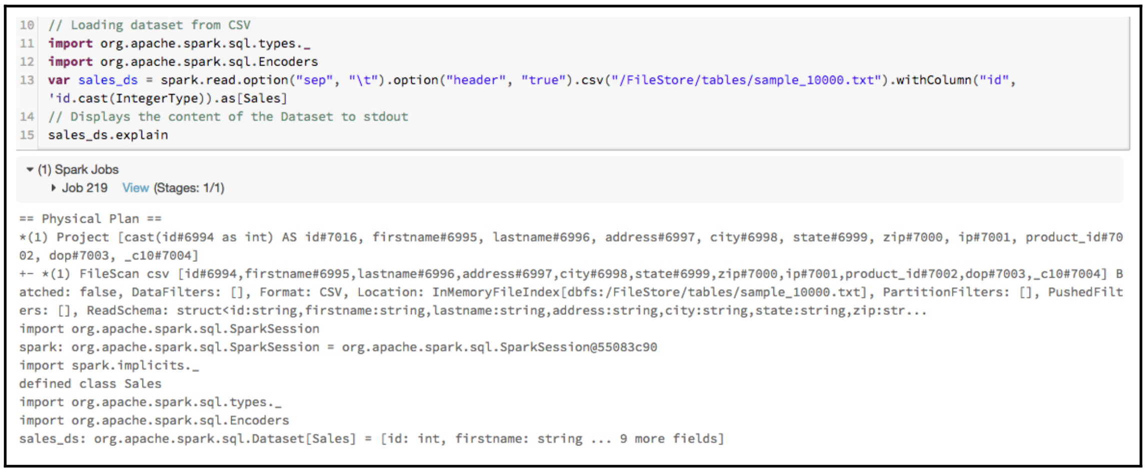
```
val sales_ds = spark.read.option("sep", "\t").option("header",
"true").option("mode",
"DROPMALFORMED").csv("file:///opt/data/sales/sample_10000.txt").wit
hColumn("id", 'id.cast(IntegerType)).as[Sales]
```

3. `FAILFAST`: This `mode` will abort further processing on the first mismatch between data type and schema type:

```
val sales_ds = spark.read.option("sep", "\t").option("header",
"true").option("mode",
"FAILFAST").csv("file:///opt/data/sales/sample_10000.txt").withColu
mn("id", 'id.cast(IntegerType)).as[Sales]
```

Datasets work on the concept of *lazy evaluation,* which means for every transformation, a new dataset definition is created, but no execution happens at the backend. In this case, it only creates a logical plan that describes the computation flow required to execute the transformation. The actual evaluation happens once we have an action being called on the dataset. With an action, the Spark query optimizer optimizes the logical plan and creates a physical plan of execution. This physical plan then computes the datasets in a parallel and distributed way. The `explain` function is used to check for the logical and optimized physical plan.

The following image shows the explain plan for the **sales** dataset:



```
// Loading dataset from CSV
import org.apache.spark.sql.types._
import org.apache.spark.sql.Encoders
var sales_ds = spark.read.option("sep", "\t").option("header", "true").csv("/FileStore/tables/sample_10000.txt").withColumn("id",
'id.cast(IntegerType)).as[Sales]
// Displays the content of the Dataset to stdout
sales_ds.explain
```

```
== Physical Plan ==
*(1) Project [cast(id#6994 as int) AS id#7016, firstname#6995, lastname#6996, address#6997, city#6998, state#6999, zip#7000, ip#7001, product_id#70
02, dop#7003, _c10#7004]
+- *(1) FileScan csv [id#6994,firstname#6995,lastname#6996,address#6997,city#6998,state#6999,zip#7000,ip#7001,product_id#7002,dop#7003,_c10#7004] B
atched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[dbfs:/FileStore/tables/sample_10000.txt], PartitionFilters: [], PushedFilt
ers: [], ReadSchema: struct<id:string,firstname:string,lastname:string,address:string,city:string,state:string,zip:str...
import org.apache.spark.sql.SparkSession
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@55083c90
import spark.implicits._
defined class Sales
import org.apache.spark.sql.types._
import org.apache.spark.sql.Encoders
sales_ds: org.apache.spark.sql.Dataset[Sales] = [id: int, firstname: string ... 9 more fields]
```

Encoders

Encoders are required to map domain-specific objects of type `T` to Spark's type system or internal Spark SQL representation. An `Encoder` of type `T` is a trait represented by `Encoder[T]`.

Encoders are available with every Spark session, and you can explicitly import them with spark `implicits` as `import spark.implicits._`.

For example, given an `Employee` class with the fields name (`String`) and salary (`int`), an encoder is used as an indicator to serialize the `Employee` object to binary form. This binary structure provides the following advantages:

- Occupies less memory
- Data is stored in columnar format for efficient processing

Let's take a look at the major encoder features:

- **Fast serialization:** Encoders are used for runtime code with custom bytecode generation for serialization and deserialization. These are significantly faster than Java and Kryo serializers. Along with faster serialization, encoders also provide significant data compression, which helps with better network transfers. Encoders produce data in Tungsten binary format, which also allows different operations in place, rather than materializing data to an object.
- **Support for semi-structured data:** Encoders allow Spark to process complex JSON with type-safe Scala and Java.

Let's look at an example. Consider the following `sales` dataset in JSON structure, or use the previous commands to write the JSON file from a CSV file:

```
{"id": "1", "firstname": "Elaine", "lastname": "Bishop","address": "15903
North North Adams Blvd.", "city": "Hawaiian Gardens", "state":
"Alaska","zip": "06429", "ip": "192.168.56.105", "product_id": "PI_03",
"dop":"8/6/2018"}

{"id": "2", "firstname": "Sage", "lastname": "Carroll","address": "6880
Greenland Ct.", "city": "Guayanilla", "state": "Nevada","zip": "08899",
"ip": "192.168.56.40", "product_id": "PI_04", "dop":"13/6/2018"}
```

To convert JSON data fields into a type, we can define a `case class` with a structure and `map` input data in to the defined structure. Columns in the `case class` are mapped to keys in JSON, and types are mapped as defined in the `case class`:

```
case class Sales(id: String,firstname: String,lastname: String,address:
String,city: String,state: String,zip: String,ip: String,product_id:
String,dop: String )

val sales = sqlContext.read.json("sales.json").as[Sales]

sales.map(s => s"${s.firstname} purchased product ${s.product_id} on
${s.dop}")
```

Encoders also check the type of the expected schema with data, and give an error in the case of any type mismatch. For example, if we define a byte type in a class where the encoder finds more integers, it will complain instead of processing TBs of data with auto casting integers to byte and losing precision:

```
case class Sales(id: byte)

val sales= sqlContext.read.json("sales.json").as[Sales]
```

```
org.apache.spark.sql.AnalysisException: Cannot upcast id
from int to smallint as it may truncate
```

Encoders can also handle complex types, including arrays and maps.

Internal row

Encoders are coded as **traits** in Spark 2.0. They can be thought of as an efficient means of serialization/deserialization for Spark SQL 2.0, similar to **SerDes** in Hive:

```
trait Encoder[T] extends Serializable {
  def schema: StructType
  def clsTag: ClassTag[T]
}
```

Encoders internally convert type `T` to Spark SQL's `InternalRow` type, which is the binary row representation.

Creating custom encoders

Encoders can be created based on Java and Kryo serializers. Encoder factory objects are available in the `org.apache.spark.sql` package:

```
import org.apache.spark.sql.Encoders

// Normal Encoder
scala> Encoders.LONG
res1: org.apache.spark.sql.Encoder[Long] = class[value[0]: bigint]

// Kryo and Java Serialization Encoders
case class Sales(id: String, firstname: String, product_id: Boolean)

scala> Encoders.kryo[Sales]
res3: org.apache.spark.sql.Encoder[Sales] = class[value[0]: binary]

scala> Encoders.javaSerialization[Sales]
res5: org.apache.spark.sql.Encoder[Sales] = class[value[0]: binary]

// Scala tuple encoders
scala> Encoders.tuple(Encoders.scalaLong, Encoders.STRING,
Encoders.scalaBoolean)
res9: org.apache.spark.sql.Encoder[(Long, String, Boolean)] = class[_1[0]:
bigint, _2[0]: string, _3[0]: boolean]
```

Summary

In this chapter, we started by loading a dataset into a DataFrame, and then applying different transformations to the DataFrame. Later, we went through the latest additions of dataset APIs and encoders in Spark 2.0.

In the next chapter, we will go through Spark's architecture and its components in detail. We will also see, in detail, the flow of a Spark application once it is submitted.

6
Spark SQL

In our previous chapter, we learned about DataFrames and datasets and how we can use or write custom encoders to have type-safe operations on datasets. This chapter explains the SQL component of Spark, which helps developers working on Hive or familiar with RDBMS SQL to use a similar style in Spark.

We will be covering the following topics in this chapter:

- Spark metastore
- SQL language manual
- SQL database using **Java Database Connectivity** (**JDBC**)

Spark SQL

Spark SQL is an abstraction of data using **SchemaRDD**, which allows you to define datasets with schema and then query datasets using SQL. To start with, you just have to type `spark-sql` in the Terminal with Spark installed. This will open a Spark shell for you.

Spark metastore

To store databases, table names, and schema, Spark installs a default database, `metastore.db`, at the same location from where you started the SQL shell.

Using the Hive metastore in Spark SQL

Spark provides the flexibility to leverage the existing Hive metastore. This will allow users to access table definitions as available to Hive in Spark and to run the same HiveQL in Spark. The difference will be that queries running on Spark will be executed as per the Spark execution plan, and underlying data will be processed as per Spark execution and optimizations. These queries wont follow the MapReduce path, which is the default in Hive.

For many queries, users can see a tremendous performance gain with the Spark execution engine compared to the MapReduce engine, due to the optimized plan of execution in Spark.

Hive configuration with Spark

Hive on Spark gives Hive the capacity to use Apache as its execution motor. We will be using the following steps to configure Hive:

1. Copy `hive-site.xml` to the Spark configuration folder as follows:

```
cp $HIVE_HOME/conf/hive-site.xml $SPARK_HOME/conf/
```

2. Add the following line to `~/.bash_profile`:

```
export SPARK_CLASSPATH=$HIVE_HOME/lib/mysql-connector-java-3.1.14-
bin.jar

source ~/.bash_profile
```

3. Run the following command to access Spark SQL:

```
spark-sql
```

Check for existing databases and tables with the `Show Databases` and `Show Tables` commands. You'll find all of the databases and corresponding tables that you have in Hive.

SQL language manual

Spark SQL provides a set of **Data Definition Languages** (**DDLs**) and **Data Manipulation Languages** (**DMLs**). These are the same as, or very similar to, Hive and other basic SQL language specifications.

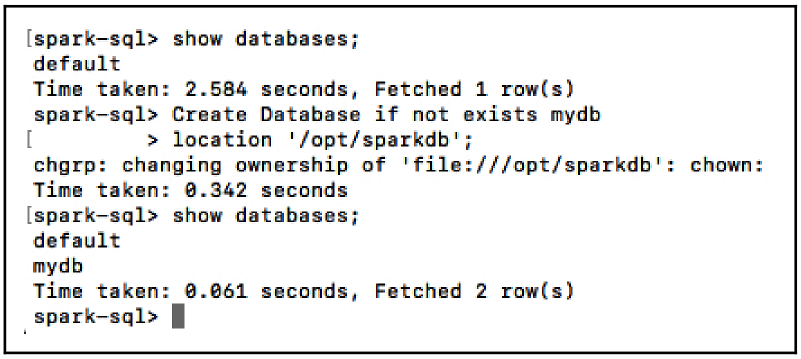
Database

In this section, we will be looking at some operations that we can perform on a database:

1. `Create Database`: We will be using the following command to create a database:

```
Create Database if not exists mydb
location '/opt/sparkdb';
```

The output following execution will be similar to this:

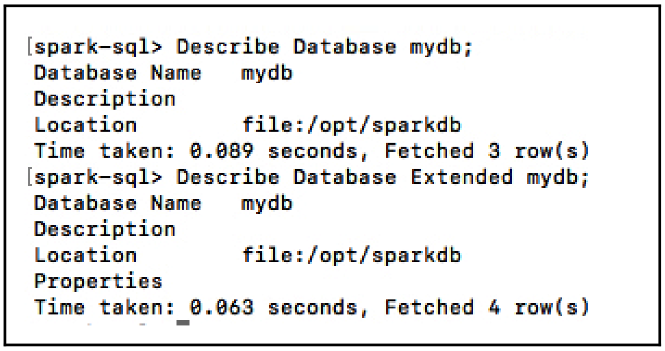


```
[spark-sql> show databases;
 default
 Time taken: 2.584 seconds, Fetched 1 row(s)
 spark-sql> Create Database if not exists mydb
[         > location '/opt/sparkdb';
 chgrp: changing ownership of 'file:///opt/sparkdb': chown:
 Time taken: 0.342 seconds
[spark-sql> show databases;
 default
 mydb
 Time taken: 0.061 seconds, Fetched 2 row(s)
 spark-sql>
```

2. `Describe Database`: We will be using the following command to describe a database:

```
Describe Database [extended] mydb;
```

The output after execution will be similar to this:



```
[spark-sql> Describe Database mydb;
 Database Name   mydb
 Description
 Location        file:/opt/sparkdb
 Time taken: 0.089 seconds, Fetched 3 row(s)
[spark-sql> Describe Database Extended mydb;
 Database Name   mydb
 Description
 Location        file:/opt/sparkdb
 Properties
 Time taken: 0.063 seconds, Fetched 4 row(s)
```

3. SHOW DATABASES: We will be using the following command to display a database:

```
SHOW DATABASES [LIKE 'pattern']
```

pattern could be any partial search string or *.

4. use mydb: The following command can be given to use a database:

```
use mydb;
```

5. DROP DATABASE: We will be using the following command to describe a database:

```
DROP DATABASE [IF EXISTS] mydb [(RESTRICT|CASCADE)]
```

- CASCADE: This will delete all underlying tables from the database
- RESTRICT: This will raise an exception if we run it on a non-empty database

Table and view

In this section, we will be looking at some operations that we can perform on a table and view:

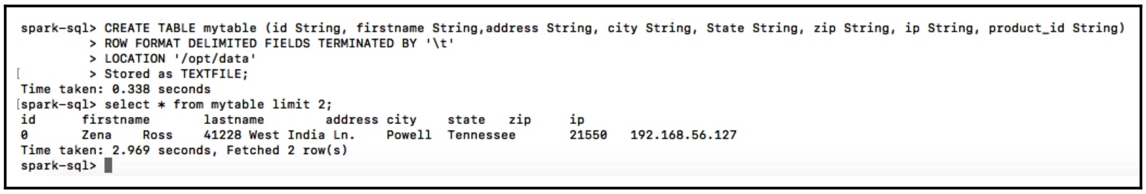
1. Create table: We will be using the following command to create a table:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [mydb.]mytable
    [(col_name1:col_type1)]
    --[PARTITIONED BY (col_name2:col_type2)]
    [ROW FORMAT row_format]
    [STORED AS file_format]
    [LOCATION path]
    [TBLPROPERTIES (key1=val1, key2=val2, ...)]
    [AS select_statement]
```

Here's an example of creating a table with actual values:

```
CREATE TABLE mytable (id String, firstname String,address String,
city String, State String, zip String, ip String, product_id
String) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LOCATION '/opt/data'
Stored as TEXTFILE;
```

You will see the following screen on execution of the previous command:



```
spark-sql> CREATE TABLE mytable (id String, firstname String,address String, city String, State String, zip String, ip String, product_id String)
         > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
         > LOCATION '/opt/data'
[        > Stored as TEXTFILE;
Time taken: 0.338 seconds
[spark-sql> select * from mytable limit 2;
id      firstname       lastname        address city    state   zip     ip
0       Zena    Ross    41228 West India Ln.    Powell  Tennessee       21550   192.168.56.127
Time taken: 2.969 seconds, Fetched 2 row(s)
spark-sql> █
```

Here are the parameters that need to be defined during table creation:

- **Datasource**: This is the file format with which this table is associated. It could be CSV, JSON, TEXT, ORC, or Parquet.
- **n**: Specifies the number of buckets if you want to create bucketed table.

2. Create view: We will be using the following command to create a `VIEW`:

```
CREATE [OR REPLACE] VIEW mydb.myview
    [(col1_name, col2_name)]
    [TBLPROPERTIES (key1=val1, ...)]
    AS select ...
```

This will create a logical view on one or more tables. The view definition will only store the corresponding query definition and, when the view is used, the underlying query will be called at runtime.

3. Describe table: We will be using the following command to `DESCRIBE` a table:

```
DESCRIBE mydb.mytable
```

Extended: `Describe Extended` will give more detailed information about the table definition.

4. Alter table or view: There are various operations under `ALTER` that we can perform. We will be taking a look at the following operations:
    - `RENAME`: We will be using the following command to rename a table or view:

      ```
      ALTER TABLE|VIEW mydb.mytable RENAME TO mydb.mytable1
      ```

    - `SET PROPERTIES`: The following command can be used to set the properties of a table or view:

      ```
      ALTER TABLE|VIEW mytable SET TBLPROPERTIES (key1=val1,
      key2=val2, ...)
      ```

    - Drop properties: The following command can be used to drop the properties of a table or view:

      ```
      ALTER TABLE|VIEW mytable UNSET TBLPROPERTIES IF EXISTS
      (key1, key2, ...)
      ```

5. `DROP TABLE`: We will be using the following command to drop a table:

   ```
   DROP TABLE mydb.mytable
   ```

6. This show table properties: We will be using the following command to show the properties of a particular table:

   ```
   SHOW TBLPROPERTIES mydb.mytable [(prop_key)]
   ```

7. `SHOW TABLES`: The command that follows is used to show tables:

   ```
   SHOW TABLES [LIKE 'pattern']
   ```

Shows all tables in the current database. Use `pattern` if you want to list only specific tables based on a pattern.

8. `TRUNACTE TABLE`: We will be using the following command to truncate a particular table:

   ```
   TRUNCATE TABLE mytable
   ```

This will delete all rows from the specified table. It does not work on view or temporary tables.

9. `SHOW CREATE TABLE`: The following command provides the create table statement for `mytable`:

   ```
   SHOW CREATE TABLE mydb.mytable
   ```

10. `SHOW COLUMNS`: We will be using the following command to display the list of columns in the specified table:

    ```
    SHOW COLUMNS (FROM | IN) mydb.mytable
    ```

11. `INSERT`: We will be using the following command to insert values into a table:

    ```
    INSERT INTO mydb.mytable select ... from mydb.mytable1
    ```

In the event that the table is divided, we must determine a particular partition of the table. We can use the following command to `INSERT` into `PARTITION` of a table:

```
INSERT INTO mydb.mytable PARTITION (part_col_name1=val1) select ... from
mydb.mytable1
```

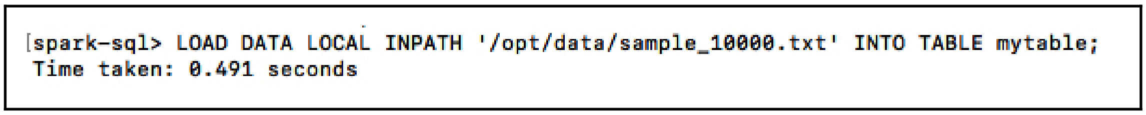
Load data

We are allowed to load data into Hive tables in three different ways. Two of the methods are DML tasks of Hive. The third is utilizing HDFS order. These three methods are explained as follows:

- **Load data from local filesystem**: We will be using the following command to load data from a local filesystem:

  ```
  LOAD DATA LOCAL INPATH 'local_path' INTO TABLE mydb.mytable
  ```

  The following screenshot shows how we can load `sample_10000.txt` from the local filesystem into a Spark table:



```
spark-sql> LOAD DATA LOCAL INPATH '/opt/data/sample_10000.txt' INTO TABLE mytable;
Time taken: 0.491 seconds
```

- **Load data from HDFS**: We will be using the following command to load data from HDFS:

```
LOAD DATA INPATH 'hdfs_path' INTO TABLE mydb.mytable
```

- **Load data into a partition of a table**: We can use the following command to load data into a partition of a table:

```
LOAD DATA [LOCAL] INPATH 'path' INTO TABLE mydb.mytable PARTITION
(part_col1_name=val1)
```

Creating UDFs

Users can define **User-Defined Functions** (**UDFs**) for custom logic in Scala or Python. The formats for UDF definition and registration are explained as follows:

- The syntax for registering a Spark SQL function as a UDF in Scala is given as follows:

```
val squared = (s: Int) => { s * s }
spark.udf.register("square", squared)
```

- The syntax for calling a Spark SQL function as a UDF in Scala is given as follows:

```
spark.range(1, 20).createOrReplaceTempView(("udf_test"))

%sql select id, square(id) as id_squared from udf_test
```

- The syntax for registering a Spark SQL function as a UDF in Python is given as follows:

```
def squared(s):
  return s * s
spark.udf.register("squaredWithPython", squared)
```

- The syntax for calling a Spark SQL function as a UDF in Python is given as follows:

```
spark.range(1, 20).registerTempTable("test")
%sql select id, squaredWithPython(id) as id_squared from test
```

SQL database using JDBC

Spark SQL also enables users to query directly from different RDBMS data sources. The results of the query are returned as a DataFrame that can be further queried with Spark SQL or joined with other datasets.

To use a JDBC connection, you need to add the JDBC driver jars for the required database in the Spark classpath.

For example, `mysql` can be connected with Spark SQL with the following commands:

```
import org.apache.spark.sql.SparkSession

object JDBCMySQL {
 def main(args: Array[String]) {
 //At first create a Spark Session as the entry point of your app
 val spark:SparkSession = SparkSession
 .builder()
 .appName("JDBC-MYSQL")
 .master("local[*]")
 .config("spark.sql.warehouse.dir", "C:/Spark")
 .getOrCreate();

 val dataframe_mysql = spark.read.format("jdbc")
 .option("url", "jdbc:mysql://localhost:3306/mydb") // mydb is database
name
 .option("driver", "com.mysql.jdbc.Driver")
 .option("dbtable", "mytable") //replace table name
 .option("user", "root") //replace user name
 .option("password", "spark") // replace password
 .load()

 dataframe_mysql.show()
 }
}
```

Summary

In this chapter, we learned how we can connect Spark to the Hive metastore and use the Spark SQL language to perform DDL operations in Spark. Also, we went through how we can connect Spark SQL to different RDBMS datastores and query tables, which provide DataFrames as results. In the next chapter, we will be studying Spark Streaming, machine learning, and graph analysis.