9 Data visualization to the end user

This chapter covers

- Considering options for data visualization for your end users
- Setting up a basic Crossfilter MapReduce application
- Creating a dashboard with dc.js
- Working with dashboard development tools

APPLICATION FOCUSED CHAPTER You'll notice quickly this chapter is certainly different from chapters 3 to 8 in that the focus here lies on step 6 of the data science process. More specifically, what we want to do here is create a small data science application. Therefore, we won't follow the data science process steps here. The data used in the case study is only partly real but functions as data flowing from either the data preparation or data modeling stage. Enjoy the ride.

Often, data scientists must deliver their new insights to the end user. The results can be communicated in several ways:

- *A one-time presentation*—Research questions are one-shot deals because the business decision derived from them will bind the organization to a certain

course for many years to come. Take, for example, company investment decisions: *Do we distribute our goods from two distribution centers or only one? Where do they need to be located for optimal efficiency?* When the decision is made, the exercise may not be repeated until you've retired. In this case, the results are delivered as a report with a presentation as the icing on the cake.

- *A new viewport on your data*—The most obvious example here is customer segmentation. Sure, the segments themselves will be communicated via reports and presentations, but in essence they form tools, not the end result itself. When a clear and relevant customer segmentation is discovered, it can be fed back to the database as a new dimension on the data from which it was derived. From then on, people can make their own reports, such as how many products were sold to each segment of customers.
- *A real-time dashboard*—Sometimes your task as a data scientist doesn't end when you've discovered the new information you were looking for. You can send your information back to the database and be done with it. But when other people start making reports on this newly discovered gold nugget, they might interpret it incorrectly and make reports that don't make sense. As the data scientist who discovered this new information, you must set the example: make the first refreshable report so others, mainly reporters and IT, can understand it and follow in your footsteps. Making the first dashboard is also a way to shorten the delivery time of your insights to the end user who wants to use it on an everyday basis. This way, at least they already have something to work with until the reporting department finds the time to create a permanent report on the company's reporting software.

You might have noticed that a few important factors are at play:

- What *kind of decision* are you supporting? Is it a strategic or an operational one? Strategic decisions often only require you to analyze and report once, whereas operational decisions require the report to be refreshed regularly.
- *How big is your organization?* In smaller ones you'll be in charge of the entire cycle: from data gathering to reporting. In bigger ones a team of reporters might be available to make the dashboards for you. But even in this last situation, delivering a prototype dashboard can be beneficial because it presents an example and often shortens delivery time.

Although the entire book is dedicated to generating insights, in this last chapter we'll focus on delivering an operational dashboard. Creating a presentation to promote your findings or presenting strategic insights is out of the scope of this book.

9.1 Data visualization options

You have several options for delivering a dashboard to your end users. Here we'll focus on a single option, and by the end of this chapter you'll be able to create a dashboard yourself.

This chapter's case is that of a hospital pharmacy with a stock of a few thousand medicines. The government came out with a new norm to all pharmacies: all medicines should be checked for their sensitivity to light and be stored in new, special containers. One thing the government didn't supply to the pharmacies was an actual list of light-sensitive medicines. This is no problem for you as a data scientist because every medicine has a patient information leaflet that contains this information. You distill the information with the clever use of text mining and assign a "light sensitive" or "not light sensitive" tag to each medicine. This information is then uploaded to the central database. In addition, the pharmacy needs to know how many containers would be necessary. For this they give you access to the pharmacy stock data. When you draw a sample with only the variables you require, the data set looks like figure 9.1 when opened in Excel.

	A	B	C	D	E	F
1	MedName	LightSen	Date	StockOut	StockIn	Stock
2	Acupan 30 mg	No	1/01/2015	-8	150	142
3	Acupan 30 mg	No	2/01/2015	-6	5	141
4	Acupan 30 mg	No	3/01/2015	-2	0	139
5	Acupan 30 mg	No	4/01/2015	0	5	144
6	Acupan 30 mg	No	5/01/2015	-8	0	136
7	Acupan 30 mg	No	6/01/2015	-1	0	135
8	Acupan 30 mg	No	7/01/2015	-1	15	149
9	Acupan 30 mg	No	8/01/2015	-10	10	149
10	Acupan 30 mg	No	9/01/2015	-8	15	156

Figure 9.1 Pharmacy medicines data set opened in Excel: the first 10 lines of stock data are enhanced with a light-sensitivity variable

As you can see, the information is time-series data for an entire year of stock movement, so every medicine thus has 365 entries in the data set. Although the case study is an existing one and the medicines in the data set are real, the values of the other variables presented here were randomly generated, as the original data is classified. Also, the data set is limited to 29 medicines, a little more than 10,000 lines of data. Even though people do create reports using crossfilter.js (a Javascript MapReduce library) and dc.js (a Javascript dashboarding library) with more than a million lines of data, for the example's sake you'll use a fraction of this amount. Also, it's not recommended to load your entire database into the user's browser; the browser will freeze while loading, and if it's too much data, the browser will even crash. Normally data is precalculated on the server and parts of it are requested using, for example, a REST service.

To turn this data into an actual dashboard you have many options and you can find a short overview of the tools later in this chapter.

Among all the options, for this book we decided to go with dc.js, which is a crossbreed between the JavaScript MapReduce library Crossfilter and the data visualization library d3.js. Crossfilter was developed by Square Register, a company that handles payment transactions; it's comparable to PayPal but its focus is on mobile. Square

developed Crossfilter to allow their customers extremely speedy slice and dice on their payment history. Crossfilter is not the only JavaScript library capable of MapReduce processing, but it most certainly does the job, is open source, is free to use, and is maintained by an established company (Square). Example alternatives to Crossfilter are Map.js, Meguro, and Underscore.js. JavaScript might not be known as a data crunching language, but these libraries do give web browsers that extra bit of punch in case data does need to be handled in the browser. We won't go into how JavaScript can be used for massive calculations within collaborative distributed frameworks, but an army of dwarfs can topple a giant. If this topic interests you, you can read more about it at https://www.igvita.com/2009/03/03/collaborative-map-reduce-in-the-browser/ and at http://dyn.com/blog/browsers-vs-servers-using-javascript-for-number-crunching-theories/.

d3.js can safely be called the most versatile JavaScript data visualization library available at the time of writing; it was developed by Mike Bostock as a successor to his Protovis library. Many JavaScript libraries are built on top of d3.js.

NVD3, C3.js, xCharts, and Dimple offer roughly the same thing: an abstraction layer on top of d3.js, which makes it easier to draw simple graphs. They mainly differ in the type of graphs they support and their default design. Feel free to visit their websites and find out for yourself:

- *NVD3*—http://nvd3.org/
- *C3.js*—http://c3js.org/
- *xCharts*—http://tenxer.github.io/xcharts/
- *Dimple*—http://dimplejs.org/

Many options exist. So why dc.js?

The main reason: compared to what it delivers, an interactive dashboard where clicking one graph will create filtered views on related graphs, dc.js is surprisingly easy to set up. It's so easy that you'll have a working example by the end of this chapter. As a data scientist, you already put in enough time on your actual analysis; easy-to-implement dashboards are a welcome gift.

To get an idea of what you're about to create, you can go to the following website, http://dc-js.github.io/dc.js/, and scroll down to the NASDAQ example, shown in figure 9.2.

Click around the dashboard and see the graphs react and interact when you select and deselect data points. Don't spend too long though; it's time to create this yourself.

As stated before, dc.js has two big prerequisites: d3.js and crossfilter.js. d3.js has a steep learning curve and there are several books on the topic worth reading if you're interested in full customization of your visualizations. But to work with dc.js, no knowledge of it is required, so we won't go into it in this book. Crossfilter.js is another matter; you'll need to have a little grasp of this MapReduce library to get dc.js up and running on your data. But because the concept of MapReduce itself isn't new, this will go smoothly.

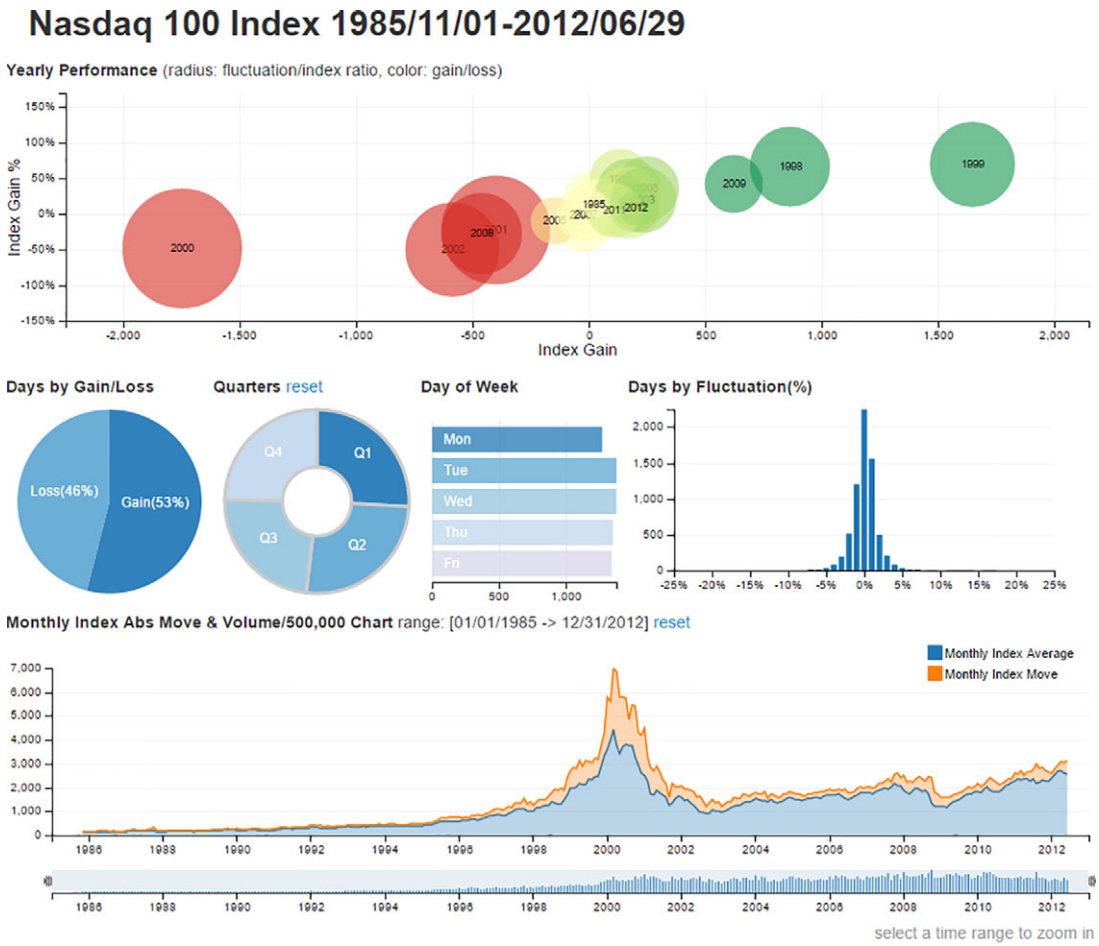

Figure 9.2 A dc.js interactive example on its official website

9.2 Crossfilter, the JavaScript MapReduce library

JavaScript isn't the greatest language for data crunching. But that didn't stop people, like the folks at Square, from developing MapReduce libraries for it. If you're dealing with data, every bit of speed gain helps. You don't want to send enormous loads of data over the internet or even your internal network though, for these reasons:

- Sending a bulk of data will tax the *network* to the point where it will bother other users.
- The *browser* is on the receiving end, and while loading in the data it will temporarily *freeze*. For small amounts of data this is unnoticeable, but when you start looking at 100,000 lines, it can become a visible lag. When you go over

1,000,000 lines, depending on the width of your data, your browser could give up on you.

Conclusion: it's a balance exercise. For the data you do send, there is a Crossfilter to handle it for you once it arrives in the browser. In our case study, the pharmacist requested the central server for stock data of 2015 for 29 medicines she was particularly interested in. We already took a look at the data, so let's dive into the application itself.

9.2.1 Setting up everything

It's time to build the actual application, and the ingredients of our small dc.js application are as follows:

- *JQuery*—To handle the interactivity
- *Crossfilter.js*—A MapReduce library and prerequisite to dc.js
- *d3.js*—A popular data visualization library and prerequisite to dc.js
- *dc.js*—The visualization library you will use to create your interactive dashboard
- *Bootstrap*—A widely used layout library you'll use to make it all look better

You'll write only three files:

- *index.html*—The HTML page that contains your application
- *application.js*—To hold all the JavaScript code you'll write
- *application.css*—For your own CSS

In addition, you'll need to run our code on an HTTP server. You could go through the effort of setting up a LAMP (Linux, Apache, MySQL, PHP), WAMP (Windows, Apache, MySQL, PHP), or XAMPP (Cross Environment, Apache, MySQL, PHP, Perl) server. But for the sake of simplicity we won't set up any of those servers here. Instead you can do it with a single Python command. Use your command-line tool (Linux shell or Windows CMD) and move to the folder containing your index.html (once it's there). You should have Python installed for other chapters of this book so the following command should launch a Python HTTP server on your localhost.

```
python -m SimpleHTTPServer
```

For Python 3.4

```
python -m http.server 8000
```

As you can see in figure 9.3, an HTTP server is started on localhost port 8000. In your browser this translates to "localhost:8000"; putting "0.0.0.0:8000" won't work.

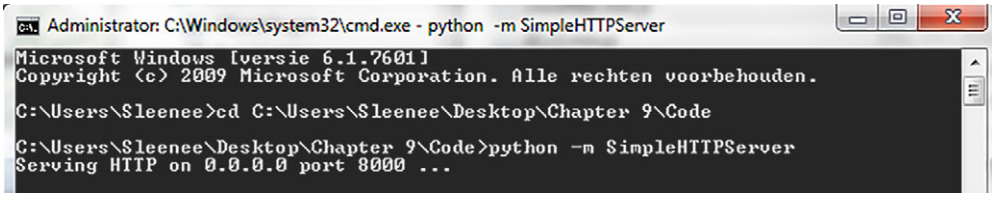

Figure 9.3 Starting up a simple Python HTTP server

Make sure to have all the required files available in the same folder as your index.html. You can download them from the Manning website or from their creators' websites.

- *dc.css and dc.min.js*—https://dc-js.github.io/dc.js/
- *d3.v3.min.js*—http://d3js.org/
- *crossfilter.min.js*—http://square.github.io/crossfilter/

Now we know how to run the code we're about to create, so let's look at the index.html page, shown in the following listing.

Listing 9.1 An initial version of index.html

```
<html>
<head>
    <title>Chapter 10.  Data Science Application</title>

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap
/3.3.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/
3.3.0/css/bootstrap-theme.min.css">

    <link rel="stylesheet" href="dc.css">
    <link rel="stylesheet" href="application.css">
</head>
<body>

    <main class='container'>
        <h1>Chapter 10:  Data Science Application</h1>
        <div class="row">
            <div class='col-lg-12'>
                <div id="inputtable" class="well well-sm"></div>
            </div>
        </div>
        <div class="row">
            <div class='col-lg-12'>
                <div id="filteredtable" class="well well-sm"></div>
            </div>
        </div>
    </main>
```

All CSS is loaded here.

Make sure to have dc.css downloaded from the Manning download page or from the dc website: https://dc-js.github.io/dc.js/. It must be present in the same folder as index.html file.

Main container incorporates everything visible to user.

```
    <script src="https://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.0/js
/bootstrap.min.js"></script>

    <script src="crossfilter.min.js"></script>
    <script src="d3.v3.min.js"></script>
    <script src="dc.min.js"></script>
    <script src="application.js"></script>
</body>
</html>
```

All Javascript is loaded here.

Make sure to have crossfilter.min.js, d3.v3.min.js, and dc.min.js downloaded from their websites or from the Manning website. Crossfilter: http://square.github.io/crossfilter/, d3.js: http://d3js.org/, dc.min.js: https://dc-js.github.io/dc.js/.

No surprises here. The header contains all the CSS libraries you'll use, so we'll load our JavaScript at the end of the HTML body. Using a JQuery onload handler, your application will be loaded when the rest of the page is ready. You start off with two table placeholders: one to show what your input data looks like, `<div id="input-table"></div>`, and the other one will be used with Crossfilter to show a filtered table, `<div id="filteredtable"></div>`. Several Bootstrap CSS classes were used, such as "*well*", "*container*", the Bootstrap grid system with "*row*" and "*col-xx-xx*", and so on. They make the whole thing look nicer but they aren't mandatory. More information on the Bootstrap CSS classes can be found on their website at http://getbootstrap.com/css/.

Now that you have your HTML set up, it's time to show your data onscreen. For this, turn your attention to the application.js file you created. First, we wrap the entire code "to be" in a JQuery onload handler.

```
$(function() {
    //All future code will end up in this wrapper
})
```

Now we're certain our application will be loaded only when all else is ready. This is important because we'll use JQuery selectors to manipulate the HTML. It's time to load in data.

```
d3.csv('medicines.csv',function(data) {
    main(data)
});
```

You don't have a REST service ready and waiting for you, so for the example you'll draw the data from a .csv file. This file is available for download on Manning's website. d3.js offers an easy function for that. After loading in the data you hand it over to your main application function in the d3.csv callback function.

Apart from the main function you have a `CreateTable` function, which you will use to…you guessed it…create your tables, as shown in the following listing.

Listing 9.2 The `CreateTable` function

```
var tableTemplate = $([
        "<table class='table table-hover table-condensed table-striped'>",
        "  <caption></caption>",
        "  <thead><tr/></thead>",
        "  <tbody></tbody>",
        "</table>"
    ].join('\n'));

    CreateTable = function(data,variablesInTable,title){
        var table = tableTemplate.clone();
        var ths = variablesInTable.map(function(v) { return $("<th>").text(v)
     });
        $('caption', table).text(title);
        $('thead tr', table).append(ths);
        data.forEach(function(row) {
            var tr = $("<tr>").appendTo($('tbody', table));
            variablesInTable.forEach(function(varName) {
                var val = row, keys = varName.split('.');
                keys.forEach(function(key) { val = val[key] });
                tr.append($("<td>").text(val));
            });
        });
        return table;
    }
```

`CreateTable()` requires three arguments:

- `data`—The data it needs to put into a table.
- `variablesInTable`—What variables it needs to show.
- `Title`—The title of the table. It's always nice to know what you're looking at.

`CreateTable()` uses a predefined variable, `tableTemplate`, that contains our overall table layout. `CreateTable()` can then add rows of data to this template.

Now that you have your utilities, let's get to the `main` function of the application, as shown in the following listing.

Listing 9.3 JavaScript `main` function

```
main = function(inputdata){

    var medicineData = inputdata ;

    var dateFormat = d3.time.format("%d/%m/%Y");
    medicineData.forEach(function (d) {
        d.Day = dateFormat.parse(d.Date);
    })
    var variablesInTable =
['MedName','StockIn','StockOut','Stock','Date','LightSen']
      var sample = medicineData.slice(0,5);
      var inputTable = $("#inputtable");
```

Our data: normally this is fetched from a server but in this case we read it from a local .csv file

Put the variables we'll show in the table in an array so we can loop through them when creating table code

Convert date to correct format so Crossfilter will recognize date variable

Only show a sample of data

Create table

```
        inputTable
                .empty()
                .append(CreateTable(sample,variablesInTable,"The input table"));
}
```

You start off by showing your data on the screen, but preferably not all of it; only the first five entries will do, as shown in figure 9.4. You have a date variable in your data and you want to make sure Crossfilter will recognize it as such later on, so you first parse it and create a new variable called `Day`. You show the original, `Date`, to appear in the table for now, but later on you'll use `Day` for all your calculations.

The input table

MedName	StockIn	StockOut	Stock	Date	LightSen
Acupan 30 mg	150	-7	143	1/01/2015	No
Acupan 30 mg	5	-6	142	2/01/2015	No
Acupan 30 mg	15	-9	148	3/01/2015	No
Acupan 30 mg	0	-11	137	4/01/2015	No
Acupan 30 mg	10	-8	139	5/01/2015	No

Figure 9.4 Input medicine table shown in browser: first five lines

This is what you end up with: the same thing you saw in Excel before. Now that you know the basics are working, you'll introduce Crossfilter into the equation.

9.2.2 Unleashing Crossfilter to filter the medicine data set

Now let's go into Crossfilter to use filtering and MapReduce. Henceforth you can put all the upcoming code after the code of section 9.2.1 within the `main()` function. The first thing you'll need to do is declare a Crossfilter instance and initiate it with your data.

```
CrossfilterInstance = crossfilter(medicineData);
```

From here you can get to work. On this instance you can register dimensions, which are the columns of your table. Currently Crossfilter is limited to 32 dimensions. If you're handling data wider than 32 dimensions, you should consider narrowing it down before sending it to the browser. Let's create our first dimension, the medicine name dimension:

```
var medNameDim = CrossfilterInstance.dimension(function(d) {return
   d.MedName;});
```

Your first dimension is the name of the medicines, and you can already use this to filter your data set and show the filtered data using our `CreateTable()` function.

```
    var dataFiltered= medNameDim.filter('Grazax 75 000 SQ-T')
    var filteredTable = $('#filteredtable');
    filteredTable
  .empty().append(CreateTable(dataFiltered.top(5),variablesInTable,'Our
First Filtered Table'));
```

You show only the top five observations (figure 9.5); you have 365 because you have the results from a single medicine for an entire year.

Our First Filtered Table

MedName	StockIn	StockOut	Stock	Date	LightSen
Grazax 75 000 SQ-T	15	0	205	31/08/2015	Yes
Grazax 75 000 SQ-T	0	-4	62	30/12/2015	Yes
Grazax 75 000 SQ-T	10	-15	66	29/12/2015	Yes
Grazax 75 000 SQ-T	15	0	71	28/12/2015	Yes
Grazax 75 000 SQ-T	10	-4	56	27/12/2015	Yes

Figure 9.5 Data filtered on medicine name Grazax 75 000 SQ-T

This table doesn't look sorted but it is. The `top()` function sorted it on medicine name. Because you only have a single medicine selected it doesn't matter. Sorting on date is easy enough using your new `Day` variable. Let's register another dimension, the date dimension:

```
    var DateDim = CrossfilterInstance.dimension(
function(d) {return d.Day;});
```

Now we can sort on date instead of medicine name:

```
    filteredTable
            .empty()
            .append(CreateTable(DateDim.bottom(5),variablesInTable,'Our
First Filtered Table'));
```

The result is a bit more appealing, as shown in figure 9.6.

This table gives you a window view of your data but it doesn't summarize it for you yet. This is where the Crossfilter MapReduce capabilities come in. Let's say you would like to know how many observations you have per medicine. Logic dictates that you

Our First Filtered Table

MedName	StockIn	StockOut	Stock	Date	LightSen
Grazax 75 000 SQ-T	65	-12	53	1/01/2015	Yes
Grazax 75 000 SQ-T	15	-11	57	2/01/2015	Yes
Grazax 75 000 SQ-T	5	-9	53	3/01/2015	Yes
Grazax 75 000 SQ-T	5	-4	54	4/01/2015	Yes
Grazax 75 000 SQ-T	0	-14	40	5/01/2015	Yes

Figure 9.6 Data filtered on medicine name Grazax 75 000 SQ-T and sorted by day

should end up with the same number for every medicine: 365, or 1 observation per day in 2015.

```
        var countPerMed = medNameDim.group().reduceCount();
        variablesInTable = ["key","value"]
        filteredTable
                .empty()
    .append(CreateTable(countPerMed.top(Infinity),
variablesInTable,'Reduced Table'));
```

Crossfilter comes with two MapReduce functions: `reduceCount()` and `reduceSum()`. If you want to do anything apart from counting and summing, you need to write reduce functions for it. The `countPerMed` variable now contains the data grouped by the medicine dimension and a line count for each medicine in the form of a key and a value. To create the table you need to address the variable `key` instead of `medName` and `value` for the count (figure 9.7).

Reduced Table

key	value
Adoport 1 mg	365
Atenolol EG 100 mg	365
Ceftriaxone Actavis 1 g	365
Cefuroxim Mylan 500 mg	365
Certican 0.25 mg	365

Figure 9.7 MapReduced table with the medicine as the group and a count of data lines as the value

By specifying `.top(Infinity)` you ask to show all 29 medicines onscreen, but for the sake of saving paper figure 9.7 shows only the first five results. Okay, you can rest easy; the data contains 365 lines per medicine. Notice how Crossfilter ignored the filter on "Grazax". If a dimension is used for grouping, the filter doesn't apply to it. Only filters on other dimensions can narrow down the results.

What about more interesting calculations that don't come bundled with Crossfilter, such as an average, for instance? You can still do that but you'd need to write three functions and feed them to a `.reduce()` method. Let's say you want to know the average stock per medicine. As previously mentioned, almost all of the MapReduce logic needs to be written by you. An average is nothing more than the division of sum by count, so you will require both; how do you go about this? Apart from the `reduceCount()` and `reduceSum()` functions, Crossfilter has the more general `reduce()` function. This function takes three arguments:

- *The reduceAdd() function*—A function that describes what happens when an extra observation is added.
- *The reduceRemove() function*—A function that describes what needs to happen when an observation disappears (for instance, because a filter is applied).
- *The reduceInit() function*—This one sets the initial values for everything that's calculated. For a sum and count the most logical starting point is 0.

Let's look at the individual reduce functions you'll require before trying to call the Crossfilter `.reduce()` method, which takes these three components as arguments. A custom reduce function requires three components: an initiation, an add function, and a remove function. The initial reduce function will set starting values of the `p` object:

```
var reduceInitAvg = function(p,v){
    return {count: 0, stockSum : 0, stockAvg:0};
}
```

As you can see, the reduce functions themselves take two arguments. These are automatically fed to them by the Crossfilter `.reduce()` method:

- `p` is an object that contains the combination situation so far; it persists over all observations. This variable keeps track of the sum and count for you and thus represents your goal, your end result.
- `v` represents a record of the input data and has all its variables available to you. Contrary to `p`, it doesn't persist but is replaced by a new line of data every time the function is called. The `reduceInit()` is called only once, but `reduceAdd()` is called every time a record is added and `reduceRemove()` every time a line of data is removed.
- The `reduceInit()` function, here called `reduceInitAvg()` because you're going to calculate an average, basically initializes the `p` object by defining its components (`count`, `sum`, and `average`) and setting their initial values. Let's look at `reduceAddAvg()`:

```
var reduceAddAvg = function(p,v){
    p.count += 1;
    p.stockSum  = p.stockSum  + Number(v.Stock);
    p.stockAvg = Math.round(p.stockSum  / p.count);
    return p;
}
```

`reduceAddAvg()` takes the same `p` and `v` arguments but now you actually use `v`; you don't need your data to set the initial values of `p` in this case, although you can if you want to. Your `Stock` is summed up for every record you add, and then the average is calculated based on the accumulated sum and record count:

```
var reduceRemoveAvg = function(p,v){
    p.count -= 1;
    p.stockSum  = p.stockSum  -  Number(v.Stock);
    p.stockAvg = Math.round(p.stockSum  / p.count);
    return p;
}
```

The `reduceRemoveAvg()` function looks similar but does the opposite: when a record is removed, the count and sum are lowered. The average always calculates the same way, so there's no need to change that formula.

The moment of truth: you apply this homebrewed MapReduce function to the data set:

```
        dataFiltered = medNameDim.group().reduce(reduceAddAvg,
reduceRemoveAvg,reduceInitAvg)

        variablesInTable = ["key","value.stockAvg"]
        filteredTable
                .empty()
    .append(CreateTable(dataFiltered.top(Infinity),
variablesInTable,'Reduced Table'));
```

Business as usual: draw result table.

reduce() takes the 3 functions (reduceInitAvg(), reduceAddAvg(), and reduceRemoveAvg()) as input arguments.

Notice how the name of your output variable has changed from `value` to `value.stockAvg`. Because you defined the `reduce` functions yourself, you can output many variables if you want to. Therefore, `value` has changed into an object containing all the variables you calculated; `stockSum` and `count` are also in there.

The results speak for themselves, as shown in figure 9.8. It seems we've borrowed Cimalgex from other hospitals, going into an average negative stock.

This is all the Crossfilter you need to know to work with dc.js, so let's move on and bring out those interactive graphs.

Reduced Table

key	value.stockAvg
Adoport 1 mg	36
Atenolol EG 100 mg	49
Ceftriaxone Actavis 1 g	207
Cefuroxim Mylan 500 mg	118
Certican 0.25 mg	158
Cimalgex 8 mg	-24

Figure 9.8 MapReduced table with average stock per medicine

9.3 Creating an interactive dashboard with dc.js

Now that you know the basics of Crossfilter, it's time to take the final step: building the dashboard. Let's kick off by making a spot for your graphs in the `index.html` page. The new body looks like the following listing. You'll notice it looks similar to our initial setup apart from the added graph placeholder `<div>` tags and the reset button `<button>` tag.

Listing 9.4 A revised index.html with space for graphs generated by dc.js

```
Layout:
        Title
        | input table                          | (row 1)
        | filtered table                       | (row 2)
        [ reset button ]
        | stock-over-time chart | stock-per-medicine chart | (row 3)
        | light-sensitive chart |                          | (row 4)
            (column 1)          (column 2)

<body>
    <main class='container'>

        <h1>Chapter 10:  Data Science Application</h1>
        <div class="row">
            <div class='col-lg-12'>
                <div id="inputtable" class="well well-sm">
</div>                                          <-- This is a placeholder <div> for input data table inserted later.

            </div>
        </div>
        <div class="row">
            <div class='col-lg-12'>
                <div id="filteredtable" class="well well-sm">
</div>                                          <-- This is a placeholder <div> for filtered table inserted later.

            </div>
        </div>

        <button class="btn btn-success">Reset Filters</button>      <-- This is new: reset button.
        <div class="row">
            <div class="col-lg-6">
                <div id="StockOverTime" class="well well-sm"></div>     <-- This is new: time chart placeholder.

                <div id="LightSensitiveStock" class="well well-sm"></div>   <-- This is new: light sensitivity pie-chart placeholder.

            </div>
            <div class="col-lg-6">
                <div id="StockPerMedicine" class="well well-sm"></div>      <-- This is new: stock per medicine bar-chart placeholder.
            </div>
        </div>
    </main>
```

```
    <script src="https://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap
/3.3.0/js/bootstrap.min.js"></script>

    <script src="crossfilter.min.js"></script>
    <script src="d3.v3.min.js"></script>
    <script src="dc.min.js"></script>

    <script src="application.js"></script>
</body>
```

Crossfilter, d3, and dc libraries can be downloaded from their respective websites.

Our own application JavaScript code.

Standard practice. JS libraries are last to speed page load.
jQuery: vital HTML-JavaScript interaction
Bootstrap: simplified CSS and layout from folks at Twitter
Crossfilter: our JavaScript MapReduce library of choice
d3: the d3 script, necessary to run dc.js
DC: our visualization library
application: our data science application; here we store all the logic
***.min.js denotes minified JavaScript for our 3rd party libraries**

We've got Bootstrap formatting going on, but the most important elements are the three `<div>` tags with IDs and the button. What you want to build is a representation of the total stock over time, `<div id="StockOverTime"></div>`, with the possibility of filtering on medicines, `<div id="StockPerMedicine"></div>`, and whether they're light-sensitive or not, `<div id="LightSensitiveStock"></div>`. You also want a button to reset all the filters, `<button class="btn btn-success">Reset Filters</button>`. This reset button element isn't required, but is useful.

Now turn your attention back to application.js. In here you can add all the upcoming code in your `main()` function as before. There is, however, one exception to the rule: `dc.renderAll();` is dc's command to draw the graphs. You need to place this render command only once, at the bottom of your `main()` function. The first graph you need is the "total stock over time," as shown in the following listing. You already have the time dimension declared, so all you need is to sum your stock by the time dimension.

Listing 9.5 Code to generate "total stock over time" graph

```
var SummatedStockPerDay =
DateDim.group().reduceSum(function(d){return d.Stock;})

var minDate = DateDim.bottom(1)[0].Day;
var maxDate = DateDim.top(1)[0].Day;
var StockOverTimeLineChart = dc.lineChart("#StockOverTime");

StockOverTimeLineChart
        .width(null) // null means size to fit container
        .height(400)
        .dimension(DateDim)
        .group(SummatedStockPerDay)
```

Stock over time data

Line chart

Deliveries per day graph

```
                .x(d3.time.scale().domain([minDate,maxDate]))
                .xAxisLabel("Year 2015")
                .yAxisLabel("Stock")
                .margins({left: 60, right: 50, top: 50, bottom: 50})

dc.renderAll();
```

Deliveries per day graph

Render all graphs

Look at all that's happening here. First you need to calculate the range of your x-axis so dc.js will know where to start and end the line chart. Then the line chart is initialized and configured. The least self-explanatory methods here are `.group()` and `.dimension()`. `.group()` takes the time dimension and represents the x-axis. `.dimension()` is its counterpart, representing the y-axis and taking your summated data as input. Figure 9.9 looks like a boring line chart, but looks can be deceiving.

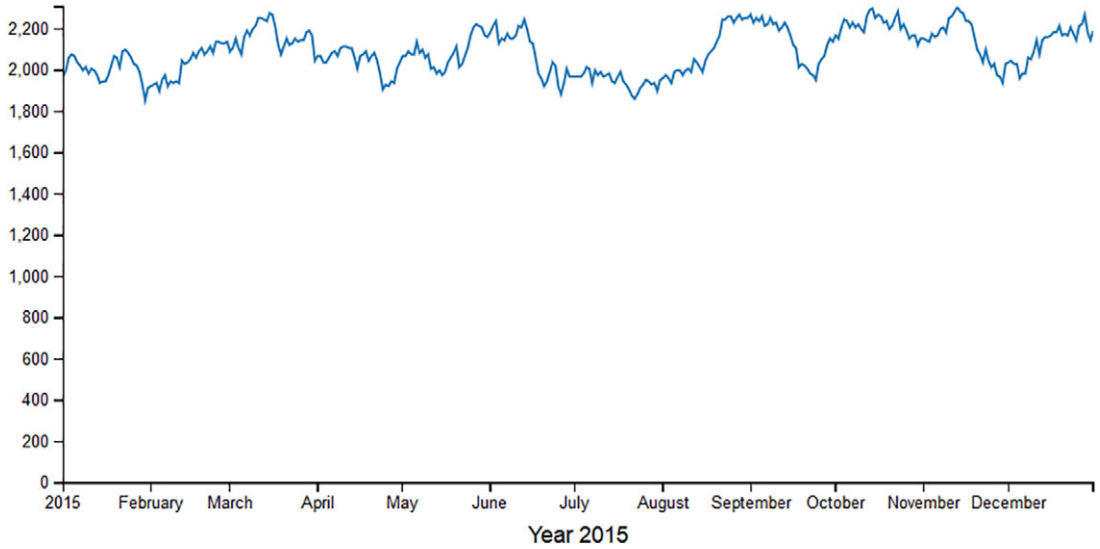

Figure 9.9 dc.js graph: sum of medicine stock over the year 2015

Things change drastically once you introduce a second element, so let's create a row chart that represents the average stock per medicine, as shown in the next listing.

Listing 9.6 Code to generate "average stock per medicine" graph

```
var AverageStockPerMedicineRowChart = dc.rowChart("#StockPerMedicine");
var AvgStockMedicine =  medNameDim.group().reduce(reduceAddAvg,
reduceRemoveAvg,reduceInitAvg);

AverageStockPerMedicineRowChart
                .width(null)
                .height(1200)
```

Null means "size to fit container"

Average stock per medicine row chart

```
        .dimension(medNameDim)
        .group(AvgStockMedicine)
        .margins({top: 20, left: 10, right: 10, bottom: 20})
        .valueAccessor(function (p) {return p.value.stockAvg;});
```

This should be familiar because it's a graph representation of the table you created earlier. One big point of interest: because you used a custom-defined `reduce()` function this time, dc.js doesn't know what data to represent. With the `.valueAccessor()` method you can specify `p.value.stockAvg` as the value of your choice. The dc.js row chart's label's font color is gray; this makes your row chart somewhat hard to read. You can remedy this by overwriting its CSS in your application.css file:

```
.dc-chart g.row text {fill: black;}
```

One simple line can make the difference between a clear and an obscure graph (figure 9.10).

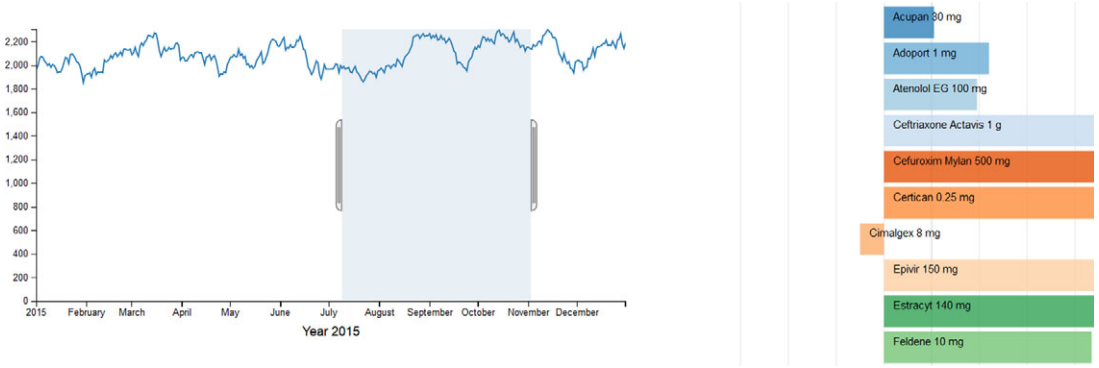

Figure 9.10 dc.js line chart and row chart interaction

Now when you select an area on the line chart, the row chart is automatically adapted to represent the data for the correct time period. Inversely, you can select one or multiple medicines on the row chart, causing the line chart to adjust accordingly. Finally, let's add the light-sensitivity dimension so the pharmacist can distinguish between stock for light-sensitive medicines and non-light-sensitive ones, as shown in the following listing.

Listing 9.7 Adding the light-sensitivity dimension

```
        var lightSenDim = CrossfilterInstance.dimension(
function(d){return d.LightSen;});
        var SummatedStockLight =  lightSenDim.group().reduceSum(
function(d) {return d.Stock;});

        var LightSensitiveStockPieChart = dc.pieChart("#LightSensitiveStock");
```

```
LightSensitiveStockPieChart
        .width(null) // null means size to fit container
        .height(300)
        .dimension(lightSenDim)
        .radius(90)
        .group(SummatedStockLight)
```

We hadn't introduced the light dimension yet, so you need to register it onto your Crossfilter instance first. You can also add a reset button, which causes all filters to reset, as shown in the following listing.

Listing 9.8 The dashboard reset filters button

```
resetFilters = function(){
    StockOverTimeLineChart.filterAll();
    LightSensitiveStockPieChart.filterAll();
    AverageStockPerMedicineRowChart.filterAll();
    dc.redrawAll();
}
$('.btn-success').click(resetFilters);
```

When an element with class btn-success is clicked (our reset button), resetFilters() is called.

resetFilters() function will reset our dc.js data and redraw graphs.

The `.filterAll()` method removes all filters on a specific dimension; `dc.redrawAll()` then manually triggers all dc charts to redraw.

The final result is an interactive dashboard (figure 9.11), ready to be used by our pharmacist to gain insight into her stock's behavior.

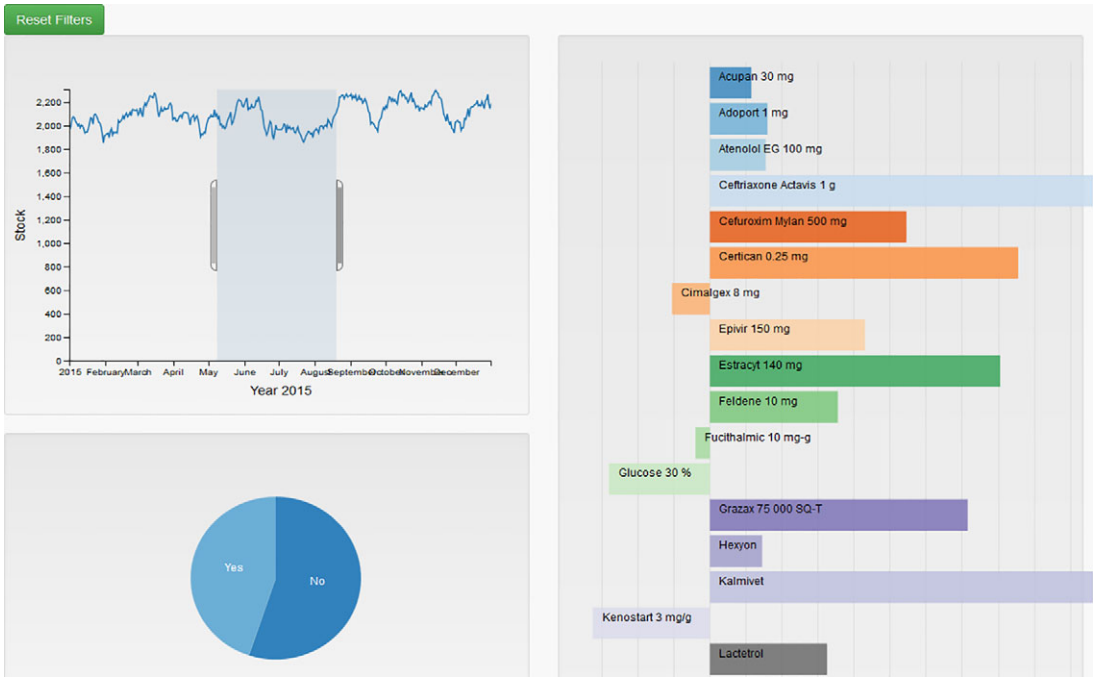

Figure 9.11 dc.js fully interactive dashboard on medicines and their stock within the hospital pharmacy

9.4 Dashboard development tools

We already have our glorious dashboard, but we want to end this chapter with a short (and far from exhaustive) overview of the alternative software choices when it comes to presenting your numbers in an appealing way.

You can go with proven and true software packages of renowned developers such as Tableau, MicroStrategy, Qlik, SAP, IBM, SAS, Microsoft, Spotfire, and so on. These companies all offer dashboard tools worth investigating. If you're working in a big company, chances are good you have at least one of those paid tools at your disposal. Developers can also offer free public versions with limited functionality. Definitely check out Tableau if you haven't already at http://www.tableausoftware.com/public/download.

Other companies will at least give you a trial version. In the end you have to pay for the full version of any of these packages, and it might be worth it, especially for a bigger company that can afford it.

This book's main focus is on free tools, however. When looking at free data visualization tools, you quickly end up in the HTML world, which proliferates with free JavaScript libraries to plot any data you want. The landscape is enormous:

- *HighCharts*—One of the most mature browser-based graphing libraries. The free license applies only to noncommercial pursuits. If you want to use it in a commercial context, prices range anywhere from $90 to $4000. See http://shop.highsoft.com/highcharts.html.
- *Chartkick*—A JavaScript charting library for Ruby on Rails fans. See http://ankane.github.io/chartkick/.
- *Google Charts*—The free charting library of Google. As with many Google products, it is free to use, even commercially, and offers a wide range of graphs. See https://developers.google.com/chart/.
- *d3.js*—This is an odd one out because it isn't a graphing library but a data visualization library. The difference might sound subtle but the implications are not. Whereas libraries such as HighCharts and Google Charts are meant to draw certain predefined charts, d3.js doesn't lay down such restrictions. d3.js is currently the most versatile JavaScript data visualization library available. You need only a quick peek at the interactive examples on the official website to understand the difference from a regular graph-building library. See http://d3js.org/.

Of course, others are available that we haven't mentioned.

You can also get visualization libraries that only come with a trial period and no free community edition, such as Wijmo, Kendo, and FusionCharts. They are worth looking into because they also provide support and guarantee regular updates.

You have options. But why or when would you even consider building your own interface with HTML5 instead of using alternatives such as SAP's BusinessObjects, SAS JMP, Tableau, Clickview, or one of the many others? Here are a few reasons:

- *No budget*—When you work in a startup or other small company, the licensing costs accompanying this kind of software can be high.

- *High accessibility*—The data science application is meant to release results to any kind of user, especially people who might only have a browser at their disposal—your own customers, for instance. Data visualization in HTML5 runs fluently on mobile.
- *Big pools of talent out there*—Although there aren't that many Tableau developers, scads of people have web-development skills. When planning a project, it's important to take into account whether you can staff it.
- *Quick release*—Going through the entire IT cycle might take too long at your company, and you want people to enjoy your analysis quickly. Once your interface is available and being used, IT can take all the time they want to industrialize the product.
- *Prototyping*—The better you can show IT its purpose and what it should be capable of, the easier it is for them to build or buy a sustainable application that does what you want it to do.
- *Customizability*—Although the established software packages are great at what they do, an application can never be as customized as when you create it yourself.

And why wouldn't you do this?

- *Company policy*—This is the biggest one: it's not allowed. Large companies have IT backup teams that allow only a certain number of tools to be used so they can keep their supporting role under control.
- *You have an experienced team of reporters at your disposal*—You'd be doing their job, and they might come after you with pitchforks.
- *Your tool does allow enough customization to suit your taste*—Several of the bigger platforms are browser interfaces with JavaScript running under the hood. Tableau, BusinessObjects Webi, SAS Visual Analytics, and so on all have HTML interfaces; their tolerance to customization might grow over time.

The front end of any application can win the hearts of the crowd. All the hard work you put into data preparation and the fancy analytics you applied is only worth as much as you can convey to those who use it. Now you're on the right track to achieve this. On this positive note we'll conclude this chapter.

9.5 Summary

- This chapter focused on the last part of the data science process, and our goal was to build a data science application where the end user is provided with an interactive dashboard. After going through all the steps of the data science process, we're presented with clean, often compacted or information dense, data. This way we can query less data and get the insights we want.
- In our example, the pharmacy stock data is considered thoroughly cleaned and prepared and this should always be the case by the time the information reaches the end user.

- JavaScript-based dashboards are perfect for quickly granting access to your data science results because they only require the user to have a web browser. Alternatives exist, such as Qlik (chapter 5).
- Crossfilter is a MapReduce library, one of many JavaScript MapReduce libraries, but it has proven its stability and is being developed and used by Square, a company that does monetary transactions. Applying MapReduce is effective, even on a single node and in a browser; it increases the calculation speed.
- dc.js is a chart library build on top of d3.js and Crossfilter that allows for quick browser dashboard building.
- We explored the data set of a hospital pharmacy and built an interactive dashboard for pharmacists. The strength of a dashboard is its *self-service* nature: they don't always need a reporter or data scientist to bring them the insights they crave.
- Data visualization alternatives are available, and it's worth taking the time to find the one that suits your needs best.
- There are multiple reasons why you'd create your own custom reports instead of opting for the (often more expensive) company tools out there:
  - *No budget*—Startups can't always afford every tool
  - *High accessibility*—Everyone has a browser
  - *Available talent*—(Comparatively) easy access to JavaScript developers
  - *Quick release*—IT cycles can take a while
  - *Prototyping*—A prototype application can provide and leave time for IT to build the production version
  - *Customizability*—Sometimes you just want it exactly as your dreams picture it.
- Of course there are reasons against developing your own application:
  - *Company policy*—Application proliferation isn't a good thing and the company might want to prevent this by restricting local development.
  - *Mature reporting team*—If you have a good reporting department, why would you still bother?
  - *Customization is satisfactory*—Not everyone wants the shiny stuff; basic can be enough.

Congratulations! You've made it to the end of this book and the true beginning of your career as a data scientist. We hope you had ample fun reading and working your way through the examples and case studies. Now that you have basic insight into the world of data science, it's up to you to choose a path. The story continues, and we all wish you great success in your quest of becoming the greatest data scientist who has ever lived! May we meet again someday. ;)