Spark Cookbook

By introducing in-memory persistent storage, Apache Spark eliminates the need to store intermediate data in filesystems, thereby increasing processing speed by up to 100 times.

This book will focus on how to analyze large and complex sets of data. Starting with installing and configuring Apache Spark with various cluster managers, you will cover setting up development environments. You will then cover various recipes to perform interactive queries using Spark SQL and real-time streaming with various sources such as Twitter Stream and Apache Kafka. You will then focus on machine learning, including supervised learning, unsupervised learning, and recommendation engine algorithms. After mastering graph processing using GraphX, you will cover various recipes for cluster optimization and troubleshooting.

What this book will do for you...

- Install and configure Apache Spark with various cluster managers
- Set up development environments
- Perform interactive queries using Spark SQL
- Get to grips with real-time streaming analytics using Spark Streaming
- Master supervised learning and unsupervised learning using MLlib
- Build a recommendation engine using MLlib
- Develop a set of common applications or project types, and solutions that solve complex big data problems
- Use Apache Spark as your single big data compute platform and master its libraries

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Over 60 recipes on Spark, covering Spark Core, Spark SQL, Spark Streaming, MLlib, and GraphX libraries


Rishi Yadav
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'External Data Sources'
- A synopsis of the book’s content
- More information on Spark Cookbook
Rishi Yadav has 17 years of experience in designing and developing enterprise applications. He is an open source software expert and advises American companies on big data trends. Rishi was honored as one of Silicon Valley’s 40 under 40 in 2014. He finished his bachelor's degree at the prestigious Indian Institute of Technology (IIT) Delhi in 1998.

About 10 years ago, Rishi started InfoObjects, a company that helps data-driven businesses gain new insights into data.

InfoObjects combines the power of open source and big data to solve business challenges for its clients and has a special focus on Apache Spark. The company has been on the Inc. 5000 list of the fastest growing companies for 4 years in a row. InfoObjects has also been awarded with the #1 best place to work in the Bay Area in 2014 and 2015.

Rishi is an open source contributor and active blogger.
The success of Hadoop as a big data platform raised user expectations, both in terms of solving different analytics challenges as well as reducing latency. Various tools evolved over time, but when Apache Spark came, it provided one single runtime to address all these challenges. It eliminated the need to combine multiple tools with their own challenges and learning curves. By using memory for persistent storage besides compute, Apache Spark eliminates the need to store intermedia data in disk and increases processing speed up to 100 times. It also provides a single runtime, which addresses various analytics needs such as machine-learning and real-time streaming using various libraries.

This book covers the installation and configuration of Apache Spark and building solutions using Spark Core, Spark SQL, Spark Streaming, MLlib, and GraphX libraries.

For more information on this book's recipes, please visit infoobjects.com/spark-cookbook.

**What this book covers**

*Chapter 1, Getting Started with Apache Spark*, explains how to install Spark on various environments and cluster managers.

*Chapter 2, Developing Applications with Spark*, talks about developing Spark applications on different IDEs and using different build tools.

*Chapter 3, External Data Sources*, covers how to read and write to various data sources.

*Chapter 4, Spark SQL*, takes you through the Spark SQL module that helps you to access the Spark functionality using the SQL interface.

*Chapter 5, Spark Streaming*, explores the Spark Streaming library to analyze data from real-time data sources, such as Kafka.
Chapter 6, Getting Started with Machine Learning Using MLlib, covers an introduction to machine learning and basic artifacts such as vectors and matrices.

Chapter 7, Supervised Learning with MLlib – Regression, walks through supervised learning when the outcome variable is continuous.

Chapter 8, Supervised Learning with MLlib – Classification, discusses supervised learning when the outcome variable is discrete.

Chapter 9, Unsupervised Learning with MLlib, covers unsupervised learning algorithms such as k-means.

Chapter 10, Recommender Systems, introduces building recommender systems using various techniques, such as ALS.

Chapter 11, Graph Processing Using GraphX, talks about various graph processing algorithms using GraphX.

Chapter 12, Optimizations and Performance Tuning, covers various optimizations on Apache Spark and performance tuning techniques.
One of the strengths of Spark is that it provides a single runtime that can connect with various underlying data sources.

In this chapter, we will connect to different data sources. This chapter is divided into the following recipes:

- Loading data from the local filesystem
- Loading data from HDFS
- Loading data from HDFS using a custom InputFormat
- Loading data from Amazon S3
- Loading data from Apache Cassandra
- Loading data from relational databases

**Introduction**

Spark provides a unified runtime for big data. HDFS, which is Hadoop's filesystem, is the most used storage platform for Spark as it provides cost-effective storage for unstructured and semi-structured data on commodity hardware. Spark is not limited to HDFS and can work with any Hadoop-supported storage.

Hadoop supported storage means a storage format that can work with Hadoop's InputFormat and OutputFormat interfaces. InputFormat is responsible for creating InputSplits from input data and dividing it further into records. OutputFormat is responsible for writing to storage.

We will start with writing to the local filesystem and then move over to loading data from HDFS. In the *Loading data from HDFS* recipe, we will cover the most common file format: regular text files. In the next recipe, we will cover how to use any InputFormat interface to load data in Spark. We will also explore loading data stored in Amazon S3, a leading cloud storage platform.
We will explore loading data from Apache Cassandra, which is a NoSQL database. Finally, we will explore loading data from a relational database.

**Loading data from the local filesystem**

Though the local filesystem is not a good fit to store big data due to disk size limitations and lack of distributed nature, technically you can load data in distributed systems using the local filesystem. But then the file/directory you are accessing has to be available on each node.

Please note that if you are planning to use this feature to load side data, it is not a good idea. To load side data, Spark has a broadcast variable feature, which will be discussed in upcoming chapters.

In this recipe, we will look at how to load data in Spark from the local filesystem.

**How to do it...**

Let's start with the example of Shakespeare's "to be or not to be":

1. Create the words directory by using the following command:
   
   `$ mkdir words`

2. Get into the words directory:
   
   `$ cd words`

3. Create the sh.txt text file and enter "to be or not to be" in it:
   
   `$ echo "to be or not to be" > sh.txt`

4. Start the Spark shell:
   
   `$ spark-shell`

5. Load the words directory as RDD:
   
   `scala> val words = sc.textFile("file:///home/hduser/words")`

6. Count the number of lines:
   
   `scala> words.count`

7. Divide the line (or lines) into multiple words:
   
   `scala> val wordsFlatMap = words.flatMap(_.split("\W+"))`

8. Convert word to (word,1)—that is, output 1 as the value for each occurrence of word as a key:
   
   `scala> val wordsMap = wordsFlatMap.map( w => (w,1))`
9. Use the reduceByKey method to add the number of occurrences for each word as a key (this function works on two consecutive values at a time, represented by \(a\) and \(b\)):

\[
\text{scala> val wordCount = wordsMap.reduceByKey( (a, b) => (a + b))}
\]

10. Print the RDD:

\[
\text{scala> wordCount.collect.foreach(println)}
\]

11. Doing all of the preceding operations in one step is as follows:

\[
\text{scala> sc.textFile("file:///home/hduser/words").flatMap(_.split("\W+"))\cdot map( w => (w, 1)).reduceByKey( (a, b) => (a + b)).foreach(println)}
\]

This gives the following output:

\[
\begin{align*}
& (to, 2) \\
& (not, 1) \\
& (be, 2) \\
& (or, 1)
\end{align*}
\]

---

**Loading data from HDFS**

HDFS is the most widely used big data storage system. One of the reasons for the wide adoption of HDFS is schema-on-read. What this means is that HDFS does not put any restriction on data when data is being written. Any and all kinds of data are welcome and can be stored in a raw format. This feature makes it ideal storage for raw unstructured data and semi-structured data.

When it comes to reading data, even unstructured data needs to be given some structure to make sense. Hadoop uses InputFormat to determine how to read the data. Spark provides complete support for Hadoop’s InputFormat so anything that can be read by Hadoop can be read by Spark as well.

The default InputFormat is TextInputFormat. TextInputFormat takes the byte offset of a line as a key and the content of a line as a value. Spark uses the sc.textFile method to read using TextInputFormat. It ignores the byte offset and creates an RDD of strings.

Sometimes the filename itself contains useful information, for example, time-series data. In that case, you may want to read each file separately. The sc.wholeTextFiles method allows you to do that. It creates an RDD with the filename and path (for example, hdfs://localhost:9000/user/hduser/words) as a key and the content of the whole file as the value.

Spark also supports reading various serialization and compression-friendly formats such as Avro, Parquet, and JSON using DataFrames. These formats will be covered in coming chapters.

In this recipe, we will look at how to load data in the Spark shell from HDFS.
Let's do the word count, which counts the number of occurrences of each word. In this recipe, we will load data from HDFS:

1. Create the `words` directory by using the following command:
   ```bash
   $ mkdir words
   ```
2. Change the directory to `words`:
   ```bash
   $ cd words
   ```
3. Create the `sh.txt` text file and enter "to be or not to be" in it:
   ```bash
   $ echo "to be or not to be" > sh.txt
   ```
4. Start the Spark shell:
   ```bash
   $ spark-shell
   ```
5. Load the `words` directory as the RDD:
   ```scala
   scala> val words = sc.textFile("hdfs://localhost:9000/user/hduser/words")
   ```

   The `sc.textFile` method also supports passing an additional argument for the number of partitions. By default, Spark creates one partition for each `InputSplit` class, which roughly corresponds to one block.

   You can ask for a higher number of partitions. It works really well for compute-intensive jobs such as in machine learning. As one partition cannot contain more than one block, having fewer partitions than blocks is not allowed.

6. Count the number of lines (the result will be 1):
   ```scala
   scala> words.count
   ```
7. Divide the line (or lines) into multiple words:
   ```scala
   scala> val wordsFlatMap = words.flatMap(_.split("\W+"))
   ```
8. Convert word to (word,1)—that is, output 1 as a value for each occurrence of word as a key:
   ```scala
   scala> val wordsMap = wordsFlatMap.map( w => (w,1))
   ```
9. Use the `reduceByKey` method to add the number of occurrences of each word as a key (this function works on two consecutive values at a time, represented by a and b):
   ```scala
   scala> val wordCount = wordsMap.reduceByKey( (a,b) => (a+b))
   ```
10. Print the RDD:
   
   scala> wordCount.collect.foreach(println)

11. Doing all of the preceding operations in one step is as follows:

   scala> sc.textFile("hdfs://localhost:9000/user/hduser/words").flatMap(_.split("\W+"))\).map( w => (w,1)).reduceByKey(\( (a,b) => (a+b)\)).foreach(println)

This gives the following output:

   
   (to,2)
   (not,1)
   (be,2)
   (or,1)

There's more...

Sometimes we need to access the whole file at once. Sometimes the filename contains useful data like in the case of time-series. Sometimes you need to process more than one line as a record. sparkContext.wholeTextFiles comes to the rescue here. We will look at weather dataset from ftp://ftp.ncdc.noaa.gov/pub/data/noaa/.

Here's what a top-level directory looks like:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Date Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>[parent directory]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1901/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1902/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1903/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1904/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1905/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1906/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1907/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1908/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1909/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1910/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1911/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
<tr>
<td>1912/</td>
<td>11/22/04, 12:00:00 AM</td>
<td></td>
</tr>
</tbody>
</table>
Looking into a particular year directory—for example, 1901 resembles the following screenshot:


<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Date Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>029070-99999-1901.gz</td>
<td>11.2kB</td>
<td>11/22/04, 12:00:00 AM</td>
</tr>
<tr>
<td>029500-99999-1901.gz</td>
<td>10.9kB</td>
<td>11/22/04, 12:00:00 AM</td>
</tr>
<tr>
<td>029600-99999-1901.gz</td>
<td>11.4kB</td>
<td>11/22/04, 12:00:00 AM</td>
</tr>
<tr>
<td>029720-99999-1901.gz</td>
<td>10.7kB</td>
<td>11/22/04, 12:00:00 AM</td>
</tr>
<tr>
<td>029810-99999-1901.gz</td>
<td>11.7kB</td>
<td>11/22/04, 12:00:00 AM</td>
</tr>
<tr>
<td>227070-99999-1901.gz</td>
<td>10.9kB</td>
<td>11/22/04, 12:00:00 AM</td>
</tr>
</tbody>
</table>

Data here is divided in such a way that each filename contains useful information, that is, USAF-WBAN-year, where USAF is the US air force station number and WBAN is the weather bureau army navy location number.

You will also notice that all files are compressed as gzip with a .gz extension. Compression is handled automatically so all you need to do is to upload data in HDFS. We will come back to this dataset in the coming chapters.

Since the whole dataset is not large, it can be uploaded in HDFS in the pseudo-distributed mode also:

1. Download data:
   
   ```
   ```

2. Load the weather data in HDFS:
   
   ```
   $ hdfs dfs -put ftp.ncdc.noaa.gov/pub/data/noaa weather/
   ```

3. Start the Spark shell:
   
   ```
   $ spark-shell
   ```

4. Load weather data for 1901 in the RDD:
   
   ```
   scala> val weatherFileRDD = sc.wholeTextFiles("hdfs://localhost:9000/user/hduser/weather/1901")
   ```

5. Cache weather in the RDD so that it is not recomputed every time it's accessed:
   
   ```
   scala> val weatherRDD = weatherFileRDD.cache
   ```
In Spark, there are various StorageLevels at which the RDD can be persisted. `rdd.cache` is a shorthand for the `rdd.persist(MEMORY_ONLY)` StorageLevel.

6. Count the number of elements:
   ```scala
   scala> weatherRDD.count
   ```

7. Since the whole contents of a file are loaded as an element, we need to manually interpret the data, so let's load the first element:
   ```scala
   scala> val firstElement = weatherRDD.first
   ```

8. Read the value of the first RDD:
   ```scala
   scala> val firstValue = firstElement._2
   ```
   The `firstElement` contains tuples in the form (string, string). Tuples can be accessed in two ways:
   - Using a positional function starting with `_1`.
   - Using the `productElement` method, for example, `tuple.productElement(0)`. Indexes here start with 0 like most other methods.

9. Split `firstValue` by lines:
   ```scala
   scala> val firstVals = firstValue.split("\n")
   ```

10. Count the number of elements in `firstVals`:
    ```scala
        scala> firstVals.size
    ```

11. The schema of weather data is very rich with the position of the text working as a delimiter. You can get more information about schemas at the national weather service website. Let's get wind speed, which is from section 66-69 (in meter/sec):
    ```scala
        scala> val windSpeed = firstVals.map(line => line.substring(65,69))
    ```

---

## Loading data from HDFS using a custom InputFormat

Sometimes you need to load data in a specific format and `TextInputFormat` is not a good fit for that. Spark provides two methods for this purpose:

- `sparkContext.hadoopFile`: This supports the old MapReduce API
- `sparkContext.newAPIHadoopFile`: This supports the new MapReduce API

These two methods provide support for all of Hadoop's built-in InputFormats interfaces as well as any custom InputFormat.
**How to do it...**

We are going to load text data in key-value format and load it in Spark using `KeyValueTextInputFormat`:

1. Create the `currency` directory by using the following command:
   
   ```
   $ mkdir currency
   ```

2. Change the current directory to `currency`:
   
   ```
   $ cd currency
   ```

3. Create the `na.txt` text file and enter currency values in key-value format delimited by tab (key: country, value: currency):
   
   ```
   $ vi na.txt
   ```
   
   ```
   United States of America        US Dollar
   Canada  Canadian Dollar
   Mexico  Peso
   ```
   
   You can create more files for each continent.

4. Upload the `currency` folder to HDFS:
   
   ```
   $ hdfs dfs -put currency /user/hduser/currency
   ```

5. Start the Spark shell:
   
   ```
   $ spark-shell
   ```

6. Import statements:
   
   ```
   scala> import org.apache.hadoop.io.Text
   scala> import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat
   ```

7. Load the `currency` directory as the RDD:
   
   ```
   val currencyFile = sc.newAPIHadoopFile("hdfs://localhost:9000/
   user/hduser/currency",classOf[KeyValueTextInputFormat],classOf[Text],classOf[Text])
   ```

8. Convert it from tuple of `(Text,Text)` to tuple of `(String,String)`:
   
   ```
   val currencyRDD = currencyFile.map( t => (t._1.toString,t._2.toString))
   ```

9. Count the number of elements in the RDD:
   
   ```
   scala> currencyRDD.count
   ```

10. Print the values:
    
    ```
    scala> currencyRDD.collect.foreach(println)
    ```
    
    ```
    ((United States of America,US Dollar)
    (Canada,Canadian Dollar)
    (Mexico,Peso)
    ```
You can use this approach to load data in any Hadoop-supported InputFormat interface.

## Loading data from Amazon S3

Amazon Simple Storage Service (S3) provides developers and IT teams with a secure, durable, and scalable storage platform. The biggest advantage of Amazon S3 is that there is no up-front IT investment and companies can build capacity (just by clicking a button) as they need.

Though Amazon S3 can be used with any compute platform, it integrates really well with Amazon's cloud services such as Amazon Elastic Compute Cloud (EC2) and Amazon Elastic Block Storage (EBS). For this reason, companies who use Amazon Web Services (AWS) are likely to have significant data is already stored on Amazon S3.

This makes a good case for loading data in Spark from Amazon S3 and that is exactly what this recipe is about.

### How to do it...

Let's start with the AWS portal:

1. Go to [http://aws.amazon.com](http://aws.amazon.com) and log in with your username and password.
2. Once logged in, navigate to Storage & Content Delivery | S3 | Create Bucket:

   ![Create a Bucket - Select a Bucket Name and Region](image)

   A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the Amazon S3 documentation.
3. Enter the bucket name—for example, `com.infoobjects.wordcount`. Please make sure you enter a unique bucket name (no two S3 buckets can have the same name globally).

4. Select Region, click on Create, and then on the bucket name you created and you will see the following screen:

5. Click on Create Folder and enter `words` as the folder name.

6. Create the `sh.txt` text file on the local filesystem:
   ```bash
   $ echo "to be or not to be" > sh.txt
   ```

7. Navigate to Words | Upload | Add Files and choose `sh.txt` from the dialog box, as shown in the following screenshot:

8. Click on Start Upload.
9. Select `sh.txt` and click on **Properties** and it will show you details of the file:

10. Set `AWS_ACCESS_KEY` and `AWS_SECRET_ACCESS_KEY` as environment variables.

11. Open the Spark shell and load the `words` directory from `s3` in the `words` RDD:

```scala
scala> val words = sc.textFile("s3n://com.infoobjects.wordcount/words")
```

Now the RDD is loaded and you can continue doing regular transformations and actions on the RDD.

Sometimes there is confusion between `s3://` and `s3n://`. `s3n://` means a regular file sitting in the S3 bucket but readable and writable by the outside world. This filesystem puts a 5 GB limit on the file size.

`s3://` means an HDFS file sitting in the S3 bucket. It is a block-based filesystem. The filesystem requires you to dedicate a bucket for this filesystem. There is no limit on file size in this system.

## Loading data from Apache Cassandra

Apache Cassandra is a NoSQL database with a masterless ring cluster structure. While HDFS is a good fit for streaming data access, it does not work well with random access. For example, HDFS will work well when your average file size is 100 MB and you want to read the whole file. If you frequently access the n-th line in a file or some other part as a record, HDFS would be too slow.

Relational databases have traditionally provided a solution to that, providing low latency, random access, but they do not work well with big data. NoSQL databases such as Cassandra fill the gap by providing relational database type access but in a distributed architecture on commodity servers.
In this recipe, we will load data from Cassandra as a Spark RDD. To make that happen, Datastax, the company behind Cassandra, has contributed spark-cassandra-connector. This connector lets you load Cassandra tables as Spark RDDs, write Spark RDDs back to Cassandra, and execute CQL queries.

**How to do it...**

Perform the following steps to load data from Cassandra:

1. Create a keyspace named `people` in Cassandra using the CQL shell:
   ```
cqlsh> CREATE KEYSPACE people WITH replication = { 'class': 'SimpleStrategy', 'replication_factor': 1 }; ```

2. Create a column family (from CQL 3.0 onwards, it can also be called a table) `person` in newer versions of Cassandra:
   ```
cqlsh> create columnfamily person(id int primary key, first_name varchar, last_name varchar);
   ```

3. Insert a few records in the column family:
   ```
cqlsh> insert into person(id, first_name, last_name) values(1, 'Barack', 'Obama');
cqlsh> insert into person(id, first_name, last_name) values(2, 'Joe', 'Smith');
   ```

4. Add Cassandra connector dependency to SBT:
   ```
   "com.datastax.spark" %% "spark-cassandra-connector" % 1.2.0
   ```

5. You can also add the Cassandra dependency to Maven:
   ```
   <dependency>
   <groupId>com.datastax.spark</groupId>
   <artifactId>spark-cassandra-connector_2.10</artifactId>
   <version>1.2.0</version>
   </dependency>
   ```

Alternatively, you can also download the spark-cassandra-connector JAR to use directly with the Spark shell:

```
$ wget http://central.maven.org/maven2/com/datastax/spark/spark-cassandra-connector_2.10/1.1.0/spark-cassandra-connector_2.10-1.2.0.jar
```

If you would like to build the uber JAR with all dependencies, refer to the There's more... section.
6. Now start the Spark shell.

7. Set the `spark.cassandra.connection.host` property in the Spark shell:
   ```scala
   scala> sc.getConf.set("spark.cassandra.connection.host", "localhost")
   ```

8. Import Cassandra-specific libraries:
   ```scala
   scala> import com.datastax.spark.connector._
   ```

9. Load the `person` column family as an RDD:
   ```scala
   scala> val personRDD = sc.cassandraTable("people","person")
   ```

10. Count the number of records in the RDD:
    ```scala
        scala> personRDD.count
    ```

11. Print data in the RDD:
    ```scala
        scala> personRDD.collect.foreach(println)
    ```

12. Retrieve the first row:
    ```scala
        scala> val firstRow = personRDD.first
    ```

13. Get the column names:
    ```scala
        scala> firstRow.columnNames
    ```

14. Cassandra can also be accessed through Spark SQL. It has a wrapper around `SQLContext` called `CassandraSQLContext`; let's load it:
    ```scala
        scala> val cc = new org.apache.spark.sql.cassandra.CassandraSQLContext(sc)
    ```

15. Load the `person` data as `SchemaRDD`:
    ```scala
        scala> val p = cc.sql("select * from people.person")
    ```

16. Retrieve the `person` data:
    ```scala
        scala> p.collect.foreach(println)
    ```

---

**There's more...**

Spark Cassandra's connector library has a lot of dependencies. The connector itself and several of its dependencies are third-party to Spark and are not available as part of the Spark installation.
These dependencies need to be made available to the driver as well as executors at runtime. One way to do this is to bundle all transitive dependencies, but that is a laborious and error-prone process. The recommended approach is to bundle all the dependencies along with the connector library. This will result in a fat JAR, popularly known as the uber JAR.

SBT provides the sbt-assembly plugin, which makes creating uber JARs very easy. The following are the steps to create an uber JAR for spark-cassandra-connector. These steps are general enough so that you can use them to create any uber JAR:

1. Create a folder named uber:
   
   $ mkdir uber

2. Change the directory to uber:

   $ cd uber

3. Open the SBT prompt:

   $ sbt

4. Give this project a name sc-uber:

   > set name := "sc-uber"

5. Save the session:

   > session save

6. Exit the session:

   > exit

   This will create build.sbt, project, and target folders in the uber folder as shown in the following screenshot:

```
hduser@localhost:/uber$ ls
build.sbt  project  target
```

7. Add the spark-cassandra-driver dependency to build.sbt at the end after leaving a blank line as shown in the following screenshot:

   $ vi build.sbt

```
name := "sc-uber"
libraryDependencies += "com.datastax.spark" %% "spark-cassandra-connector" % "1.1.0"
```
8. We will use `MergeStrategy.first` as the default. Besides that, there are some files, such as `manifest.mf`, that every JAR bundles for metadata, and we can simply discard them. We are going to use `MergeStrategy.discard` for that. The following is the screenshot of `build.sbt` with `assemblyMergeStrategy` added:

```
libraryDependencies ++ "com.datastax.spark" %% "spark-cassandra-connector" % "1.1.0"

assemblyMergeStrategy in assembly := {
  case PathList("META-INF", xs @ _) =>
    (xs map (_:toLowerCase)) match {
      case ("manifest.mf" :: Nil) | ("index.list" :: Nil) | ("dependencies" :: Nil) => MergeStrategy.discard
      case _ => MergeStrategy.first
    }
  case _ => MergeStrategy.first
}
```

9. Now create `plugins.sbt` in the project folder and type the following for the `sbt-assembly` plugin:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.12.0")
```

10. We are ready to build (assembly) a JAR now:

```
$ sbt assembly
```

The `uber` JAR is now created in `target/scala-2.10/sc-uber-assembly-0.1-SNAPSHOT.jar`.

11. Copy it to a suitable location where you keep all third-party JARs—for example, `/home/hduser/thirdparty`—and rename it to an easier name (unless you like longer names):

```
$ mv thirdparty/sc-uber-assembly-0.1-SNAPSHOT.jar thirdparty/sc-uber.jar
```

12. Load the Spark shell with the `uber` JAR using `--jars`:

```
$ spark-shell --jars thirdparty/sc-uber.jar
```

13. To submit the Scala code to a cluster, you can call `spark-submit` with the same JARS option:

```
$ spark-submit --jars thirdparty/sc-uber.jar
```

**Merge strategies in sbt-assembly**

If multiple JARs have files with the same name and the same relative path, the default merge strategy for the `sbt-assembly` plugin is to verify that content is same for all the files and error out otherwise. This strategy is called `MergeStrategy.deduplicate`. 
The following are the available merge strategies in the sbt-assembly plugin:

<table>
<thead>
<tr>
<th>Strategy name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeStrategy.deduplicate</td>
<td>The default strategy</td>
</tr>
<tr>
<td>MergeStrategy.first</td>
<td>Picks first file according to classpath</td>
</tr>
<tr>
<td>MergeStrategy.last</td>
<td>Picks last file according to classpath</td>
</tr>
<tr>
<td>MergeStrategy.singleOrError</td>
<td>Errors out (merge conflict not expected)</td>
</tr>
<tr>
<td>MergeStrategy.concat</td>
<td>Concatenates all matching files together</td>
</tr>
<tr>
<td>MergeStrategy.filterDistinctLines</td>
<td>Concatenates leaving out duplicates</td>
</tr>
<tr>
<td>MergeStrategy.rename</td>
<td>Renames files</td>
</tr>
</tbody>
</table>

**Loading data from relational databases**

A lot of important data lies in relational databases that Spark needs to query. JdbcRDD is a Spark feature that allows relational tables to be loaded as RDDs. This recipe will explain how to use JdbcRDD.

Spark SQL to be introduced in the next chapter includes a data source for JDBC. This should be preferred over the current recipe as results are returned as DataFrames (to be introduced in the next chapter), which can be easily processed by Spark SQL and also joined with other data sources.

**Getting ready**

Please make sure that the JDBC driver JAR is visible on the client node and all slaves nodes on which executor will run.

**How to do it...**

Perform the following steps to load data from relational databases:

1. Create a table named `person` in MySQL using the following DDL:

```sql
CREATE TABLE 'person' (
    'person_id' int(11) NOT NULL AUTO_INCREMENT,
    'first_name' varchar(30) DEFAULT NULL,
    'last_name' varchar(30) DEFAULT NULL,
    'gender' char(1) DEFAULT NULL,
    PRIMARY KEY ('person_id');
)
```
2. Insert some data:

   Insert into person values('Barack','Obama','M');
   Insert into person values('Bill','Clinton','M');
   Insert into person values('Hillary','Clinton','F');


4. Make the MySQL driver available to the Spark shell and launch it:

   $ spark-shell --jars /path-to-mysql-jar/mysql-connector-java-5.1.29-bin.jar

   Please note that path-to-mysql-jar is not the actual path name. You should use the actual path name.

5. Create variables for the username, password, and JDBC URL:

   scala> val url="jdbc:mysql://localhost:3306/hadoopdb"
   scala> val username = "hduser"
   scala> val password = "******"

6. Import JdbcRDD:

   scala> import org.apache.spark.rdd.JdbcRDD

7. Import JDBC-related classes:

   scala> import java.sql.{Connection, DriverManager, ResultSet}

8. Create an instance of the JDBC driver:

   scala> Class.forName("com.mysql.jdbc.Driver").newInstance

9. Load JdbcRDD:

   scala> val myRDD = new JdbcRDD( sc, () =>
       DriverManager.getConnection(url,username,password),
       "select first_name,last_name,gender from person limit ?, ?",
       1, 5, 2, r => r.getString("last_name") + ", " +
       r.getString("first_name"))

10. Now query the results:

    scala> myRDD.count
    scala> myRDD.foreach(println)

11. Save the RDD to HDFS:

    scala> myRDD.saveAsTextFile("hdfs://localhost:9000/user/hduser/person")
**How it works...**

JdbcRDD is an RDD that executes a SQL query on a JDBC connection and retrieves the results. The following is a JdbcRDD constructor:

```scala
JdbcRDD( SparkContext, getConnection: () => Connection,
  sql: String, lowerBound: Long, upperBound: Long,
  numPartitions: Int, mapRow: (ResultSet) => T =
  JdbcRDD.resultSetToObjectArray)
```

The two ?'s are bind variables for a prepared statement inside JdbcRDD. The first ? is for the offset (lower bound), that is, which row should we start computing with, the second ? is for the limit (upper bound), that is, how many rows should we read.

JdbcRDD is a great way to load data in Spark directly from relational databases on an ad-hoc basis. If you would like to load data in bulk from RDBMS, there are other approaches that would work better, for example, Apache Sqoop is a powerful tool that imports and exports data from relational databases to HDFS.
Where to buy this book
You can buy Spark Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.