5 First steps in big data

This chapter covers

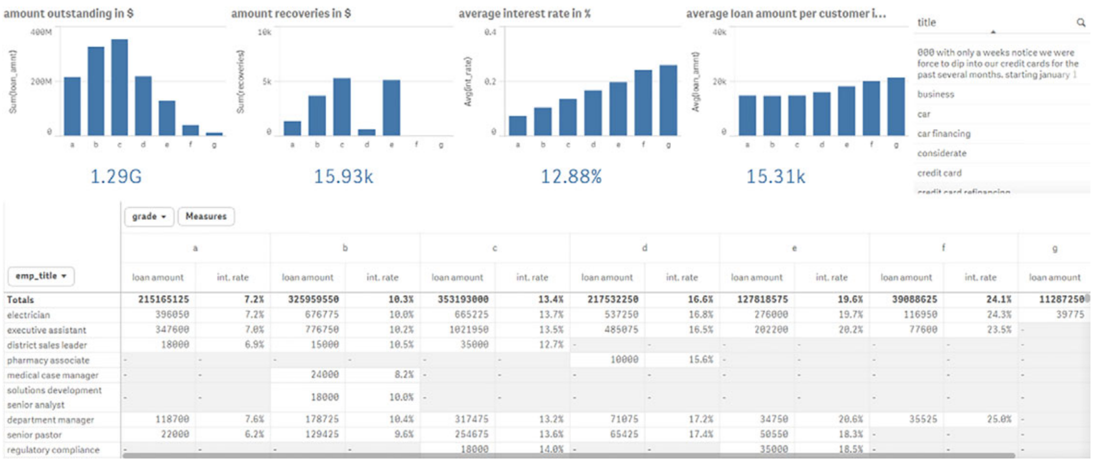
- Taking your first steps with two big data applications: Hadoop and Spark
- Using Python to write big data jobs
- Building an interactive dashboard that connects to data stored in a big data database

Over the last two chapters, we've steadily increased the size of the data. In chapter 3 we worked with data sets that could fit into the main memory of a computer. Chapter 4 introduced techniques to deal with data sets that were too large to fit in memory but could still be processed on a single computer. In this chapter you'll learn to work with technologies that can handle data that's so large a single node (computer) no longer suffices. In fact it may not even fit on a hundred computers. Now that's a challenge, isn't it?

We'll stay as close as possible to the way of working from the previous chapters; the focus is on giving you the confidence to work on a big data platform. To do this, the main part of this chapter is a case study. You'll create a dashboard that allows you to explore data from lenders of a bank. By the end of this chapter you'll have gone through the following steps:

- Load data into Hadoop, the most common big data platform.
- Transform and clean data with Spark.
- Store it into a big data database called Hive.
- Interactively visualize this data with Qlik Sense, a visualization tool.

All this (apart from the visualization) will be coordinated from within a Python script. The end result is a dashboard that allows you to explore the data, as shown in figure 5.1.



000 with only a weeks notice we were force to dip into our credit cards for the past several months, starting january 1
business
car
car financing
considerate
credit card
credit card refinancing

	a		b		c		d		e		f		g
emp_title	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount
Totals	**215165125**	**7.2%**	**325959550**	**10.3%**	**353193000**	**13.4%**	**217532250**	**16.6%**	**127818575**	**19.6%**	**39088625**	**24.1%**	**11287250**
electrician	396050	7.2%	676775	10.0%	665225	13.7%	537250	16.8%	276000	19.7%	116950	24.3%	39775
executive assistant	347600	7.0%	776750	10.2%	1021950	13.5%	485075	16.5%	202200	20.2%	77600	23.5%	-
district sales leader	18000	6.9%	15000	10.5%	35000	12.7%	-	-	-	-	-	-	-
pharmacy associate	-	-	-	-	-	-	10000	15.6%	-	-	-	-	-
medical case manager	-	-	24000	8.2%	-	-	-	-	-	-	-	-	-
solutions development senior analyst	-	-	18000	10.0%	-	-	-	-	-	-	-	-	-
department manager	118700	7.6%	178725	10.4%	317475	13.2%	71075	17.2%	34750	20.6%	35525	25.0%	-
senior pastor	22000	6.2%	129425	9.6%	254675	13.6%	65425	17.4%	50550	18.3%	-	-	-
regulatory compliance	-	-	-	-	18000	14.0%	-	-	35000	18.5%	-	-	-

Figure 5.1 Interactive Qlik dashboard

Bear in mind that we'll only scratch the surface of both practice and theory in this introductory chapter on big data technologies. The case study will touch three big data technologies (Hadoop, Spark, and Hive), but only for data manipulation, not model building. It will be up to you to combine the big data technologies you get to see here with the model-building techniques we touched upon in previous chapters.

5.1 Distributing data storage and processing with frameworks

New big data technologies such as Hadoop and Spark make it much easier to work with and control a cluster of computers. Hadoop can scale up to thousands of computers, creating a cluster with petabytes of storage. This enables businesses to grasp the value of the massive amount of data available.

5.1.1 Hadoop: a framework for storing and processing large data sets

Apache Hadoop is a framework that simplifies working with a cluster of computers. It aims to be all of the following things and more:

- *Reliable*—By automatically creating multiple copies of the data and redeploying processing logic in case of failure.
- *Fault tolerant*—It detects faults and applies automatic recovery.
- *Scalable*—Data and its processing are distributed over clusters of computers (horizontal scaling).
- *Portable*—Installable on all kinds of hardware and operating systems.

The core framework is composed of a distributed file system, a resource manager, and a system to run distributed programs. In practice it allows you to work with the distributed file system almost as easily as with the local file system of your home computer. But in the background, the data can be scattered among thousands of servers.

The different components of Hadoop

At the heart of Hadoop we find

- A distributed file system (HDFS)
- A method to execute programs on a massive scale (MapReduce)
- A system to manage the cluster resources (YARN)

On top of that, an ecosystem of applications arose (figure 5.2), such as the databases Hive and HBase and frameworks for machine learning such as Mahout. We'll use Hive in this chapter. Hive has a language based on the widely used SQL to interact with data stored inside the database.

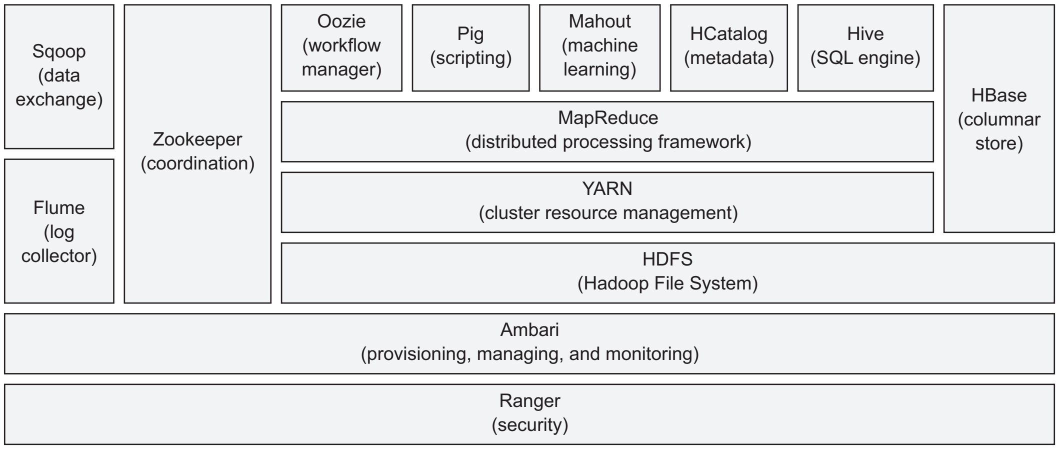


Figure 5.2 A sample from the ecosystem of applications that arose around the Hadoop Core Framework

It's possible to use the popular tool Impala to query Hive data up to 100 times faster. We won't go into Impala in this book, but more information can be found at http://impala.io/. We already had a short intro to MapReduce in chapter 4, but let's elaborate a bit here because it's such a vital part of Hadoop.

MapReduce: How Hadoop achieves parallelism

Hadoop uses a programming method called MapReduce to achieve parallelism. A MapReduce algorithm splits up the data, processes it in parallel, and then sorts, combines, and aggregates the results back together. However, the MapReduce algorithm isn't well suited for interactive analysis or iterative programs because it writes the data to a disk in between each computational step. This is expensive when working with large data sets.

Let's see how MapReduce would work on a small fictitious example. You're the director of a toy company. Every toy has two colors, and when a client orders a toy from the web page, the web page puts an order file on Hadoop with the colors of the toy. Your task is to find out how many color units you need to prepare. You'll use a MapReduce-style algorithm to count the colors. First let's look at a simplified version in figure 5.3.

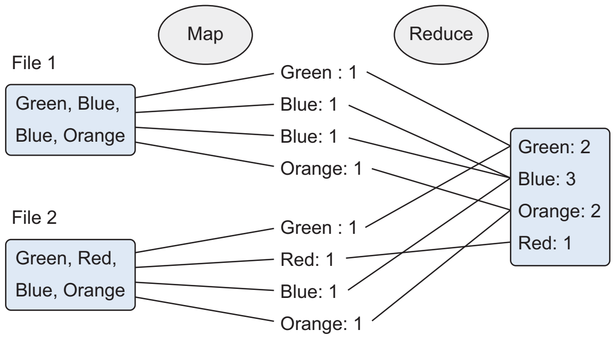


Figure 5.3 A simplified example of a MapReduce flow for counting the colors in input texts

As the name suggests, the process roughly boils down to two big phases:

- *Mapping phase*—The documents are split up into key-value pairs. Until we reduce, we can have many duplicates.
- *Reduce phase*—It's not unlike a SQL "group by." The different unique occurrences are grouped together, and depending on the reducing function, a different result can be created. Here we wanted a count per color, so that's what the reduce function returns.

In reality it's a bit more complicated than this though.

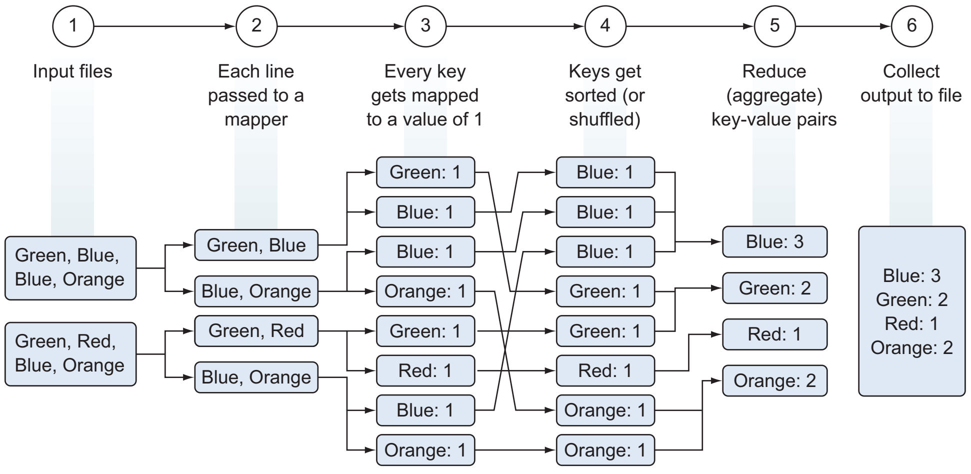


Figure 5.4 An example of a MapReduce flow for counting the colors in input texts

The whole process is described in the following six steps and depicted in figure 5.4.

1. Reading the input files.
2. Passing each line to a mapper job.
3. The mapper job parses the colors (keys) out of the file and outputs a file for each color with the number of times it has been encountered (value). Or more technically said, it maps a key (the color) to a value (the number of occurrences).
4. The keys get shuffled and sorted to facilitate the aggregation.
5. The reduce phase sums the number of occurrences per color and outputs one file per key with the total number of occurrences for each color.
6. The keys are collected in an output file.

NOTE While Hadoop makes working with big data easy, setting up a good working cluster still isn't trivial, but cluster managers such as Apache Mesos do ease the burden. In reality, many (mid-sized) companies lack the competence to maintain a healthy Hadoop installation. This is why we'll work with the Hortonworks Sandbox, a pre-installed and configured Hadoop ecosystem. Installation instructions can be found in section 1.5: An introductory working example of Hadoop.

Now, keeping the workings of Hadoop in mind, let's look at Spark.

5.1.2 Spark: replacing MapReduce for better performance

Data scientists often do interactive analysis and rely on algorithms that are inherently iterative; it can take awhile until an algorithm converges to a solution. As this is a weak point of the MapReduce framework, we'll introduce the Spark Framework to overcome it. Spark improves the performance on such tasks by an order of magnitude.

What is Spark?

Spark is a cluster computing framework similar to MapReduce. Spark, however, doesn't handle the storage of files on the (distributed) file system itself, nor does it handle the resource management. For this it relies on systems such as the Hadoop File System, YARN, or Apache Mesos. Hadoop and Spark are thus complementary systems. For testing and development, you can even run Spark on your local system.

How does Spark solve the problems of MapReduce?

While we oversimplify things a bit for the sake of clarity, Spark creates a kind of shared RAM memory between the computers of your cluster. This allows the different workers to share variables (and their state) and thus eliminates the need to write the intermediate results to disk. More technically and more correctly if you're into that: Spark uses Resilient Distributed Datasets (RDD), which are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant way.[1] Because it's an in-memory system, it avoids costly disk operations.

The different components of the Spark ecosystem

Spark core provides a NoSQL environment well suited for interactive, exploratory analysis. Spark can be run in batch and interactive mode and supports Python.

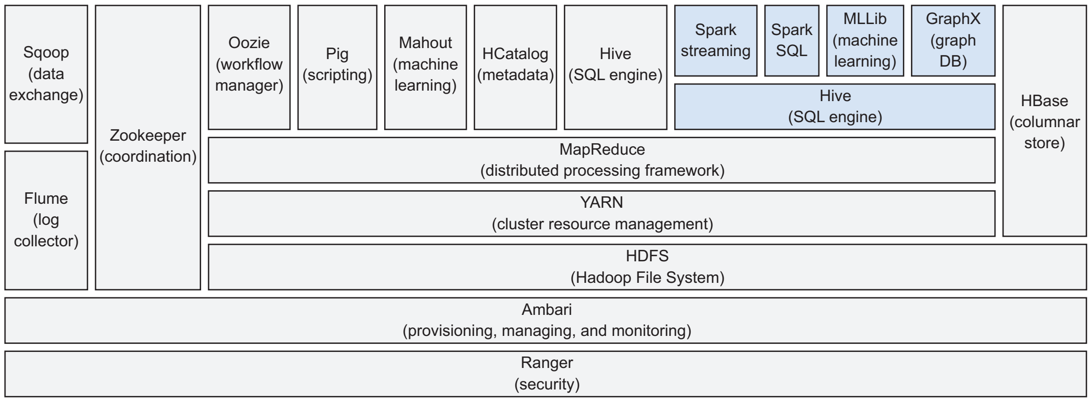


Figure 5.5 The Spark framework when used in combination with the Hadoop framework

Spark has four other large components, as listed below and depicted in figure 5.5.

1. Spark streaming is a tool for real-time analysis.
2. Spark SQL provides a SQL interface to work with Spark.
3. MLLib is a tool for machine learning inside the Spark framework.
4. GraphX is a graph database for Spark. We'll go deeper into graph databases in chapter 7.

[1] See https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf.

Now let's dip our toes into loan data using Hadoop, Hive, and Spark.

5.2 Case study: Assessing risk when loaning money

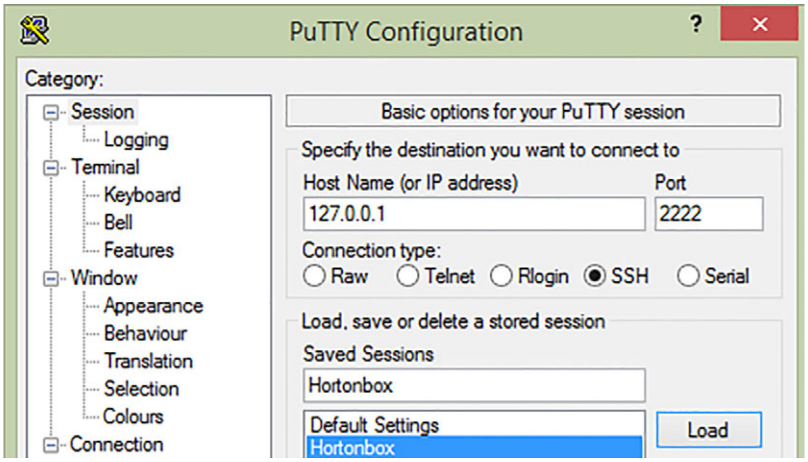
Enriched with a basic understanding of Hadoop and Spark, we're now ready to get our hands dirty on big data. The goal of this case study is to have a first experience with the technologies we introduced earlier in this chapter, and see that for a large part you can (but don't have to) work similarly as with other technologies. Note: The portion of the data used here isn't that big because that would require serious bandwidth to collect it and multiple nodes to follow along with the example.

What we'll use

- Horton Sandbox on a virtual machine. If you haven't downloaded and imported this to VM software such as VirtualBox, please go back to section 1.5 where this is explained. Version 2.3.2 of the Horton Sandbox was used when writing this chapter.
- Python libraries: Pandas and pywebhdsf. They don't need to be installed on your local virtual environment this time around; we need them directly on the Horton Sandbox. Therefore we need to fire up the Horton Sandbox (on VirtualBox, for instance) and make a few preparations.

In the Sandbox command line there are several things you still need to do for this all to work, so connect to the command line. You can do this using a program like PuTTY. If you're unfamiliar with PuTTY, it offers a command line interface to servers and can be downloaded freely at http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html.

The PuTTY login configuration is shown in figure 5.6.



Specify the destination you want to connect to

Host Name (or IP address): 127.0.0.1
Port: 2222

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Saved Sessions: Hortonbox

Default Settings
Hortonbox

Figure 5.6 Connecting to Horton Sandbox using PuTTY

The default user and password are (at the time of writing) “root” and “hadoop”, respectively. You’ll need to change this password at the first login, though.

Once connected, issue the following commands:

- `yum -y install python-pip`—This installs pip, a Python package manager.
- `pip install git+https://github.com/DavyCielen/pywebhdfs.git -upgrade`—At the time of writing there was a problem with the pywebhdfs library and we fixed that in this fork. Hopefully you won’t require this anymore when you read this; the problem has been signaled and should be resolved by the maintainers of this package.
- `pip install pandas`—To install Pandas. This usually takes awhile because of the dependencies.

An .ipynb file is available for you to open in Jupyter or (the older) Ipython and follow along with the code in this chapter. Setup instructions for Horton Sandbox are repeated there; make sure to run the code directly on the Horton Sandbox. Now, with the preparatory business out of the way, let’s look at what we’ll need to do.

In this exercise, we’ll go through several more of the data science process steps:

Step 1: The research goal. This consists of two parts:

- Providing our manager with a dashboard
- Preparing data for other people to create their own dashboards

Step 2: Data retrieval

- Downloading the data from the lending club website
- Putting the data on the Hadoop File System of the Horton Sandbox

Step 3: Data preparation

- Transforming this data with Spark
- Storing the prepared data in Hive

Steps 4 & 6: Exploration and report creation

- Visualizing the data with Qlik Sense

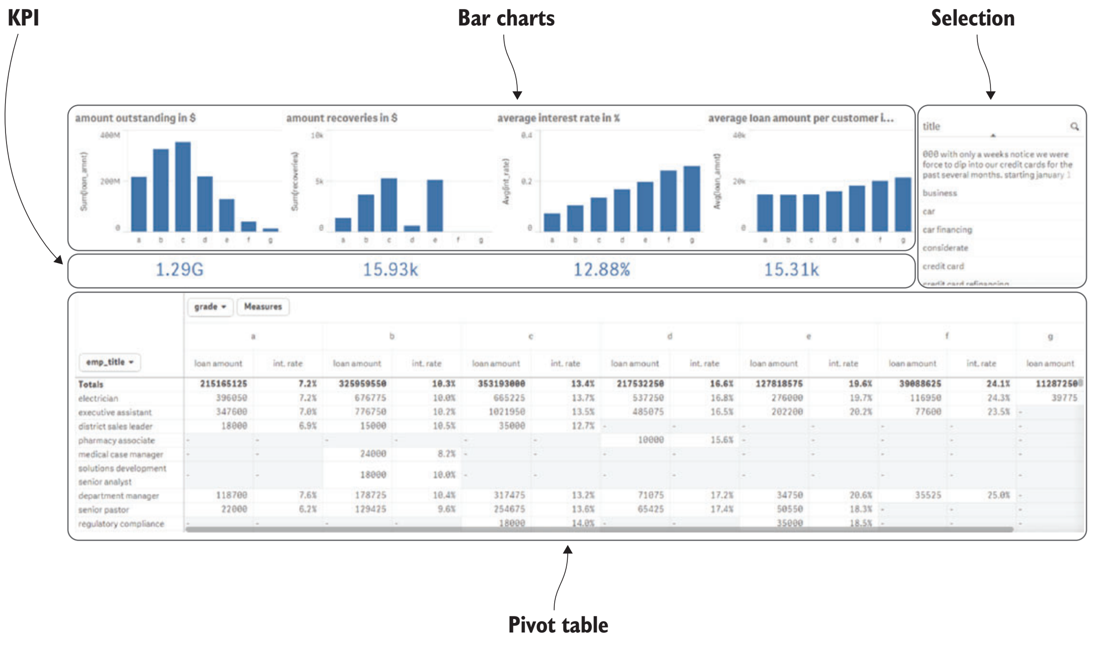
We have no model building in this case study, but you’ll have the infrastructure in place to do this yourself if you want to. For instance, you can use SPARK Machine learning to try to predict when someone will default on his debt.

It’s time to meet the Lending Club.

5.2.1 Step 1: The research goal

The Lending Club is an organization that connects people in need of a loan with people who have money to invest. Your boss also has money to invest and wants information before throwing a substantial sum on the table. To achieve this, you’ll create a report for him that gives him insight into the average rating, risks, and return for lending money to a certain person. By going through this process, you make the data accessible in a dashboard tool, thus enabling other people to explore it as well. In a sense this is the secondary goal of this case: opening up the data for self-service BI. Self-service Business Intelligence is often applied in data-driven organizations that don't have analysts to spare. Anyone in the organization can do the simple slicing and dicing themselves while leaving the more complicated analytics for the data scientist.

We can do this case study because the Lending Club makes anonymous data available about the existing loans. By the end of this case study, you'll create a report similar to figure 5.7.



	a		b		c		d		e		f		g
	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount
Totals	**215165125**	**7.2%**	**325959550**	**10.3%**	**353193000**	**13.4%**	**217532250**	**16.6%**	**127818575**	**19.6%**	**39088625**	**24.1%**	**11287250**
electrician	396050	7.2%	676775	10.0%	665225	13.7%	537250	16.8%	276000	19.7%	116950	24.3%	39775
executive assistant	347600	7.0%	776750	10.2%	1021950	13.5%	485075	16.5%	202200	20.2%	77600	23.5%	-
district sales leader	18000	6.9%	15000	10.5%	35000	12.7%	-	-	-	-	-	-	-
pharmacy associate	-	-	-	-	-	-	10000	15.6%	-	-	-	-	-
medical case manager	-	-	24000	8.2%	-	-	-	-	-	-	-	-	-
solutions development senior analyst	-	-	18000	10.0%	-	-	-	-	-	-	-	-	-
department manager	118700	7.6%	178725	10.4%	317475	13.2%	71075	17.2%	34750	20.6%	35525	25.0%	-
senior pastor	22000	6.2%	129425	9.6%	254675	13.6%	65425	17.4%	50550	18.3%	-	-	-
regulatory compliance	-	-	-	-	18000	14.0%	-	-	35000	18.5%	-	-	-

Figure 5.7 The end result of this exercise is an explanatory dashboard to compare a lending opportunity to similar opportunities.

First things first, however; let's get ourselves data.

5.2.2 Step 2: Data retrieval

It's time to work with the Hadoop File System (or hdfs). First we'll send commands through the command line and then through the Python scripting language with the help of the pywebhdfs package.

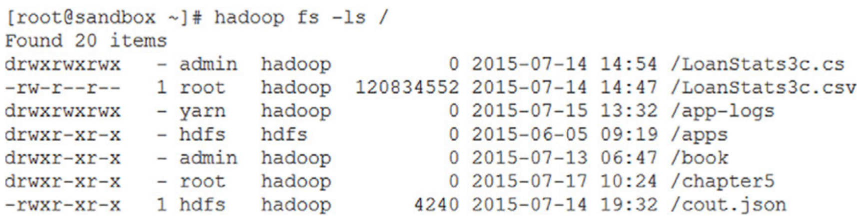
The Hadoop file system is similar to a normal file system, except that the files and folders are stored over multiple servers and you don't know the physical address of each file. This is not unfamiliar if you've worked with tools such as Dropbox or Google Drive. The files you put on these drives are stored somewhere on a server without you knowing exactly on which server. As on a normal file system, you can create, rename, and delete files and folders.

Using the command line to interact with the Hadoop file system

Let's first retrieve the currently present list of directories and files in the Hadoop root folder using the command line. Type the command `hadoop fs -ls /` in PuTTY to achieve this.

Make sure you turn on your virtual machine with the Hortonworks Sandbox before attempting a connection. In PuTTY you should then connect to 127.0.0.1:2222, as shown before in figure 5.6.

The output of the Hadoop command is shown in figure 5.8. You can also add arguments such as `hadoop fs -ls -R /` to get a recursive list of all the files and subdirectories.



```
[root@sandbox ~]# hadoop fs -ls /
Found 20 items
drwxrwxrwx   - admin  hadoop          0 2015-07-14 14:54 /LoanStats3c.cs
-rw-r--r--   1 root   hadoop  120834552 2015-07-14 14:47 /LoanStats3c.csv
drwxrwxrwx   - yarn   hadoop          0 2015-07-15 13:32 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2015-06-05 09:19 /apps
drwxr-xr-x   - admin  hadoop          0 2015-07-13 06:47 /book
drwxr-xr-x   - root   hadoop          0 2015-07-17 10:24 /chapter5
-rwxr-xr-x   1 hdfs   hadoop       4240 2015-07-14 19:32 /cout.json
```

Figure 5.8 Output from the Hadoop list command: hadoop fs –ls /. The Hadoop root folder is listed.

We'll now create a new directory "chapter5" on hdfs to work with during this chapter. The following commands will create the new directory and give everybody access to the folder:

```
sudo -u hdfs hadoop fs -mkdir /chapter5
sudo -u hdfs hadoop fs –chmod 777 /chapter5
```

You probably noticed a pattern here. The Hadoop commands are very similar to our local file system commands (POSIX style) but start with Hadoop fs and have a dash - before each command. Table 5.1 gives an overview of popular file system commands on Hadoop and their local file system command counterparts.

Table 5.1 List of common Hadoop file system commands

Goal	Hadoop file system command	Local file system command
Get a list of files and directories from a directory	hadoop fs –ls URI	ls URI
Create a directory	hadoop fs –mkdir URI	mkdir URI
Remove a directory	hadoop fs –rm –r URI	rm –r URI

Table 5.1 List of common Hadoop file system commands

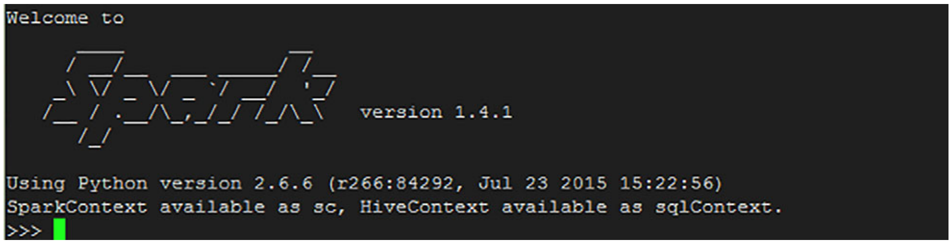
Goal	Hadoop file system command	Local file system command
Change the permission of files	hadoop fs –chmod MODE URI	chmod MODE URI
Move or rename file	hadoop fs –mv OLDURI NEWURI	mv OLDURI NEWURI

There are two special commands you'll use often. These are

- Upload files from the local file system to the distributed file system (`hadoop fs -put LOCALURI REMOTEURI`).
- Download a file from the distributed file system to the local file system (`hadoop -get REMOTEURI`).

Let's clarify this with an example. Suppose you have a .CSV file on the Linux virtual machine from which you connect to the Linux Hadoop cluster. You want to copy the .CSV file from your Linux virtual machine to the cluster hdfs. Use the command `hadoop -put mycsv.csv /data`.

Using PuTTY we can start a Python session on the Horton Sandbox to retrieve our data using a Python script. Issue the "`pyspark`" command in the command line to start the session. If all is well you should see the welcome screen shown in figure 5.9.



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

Figure 5.9 The welcome screen of Spark for interactive use with Python

Now we use Python code to fetch the data for us, as shown in the following listing.

Listing 5.1 Drawing in the Lending Club loan data

```
import requests
import zipfile
import StringIO
source = requests.get("https://resources.lendingclub.com/
     LoanStats3d.csv.zip", verify=False)
stringio = StringIO.StringIO(source.content)
unzipped = zipfile.ZipFile(stringio)
```

Downloads data from Lending Club. This is https so it should verify, but we won't bother (verify=False).

Creates virtual file.

Unzips data.

We download the file “LoanStats3d.csv.zip” from the Lending Club’s website at https://resources.lendingclub.com/LoanStats3d.csv.zip and unzip it. We use methods from the requests, zipfile, and stringio Python packages to respectively download the data, create a virtual file, and unzip it. This is only a single file; if you want all their data you could create a loop, but for demonstration purposes this will do. As we mentioned before, an important part of this case study will be data preparation with big data technologies. Before we can do so, however, we need to put it on the Hadoop file system. PyWebHdfs is a package that allows you to interact with the Hadoop file system from Python. It translates and passes your commands to rest calls for the webhdfs interface. This is useful because you can use your favorite scripting language to automate tasks, as shown in the following listing.

Listing 5.2 Storing data on Hadoop

Stores it locally because we need to transfer it to Hadoop file system.

Does preliminary data cleaning using Pandas: removes top row and bottom 2 rows because they’re useless. Opening original file will show you this.

```
import pandas as pd
from pywebhdfs.webhdfs import PyWebHdfsClient
subselection_csv = pd.read_csv(unzipped.open('LoanStats3d.csv'),
     skiprows=1,skipfooter=2,engine='python')
stored_csv = subselection_csv.to_csv('./stored_csv.csv')
hdfs = PyWebHdfsClient(user_name="hdfs",port=50070,host="sandbox")
hdfs.make_dir('chapter5')
with open('./stored_csv.csv') as file_data:
        hdfs.create_file('chapter5/LoanStats3d.csv',file_data,
     overwrite=True)
```

Creates folder "chapter5" on Hadoop file system.

Connects to Hadoop Sandbox.

Creates .csv file on Hadoop file system.

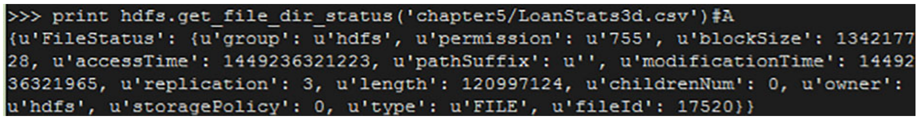
Opens locally stored csv.

We had already downloaded and unzipped the file in listing 5.1; now in listing 5.2 we made a sub-selection of the data using Pandas and stored it locally. Then we created a directory on Hadoop and transferred the local file to Hadoop. The downloaded data is in .CSV format and because it’s rather small, we can use the Pandas library to remove the first line and last two lines from the file. These contain comments and will only make working with this file cumbersome in a Hadoop environment. The first line of our code imports the Pandas package, while the second line parses the file into memory and removes the first and last two data lines. The third code line saves the data to the local file system for later use and easy inspection.

Before moving on, we can check our file using the following line of code:

```
print hdfs.get_file_dir_status('chapter5/LoanStats3d.csv')
```

The PySpark console should tell us our file is safe and well on the Hadoop system, as shown in figure 5.10.



```
>>> print hdfs.get_file_dir_status('chapter5/LoanStats3d.csv')#A
{u'FileStatus': {u'group': u'hdfs', u'permission': u'755', u'blockSize': 1342177
28, u'accessTime': 1449236321223, u'pathSuffix': u'', u'modificationTime': 14492
36321965, u'replication': 3, u'length': 120997124, u'childrenNum': 0, u'owner':
u'hdfs', u'storagePolicy': 0, u'type': u'FILE', u'fileId': 17520}}
```

Figure 5.10 Retrieve file status on Hadoop via the PySpark console

With the file ready and waiting for us on Hadoop, we can move on to data preparation using Spark, because it's not clean enough to directly store in Hive.

5.2.3 Step 3: Data preparation

Now that we've downloaded the data for analysis, we'll use Spark to clean the data before we store it in Hive.

Data preparation in Spark

Cleaning data is often an interactive exercise, because you spot a problem and fix the problem, and you'll likely do this a couple of times before you have clean and crisp data. An example of dirty data would be a string such as "UsA", which is improperly capitalized. At this point, we no longer work in jobs.py but use the PySpark command line interface to interact directly with Spark.

Spark is well suited for this type of interactive analysis because it doesn't need to save the data after each step and has a much better model than Hadoop for sharing data between servers (a kind of distributed memory).

The transformation consists of four parts:

1. Start up PySpark (should still be open from section 5.2.2) and load the Spark and Hive context.
2. Read and parse the .CSV file.
3. Split the header line from the data.
4. Clean the data.

Okay, onto business. The following listing shows the code implementation in the PySpark console.

Listing 5.3 Connecting to Apache Spark

```
from pyspark import SparkContext
from pyspark.sql import HiveContext
#sc = SparkContext()
sqlContext = HiveContext(sc)
data = sc.textFile("/chapter5/LoanStats3d.csv")
```

Imports Spark context --> not necessary when working directly in PySpark.

Imports Hive context.

In the PySpark session, the Spark context is automatically present. In other cases (Zeppelin notebook) you'll need to create this explicitly.

Creates Hive context.

Loads in data set from Hadoop directory.

```
parts = data.map(lambda r:r.split(','))
firstline = parts.first()
datalines = parts.filter(lambda x:x != firstline)
def cleans(row):
        row [7] = str(float(row [7][:-1])/100)
        return [s.encode('utf8').replace(r"_"," ").lower() for s in row]
datalines = datalines.map(lambda x: cleans(x))
```

Column 8 (index = 7) has % formatted numbers. We don't need that % sign.

Executes data cleaning line by line.

Encodes everything in utf8, replaces underscores with spaces, and lowercases everything.

Grabs first line.

Cleaning function will use power of Spark to clean data. The input of this function will be a line of data.

Splits data set with comma (,) delimiter. This is the end of line delimiter for this file.

Grabs all lines but first line, because first line is only variable names.

Let's dive a little further into the details for each step.

Step 1: Starting up Spark in interactive mode and loading the context

The Spark context import isn't required in the PySpark console because a context is readily available as variable sc. You might have noticed this is also mentioned when opening PySpark; check out figure 5.9 in case you overlooked it. We then load a Hive context to enable us to work interactively with Hive. If you work interactively with Spark, the Spark and Hive contexts are loaded automatically, but if you want to use it in batch mode you need to load it manually. To submit the code in batch you would use the `spark-submit filename.py` command on the Horton Sandbox command line.

```
from pyspark import SparkContext
from pyspark.sql import HiveContext
sc = SparkContext()
sqlContext = HiveContext(sc)
```

With the environment set up, we're ready to start parsing the .CSV file.

Step 2: Reading and parsing the .CSV file

Next we read the file from the Hadoop file system and split it at every comma we encounter. In our code the first line reads the .CSV file from the Hadoop file system. The second line splits every line when it encounters a comma. Our .CSV parser is naïve by design because we're learning about Spark, but you can also use the .CSV package to help you parse a line more correctly.

```
data = sc.textFile("/chapter5/LoanStats3d.csv")
parts = data.map(lambda r:r.split(','))
```

Notice how similar this is to a functional programming approach. For those who've never encountered it, you can naïvely read `lambda r:r.split(',')` as "for every input r (a row in this case), split this input r when it encounters a comma." As in this case, "for every input" means "for every row," but you can also read it as "split every row by a comma." This functional-like syntax is one of my favorite characteristics of Spark.

Step 3: Split the header line from the data

To separate the header from the data, we read in the first line and retain every line that's not similar to the header line:

```
firstline = parts.first()
datalines = parts.filter(lambda x:x != firstline)
```

Following the best practices in big data, we wouldn't have to do this step because the first line would already be stored in a separate file. In reality, .CSV files do often contain a header line and you'll need to perform a similar operation before you can start cleaning the data.

Step 4: Clean the data

In this step we perform basic cleaning to enhance the data quality. This allows us to build a better report.

After the second step, our data consists of arrays. We'll treat every input for a lambda function as an array now and return an array. To ease this task, we build a helper function that cleans. Our cleaning consists of reformatting an input such as "10,4%" to 0.104 and encoding every string as utf-8, as well as replacing underscores with spaces and lowercasing all the strings. The second line of code calls our helper function for every line of the array.

```
def cleans(row):
        row [7] = str(float(row [7][:-1])/100)
        return [s.encode('utf8').replace(r"_"," ").lower() for s in row]
datalines = datalines.map(lambda x: cleans(x))
```

Our data is now prepared for the report, so we need to make it available for our reporting tools. Hive is well suited for this, because many reporting tools can connect to it. Let's look at how to accomplish this.

Save the data in Hive

To store data in Hive we need to complete two steps:

1. Create and register metadata.
2. Execute SQL statements to save data in Hive.

In this section, we'll once again execute the next piece of code in our beloved PySpark shell, as shown in the following listing.

Listing 5.4 Storing data in Hive (full)

Creates metadata: the Spark SQL StructField function represents a field in a StructType. The StructField object is comprised of three fields: name (a string), dataType (a DataType), and "nullable" (a boolean). The field of name is the name of a StructField. The field of dataType specifies the data type of a StructField. The field of nullable specifies if values of a StructField can contain None values.

Imports SQL data types.

```
from pyspark.sql.types import *
fields = [StructField(field_name,StringType(),True) for field_name in
     firstline]
schema = StructType(fields)
schemaLoans = sqlContext.createDataFrame(datalines, schema)
schemaLoans.registerTempTable("loans")
```

Creates data frame from data (datalines) and data schema (schema).

Registers it as a table called loans.

StructType function creates the data schema. A StructType object requires a list of StructFields as input.

```
sqlContext.sql("drop table if exists LoansByTitle")
sql = '''create table LoansByTitle stored as parquet as select title,
     count(1) as number from loans group by title order by number desc'''
sqlContext.sql(sql)

sqlContext.sql('drop table if exists raw')
sql = '''create table raw stored as parquet as select title,
     emp_title,grade,home_ownership,int_rate,recoveries,
     collection_recovery_fee,loan_amnt,term from loans'''
```

Drops table (in case it already exists) and stores a subset of raw data in Hive.

Drops table (in case it already exists), summarizes, and stores it in Hive. LoansByTitle represents the sum of loans by job title.

Let's drill deeper into each step for a bit more clarification.

Step 1: Create and register metadata

Many people prefer to use SQL when they work with data. This is also possible with Spark. You can even read and store data in Hive directly as we'll do. Before you can do that, however, you'll need to create metadata that contains a column name and column type for every column.

The first line of code is the imports. The second line parses the field name and the field type and specifies if a field is mandatory. The `StructType` represents rows as an array of structfields. Then you place it in a dataframe that's registered as a (temporary) table in Hive.

```
from pyspark.sql.types import *
fields = [StructField(field_name,StringType(),True) for field_name in firstline]
schema = StructType(fields)
schemaLoans = sqlContext.createDataFrame(datalines, schema)
schemaLoans.registerTempTable("loans")
```

With the metadata ready, we're now able to insert the data into Hive.

Step 2: Execute queries and store table in Hive

Now we're ready to use a SQL-dialect on our data. First we'll make a summary table that counts the number of loans per purpose. Then we store a subset of the cleaned raw data in Hive for visualization in Qlik.

Executing SQL-like commands is as easy as passing a string that contains the SQL-command to the `sqlContext.sql` function. Notice that we aren't writing pure SQL because we're communicating directly with Hive. Hive has its own SQL-dialect called HiveQL. In our SQL, for instance, we immediately tell it to store the data as a Parquet file. Parquet is a popular big data file format.

```
sqlContext.sql("drop table if exists LoansByTitle")
sql = '''create table LoansByTitle stored as parquet as select title,
     count(1) as number from loans group by title order by number desc'''
sqlContext.sql(sql)

sqlContext.sql('drop table if exists raw')
sql = '''create table raw stored as parquet as select title,
     emp_title,grade,home_ownership,int_rate,recoveries,collection_recovery_f
     ee,loan_amnt,term from loans'''
sqlContext.sql(sql)
```

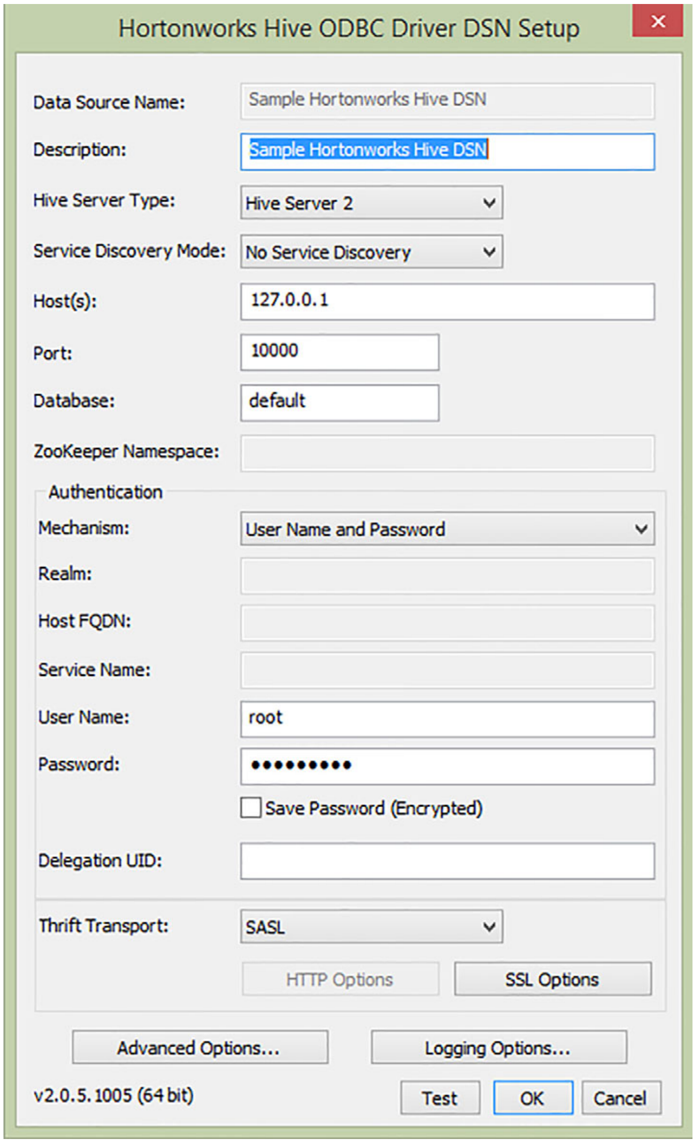
With the data stored in Hive, we can connect our visualization tools to it.

5.2.4 Step 4: Data exploration & Step 6: Report building

We'll build an interactive report with Qlik Sense to show to our manager. Qlik Sense can be downloaded from http://www.qlik.com/try-or-buy/download-qlik-sense after subscribing to their website. When the download begins you will be redirected to a page containing several informational videos on how to install and work with Qlik Sense. It's recommended to watch these first.

We use the Hive ODBC connector to read data from Hive and make it available for Qlik. A tutorial on installing ODBC connectors in Qlik is available. For major operating systems, this can be found at http://hortonworks.com/hdp/addons/.

NOTE In Windows, this might not work out of the box. Once you install the ODBC, make sure to check your Windows ODBC manager (CTRL+F and look for ODBC). In the manager, go to "System-DSN" and select the "Sample Hive Hortonworks DSN". Make sure your settings are correct (as shown in figure 5.11) or Qlik won't connect to the Hortonworks Sandbox.



Hortonworks Hive ODBC Driver DSN Setup

Data Source Name:	Sample Hortonworks Hive DSN
Description:	Sample Hortonworks Hive DSN
Hive Server Type:	Hive Server 2
Service Discovery Mode:	No Service Discovery
Host(s):	127.0.0.1
Port:	10000
Database:	default
ZooKeeper Namespace:	

Authentication

Mechanism:	User Name and Password
Realm:	
Host FQDN:	
Service Name:	
User Name:	root
Password:	•••••••••
	☐ Save Password (Encrypted)
Delegation UID:	
Thrift Transport:	SASL

Figure 5.11 Windows Hortonworks ODBC configuration

Let's hope you didn't forget your Sandbox password; as you can see in figure 5.11, you need it again.

Now open Qlik Sense. If installed in Windows you should have gotten the option to place a shortcut to the .exe on your desktop. Qlik isn't freeware; it's a commercial product with a bait version for single customers, but it will suffice for now. In the last chapter we'll create a dashboard using free JavaScript libraries.

Qlik can either take the data directly into memory or make a call every time to Hive. We've chosen the first method because it works faster.

This part has three steps:

1. Load data inside Qlik with an ODBC connection.
2. Create the report.
3. Explore data.

Let start with the first step, loading data into Qlik.

Step 1: Load data in Qlik

When you start Qlik Sense it will show you a welcome screen with the existing reports (called apps), as shown in figure 5.12.

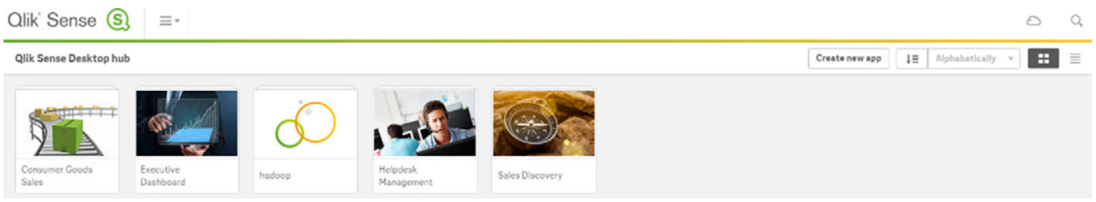
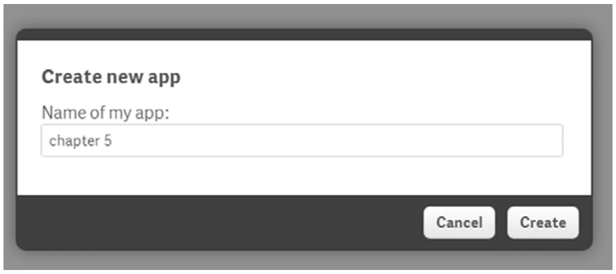


Figure 5.12 The Qlik Sense welcome screen

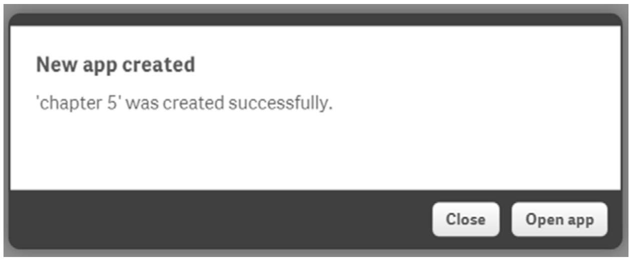
To start a new app, click on the *Create new app* button on the right of the screen, as shown in figure 5.13. This opens up a new dialog box. Enter "chapter 5" as the new name of our app.



Name of my app: chapter 5

Figure 5.13 The Create new app message box

A confirmation box appears (figure 5.14) if the app is created successfully.

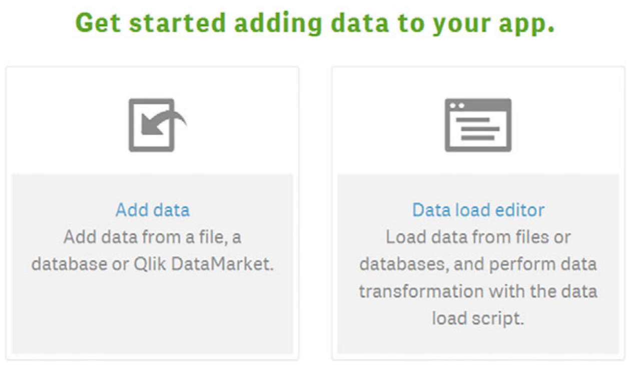


New app created

'chapter 5' was created successfully.

Figure 5.14 A box confirms that the app was created successfully.

Click on the Open app button and a new screen will prompt you to add data to the application (figure 5.15).



Get started adding data to your app.

Add data
Add data from a file, a database or Qlik DataMarket.

Data load editor
Load data from files or databases, and perform data transformation with the data load script.

Figure 5.15 A start-adding-data screen pops up when you open a new app.

Click on the Add data button and choose ODBC as a data source (figure 5.16).

In the next screen (figure 5.17) select User DSN, Hortonworks, and specify the root as username and hadoop as a password (or the new one you gave when logging into the Sandbox for the first time).

NOTE The Hortonworks option doesn't show up by default. You need to install the HDP 2.3 ODBC connector for this option to appear (as stated before). If you haven't succeeded in installing it at this point, clear instructions for this can be found at https://blogs.perficient.com/multi-shoring/blog/2015/09/29/how-to-connect-hortonworks-hive-from-qlikview-with-odbc-driver/.

Click on the arrow pointing to the right to go to the next screen.

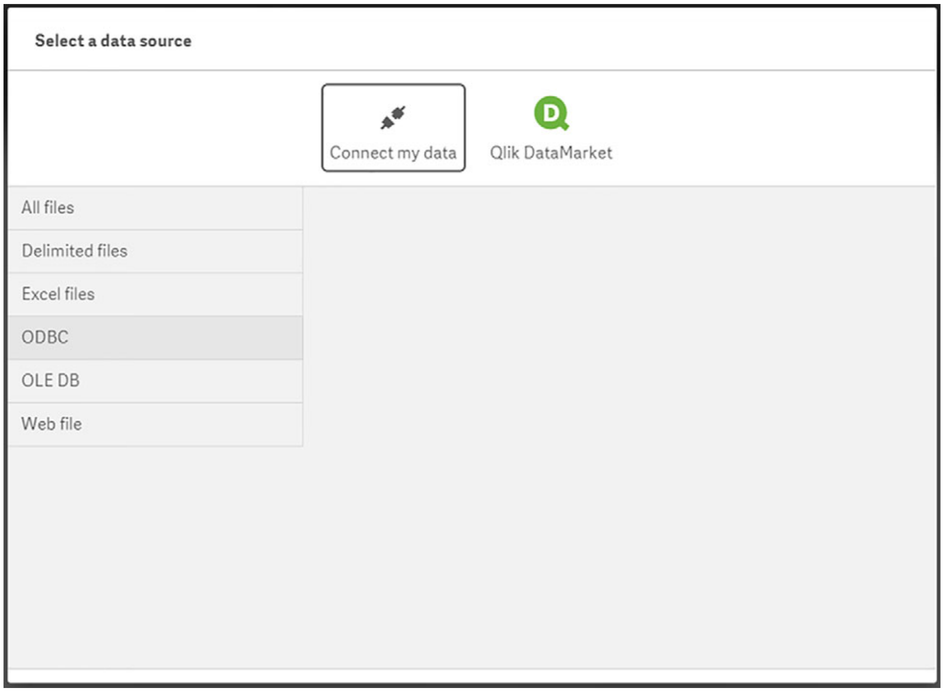
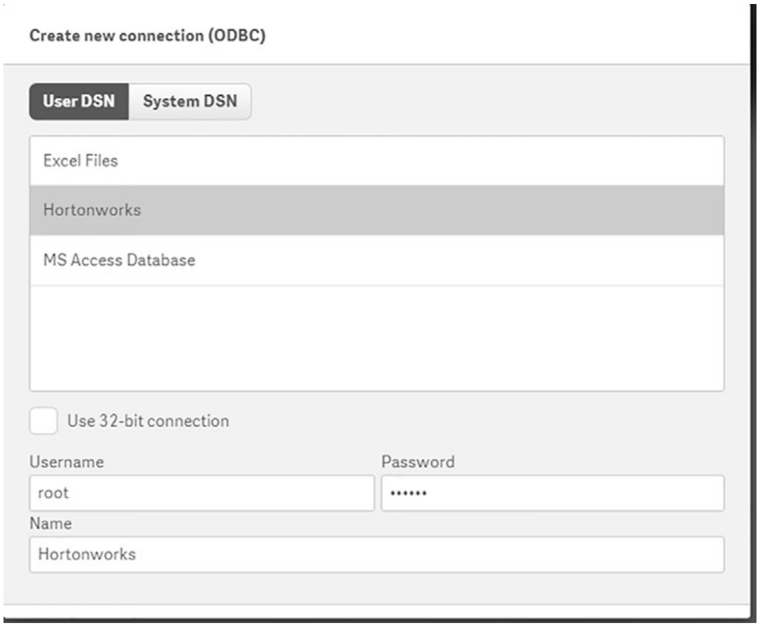
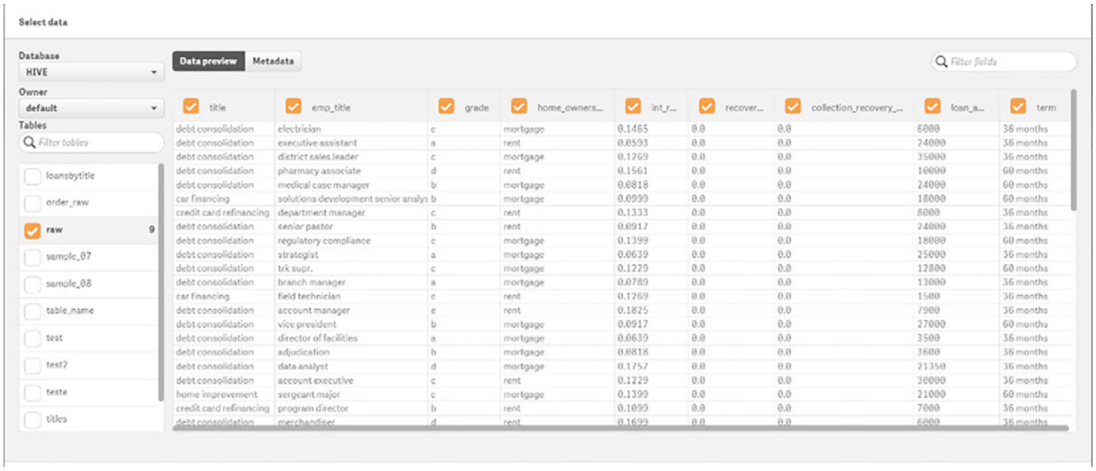


Figure 5.16 Choose ODBC as data source in the Select a data source screen



Username: root
Password: ••••••
Name: Hortonworks

Figure 5.17 Choose Hortonworks on the User DSN and specify the username and password.


Select data

Database: HIVE

Owner: default

Tables

Filter tables

- loansbytitle
- order_raw
- raw 9
- sample_07
- sample_08
- table_name
- test
- test2
- teste
- titles

Data preview | Metadata

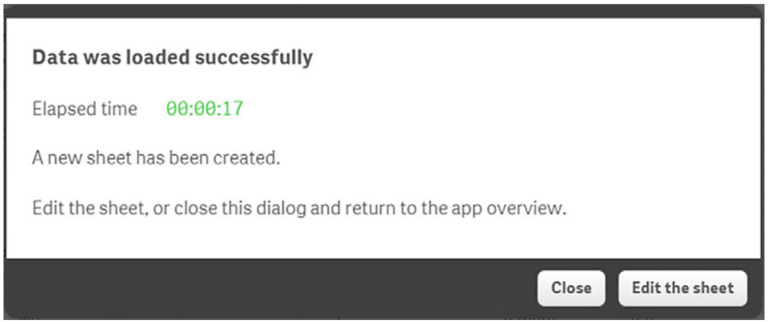
Filter fields

title	emp_title	grade	home_owners...	int_r...	recover...	collection_recovery_...	loan_a...	term
debt consolidation	electrician	c	mortgage	0.1465	0.0	0.0	6000	36 months
debt consolidation	executive assistant	a	rent	0.0593	0.0	0.0	24000	36 months
debt consolidation	district sales leader	c	mortgage	0.1269	0.0	0.0	35000	36 months
debt consolidation	pharmacy associate	d	rent	0.1561	0.0	0.0	10000	60 months
debt consolidation	medical case manager	b	mortgage	0.0818	0.0	0.0	24000	60 months
car financing	solutions development senior analys	b	mortgage	0.0999	0.0	0.0	18000	60 months
credit card refinancing	department manager	c	rent	0.1333	0.0	0.0	8000	36 months
debt consolidation	senior pastor	b	rent	0.0917	0.0	0.0	24000	36 months
debt consolidation	regulatory compliance	c	mortgage	0.1399	0.0	0.0	18000	60 months
debt consolidation	strategist	a	mortgage	0.0639	0.0	0.0	25000	36 months
debt consolidation	trk supr.	c	mortgage	0.1229	0.0	0.0	12800	60 months
debt consolidation	branch manager	a	mortgage	0.0789	0.0	0.0	13000	36 months
car financing	field technician	c	rent	0.1269	0.0	0.0	1500	36 months
debt consolidation	account manager	e	rent	0.1825	0.0	0.0	7900	36 months
debt consolidation	vice president	b	mortgage	0.0917	0.0	0.0	27000	60 months
debt consolidation	director of facilities	a	mortgage	0.0639	0.0	0.0	3500	36 months
debt consolidation	adjudication	b	mortgage	0.0818	0.0	0.0	3600	36 months
debt consolidation	data analyst	d	mortgage	0.1757	0.0	0.0	21350	36 months
debt consolidation	account executive	c	rent	0.1229	0.0	0.0	30000	36 months
home improvement	sergeant major	c	mortgage	0.1399	0.0	0.0	21000	60 months
credit card refinancing	program director	b	rent	0.1099	0.0	0.0	7000	36 months
debt consolidation	merchandiser	d	rent	0.1699	0.0	0.0	6000	36 months

Figure 5.18 Hive interface raw data column overview

Choose the Hive data, and default as user in the next screen (figure 5.18). Select raw as Tables to select and select every column for import; then click the button Load and Finish to complete this step.

After this step, it will take a few seconds to load the data in Qlik (figure 5.19).


Data was loaded successfully

Elapsed time 00:00:17

A new sheet has been created.

Edit the sheet, or close this dialog and return to the app overview.

Figure 5.19 A confirmation that the data is loaded in Qlik

Step 2: Create the report

Choose Edit the sheet to start building the report. This will add the report editor (figure 5.20).

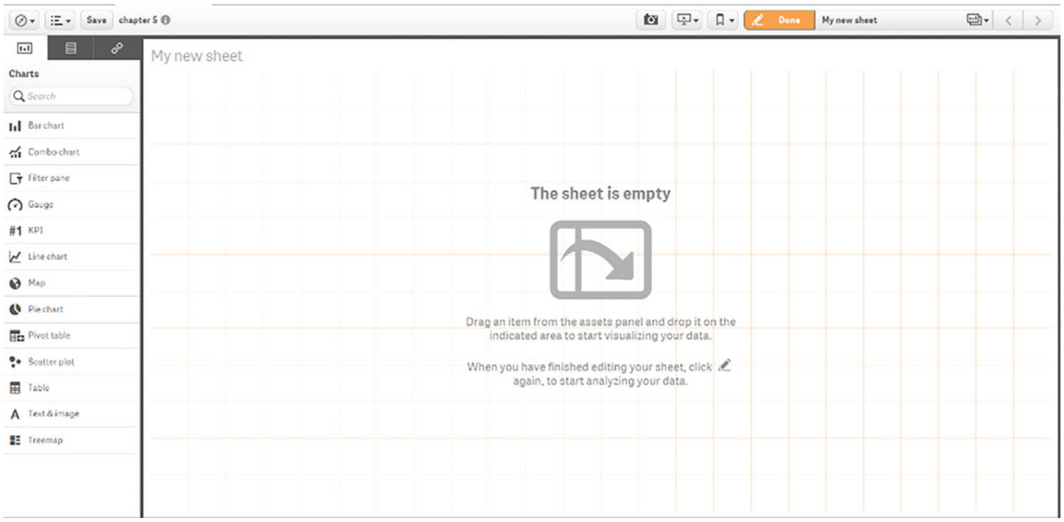

Save
chapter 5
Done
My new sheet
Charts
Search
Bar chart
Combo chart
Filter pane
Gauge
KPI
Line chart
Map
Pie chart
Pivot table
Scatter plot
Table
Text & image
Treemap
My new sheet
The sheet is empty
Drag an item from the assets panel and drop it on the indicated area to start visualizing your data.
When you have finished editing your sheet, click again, to start analyzing your data.

Figure 5.20 An editor screen for reports opens

Substep 1: Adding a selection filter to the report The first thing we'll add to the report is a selection box that shows us why each person wants a loan. To achieve this, drop the title measure from the left asset panel on the report pane and give it a comfortable size and position (figure 5.21). Click on the Fields table so you can drag and drop fields.

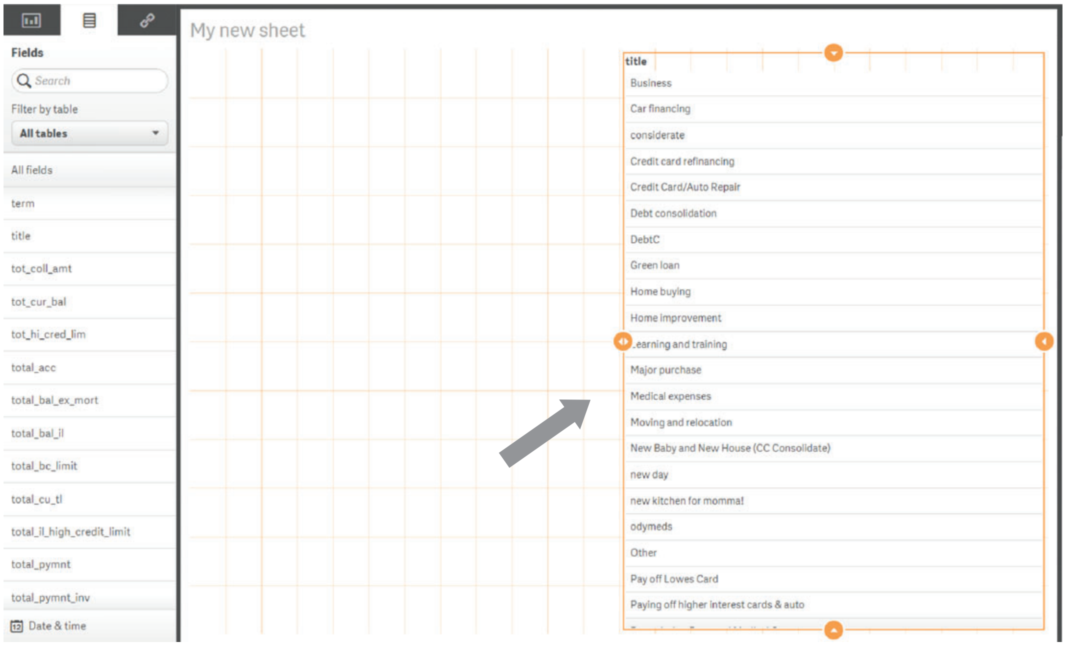


Figure 5.21 Drag the title from the left Fields pane to the report pane.

Substep 2: Adding a KPI to the report A KPI chart shows an aggregated number for the total population that's selected. Numbers such as the average interest rate and the total number of customers are shown in this chart (figure 5.22).

Adding a KPI to a report takes four steps, as listed below and shown in figure 5.23.

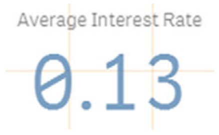
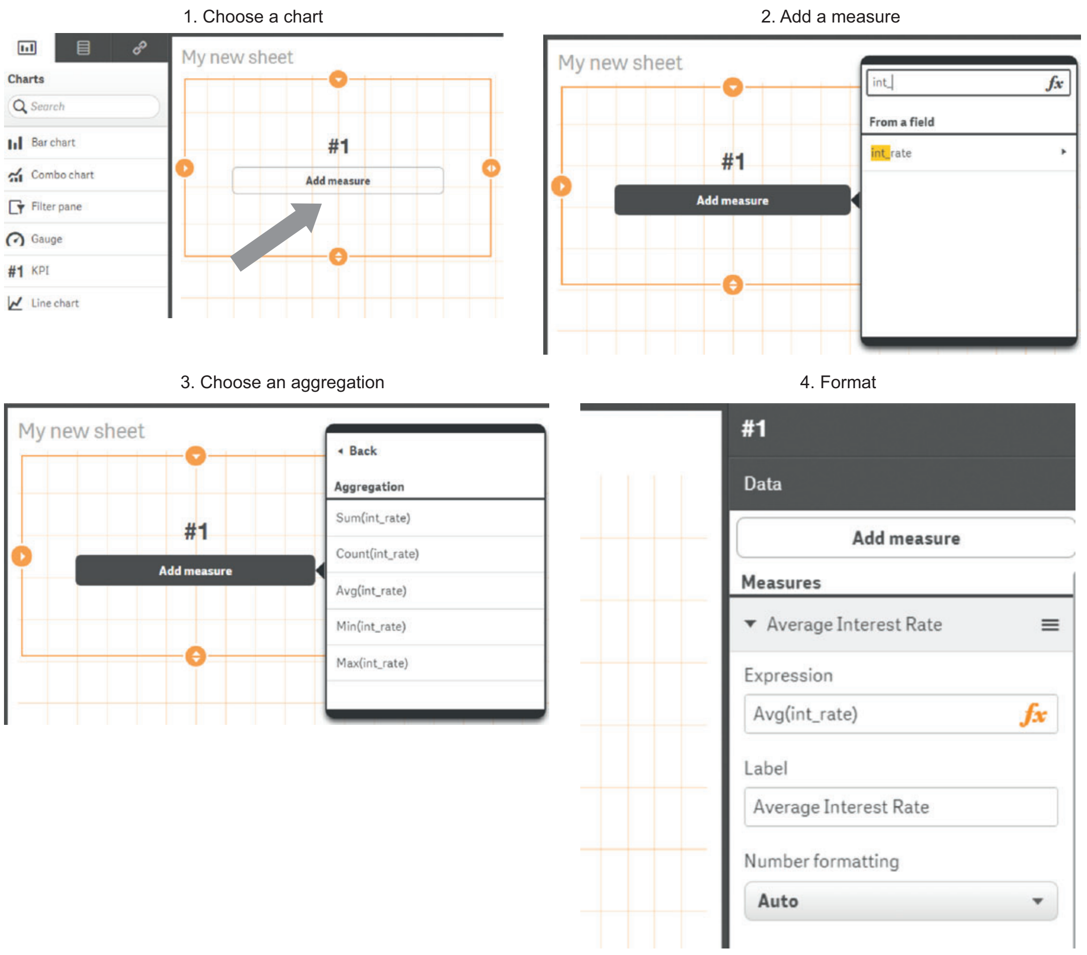


Figure 5.22 An example of a KPI chart

1. *Choose a chart*—Choose KPI as the chart and place it on the report screen; resize and position to your liking.
2. *Add a measure*—Click the add measure button inside the chart and select int_rate.
3. *Choose an aggregation method*—Avg(int_rate).
4. *Format the chart*—On the right pane, fill in average interest rate as Label.



Expression: Avg(int_rate)
Label: Average Interest Rate
Number formatting: Auto

Figure 5.23 The four steps to add a KPI chart to a Qlik report

In total we'll add four KPI charts to our report, so you'll need to repeat these steps for the following KPI's:

- Average interest rate
- Total loan amount
- Average loan amount
- Total recoveries

Substep 3: Adding bar charts to our report Next we'll add four bar charts to the report. These will show the different numbers for each risk grade. One bar chart will explain the average interest rate per risk group, and another will show us the total loan amount per risk group (figure 5.24).

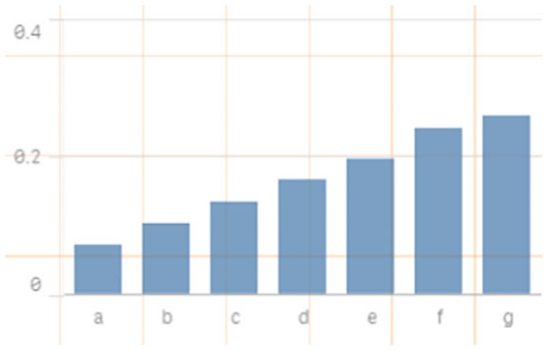


Figure 5.24 An example of a bar chart

Adding a bar chart to a report takes five steps, as listed below and shown in figure 5.25.

1. *Choose a chart*—Choose bar chart as the chart and place it on the report screen; resize and position to your liking.
2. *Add a measure*—Click the Add measure button inside the chart and select int_rate.
3. *Choose an aggregation method*—Avg(int_rate).
4. *Add a dimension*—Click Add dimension, and choose grade as the dimension.
5. *Format the chart*—On the right pane, fill in average interest rate as Label.

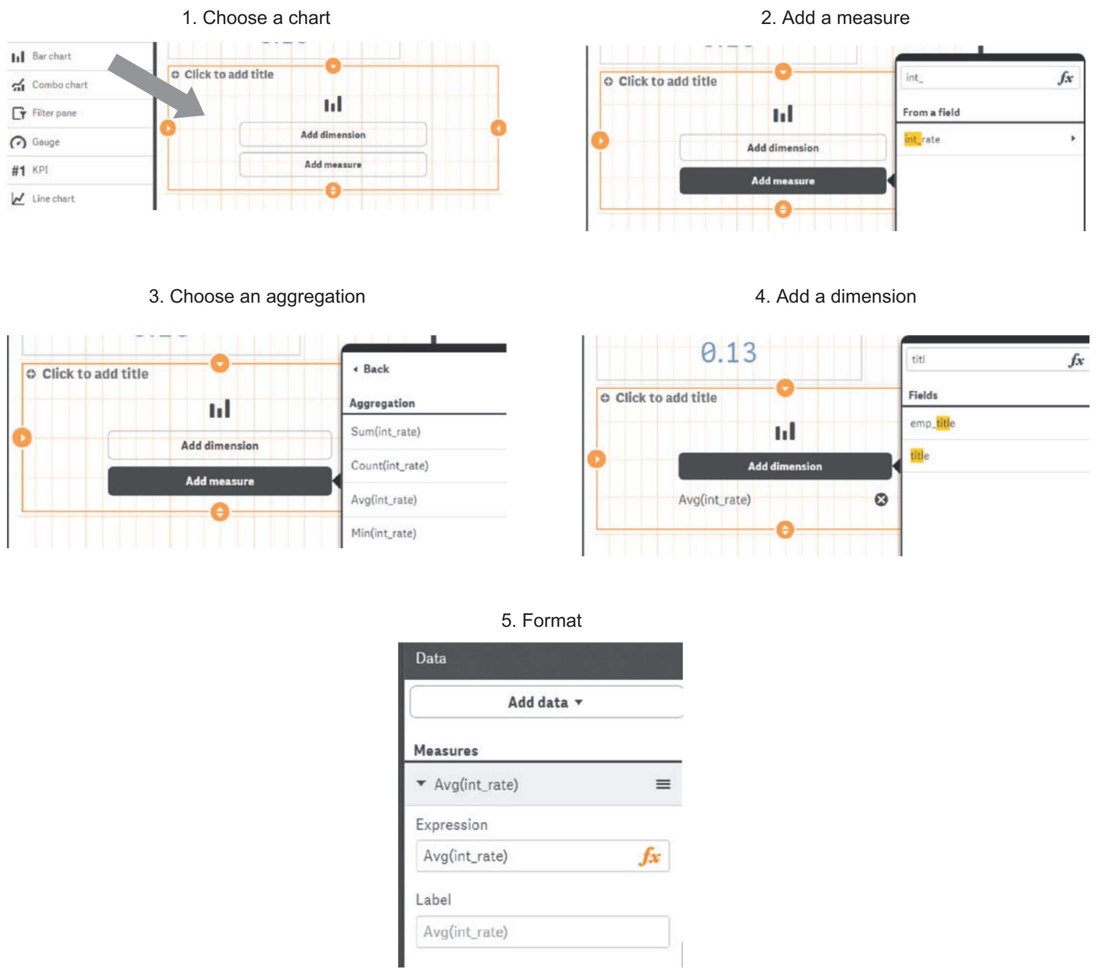
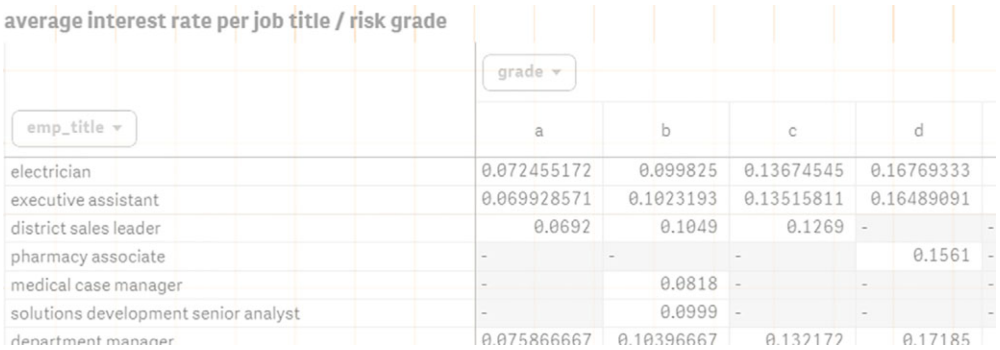


Figure 5.25 Adding a bar chart takes five steps.

Repeat this procedure for the following dimension and measure combinations:

- Average interest rate per grade
- Average loan amount per grade
- Total loan amount per grade
- Total recoveries per grade

Substep 4: Adding a cross table to the report Suppose you want to know the average interest rate paid by directors of risk group C. In this case you want to get a measure (interest rate) for a combination of two dimensions (job title and risk grade). This can be achieved with a pivot table such as in figure 5.26.


average interest rate per job title / risk grade

emp_title	grade a	b	c	d	
electrician	0.072455172	0.099825	0.13674545	0.16769333	
executive assistant	0.069928571	0.1023193	0.13515811	0.16489091	
district sales leader	0.0692	0.1049	0.1269	-	-
pharmacy associate	-	-	-	0.1561	-
medical case manager	-	0.0818	-	-	-
solutions development senior analyst	-	0.0999	-	-	-
department manager	0.075866667	0.10396667	0.132172	0.17185	

Figure 5.26 An example of a pivot table, showing the average interest rate paid per job title/risk grade combination

Adding a pivot table to a report takes six steps, as listed below and shown in figure 5.27.

1. *Choose a chart*—Choose pivot table as the chart and place it on the report screen; resize and position to your liking.
2. *Add a measure*—Click the Add measure button inside the chart and select int_rate.
3. *Choose an aggregation method*—Avg(int_rate).
4. *Add a row dimension*—Click Add dimension, and choose emp_title as the dimension.
5. *Add a column dimension*—Click Add data, choose column, and select grade.
6. *Format the chart*—On the right pane, fill in average interest rate as Label.

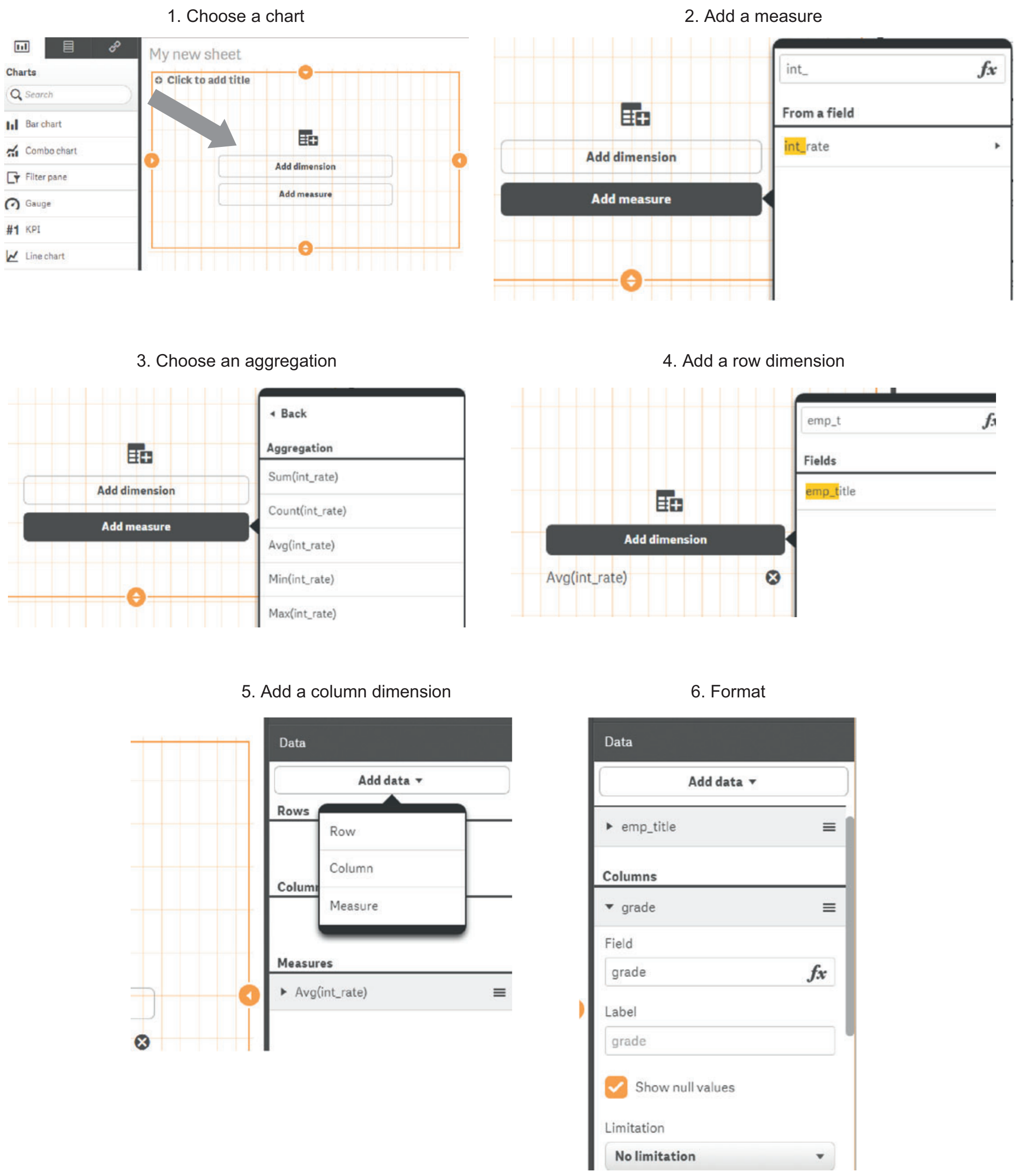
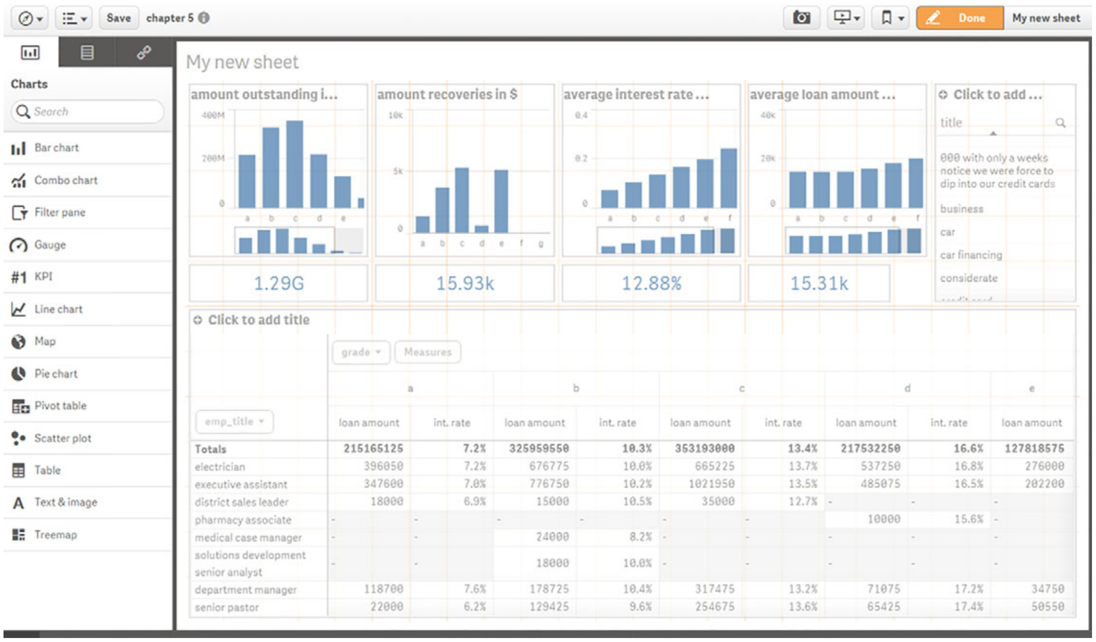


Figure 5.27 Adding a pivot table takes six steps.



	a		b		c		d		e
	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount
Totals	**215165125**	**7.2%**	**325959550**	**10.3%**	**353193000**	**13.4%**	**217532250**	**16.6%**	**127818575**
electrician	396050	7.2%	676775	10.0%	665225	13.7%	537250	16.8%	276000
executive assistant	347600	7.0%	776750	10.2%	1021950	13.5%	485075	16.5%	202200
district sales leader	18000	6.9%	15000	10.5%	35000	12.7%	-	-	-
pharmacy associate	-	-	-	-	-	-	10000	15.6%	-
medical case manager	-	-	24000	8.2%	-	-	-	-	-
solutions development senior analyst	-	-	18000	10.0%	-	-	-	-	-
department manager	118700	7.6%	178725	10.4%	317475	13.2%	71075	17.2%	34750
senior pastor	22000	6.2%	129425	9.6%	254675	13.6%	65425	17.4%	50550

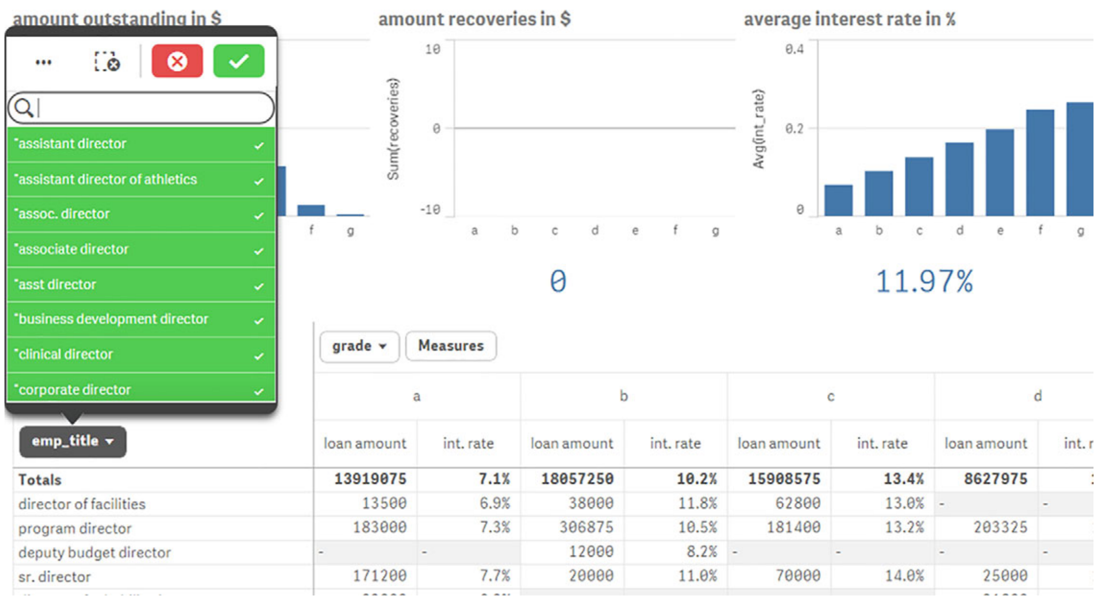
Figure 5.28 The end result in edit mode

After resizing and repositioning, you should achieve a result similar to figure 5.28. Click the Done button on the left and you're ready to explore the data.

Step 3: Explore the data

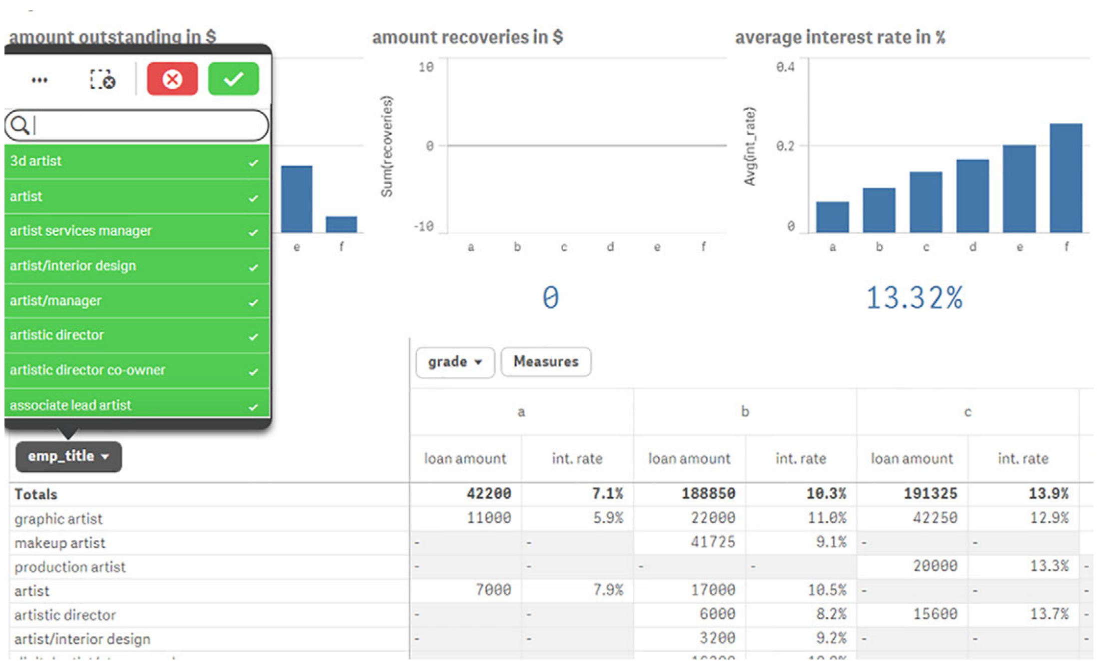
The result is an interactive graph that updates itself based on the selections you make. Why don't you try to look for the information from directors and compare them to artists? To achieve this, hit the emp_title in the pivot table and type director in the search field. The result looks like figure 5.29. In the same manner, we can look at the artists, as shown in figure 5.30. Another interesting insight comes from comparing the rating for home-buying purposes with debt consolidation purposes.

We finally did it: We created the report our manager craves, and in the process we opened the door for other people to create their own reports using this data. An interesting next step for you to ponder on would be to use this setup to find those people likely to default on their debt. For this you can use the Spark Machine learning capabilities driven by online algorithms like the ones demonstrated in chapter 4.



	a		b		c		d	
	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate	loan amount	int. r
Totals	**13919075**	**7.1%**	**18057250**	**10.2%**	**15908575**	**13.4%**	**8627975**	
director of facilities	13500	6.9%	38000	11.8%	62800	13.0%	-	-
program director	183000	7.3%	306875	10.5%	181400	13.2%	203325	
deputy budget director	-	-	12000	8.2%	-	-	-	-
sr. director	171200	7.7%	20000	11.0%	70000	14.0%	25000	

Figure 5.29 When we select directors, we can see that they pay an average rate of 11.97% for a loan.



	a		b		c	
	loan amount	int. rate	loan amount	int. rate	loan amount	int. rate
Totals	**42200**	**7.1%**	**188850**	**10.3%**	**191325**	**13.9%**
graphic artist	11000	5.9%	22000	11.0%	42250	12.9%
makeup artist	-	-	41725	9.1%	-	-
production artist	-	-	-	-	20000	13.3%
artist	7000	7.9%	17000	10.5%	-	-
artistic director	-	-	6000	8.2%	15600	13.7%
artist/interior design	-	-	3200	9.2%	-	-

Figure 5.30 When we select artists, we see that they pay an average interest rate of 13.32% for a loan.

In this chapter we got a hands-on introduction to the Hadoop and Spark frameworks. We covered a lot of ground, but be honest, Python makes working with big data technologies dead easy. In the next chapter we'll dig deeper into the world of NoSQL databases and come into contact with more big data technologies.

5.3 Summary

In this chapter you learned that

- Hadoop is a framework that enables you to store files and distribute calculations amongst many computers.
- Hadoop hides all the complexities of working with a cluster of computers for you.
- An ecosystem of applications surrounds Hadoop and Spark, ranging from databases to access control.
- Spark adds a shared memory structure to the Hadoop Framework that's better suited for data science work.
- In the chapter case study we used PySpark (a Python library) to communicate with Hive and Spark from Python. We used the pywebhdfs Python library to work with the Hadoop library, but you could do as well using the OS command line.
- It's easy to connect a BI tool such as Qlik to Hadoop.