brief contents

PART 1 BACKGROUND AND FUNDAMENTALS .........................................................1

1 ■ Hadoop in a heartbeat 3
2 ■ Introduction to YARN 22

PART 2 DATA LOGISTICS ..................................................................................59

3 ■ Data serialization—working with text and beyond 61
4 ■ Organizing and optimizing data in HDFS 139
5 ■ Moving data into and out of Hadoop 174

PART 3 BIG DATA PATTERNS ............................................................................253

6 ■ Applying MapReduce patterns to big data 255
7 ■ Utilizing data structures and algorithms at scale 302
8 ■ Tuning, debugging, and testing 337

PART 4 BEYOND MAPREDUCE ..........................................................................385

9 ■ SQL on Hadoop 387
10 ■ Writing a YARN application 425
This chapter covers
- Integrating your R scripts with MapReduce and streaming
- Understanding RHadoop, and R and streaming

R is a statistical programming language for performing data analysis and graphing the results. R lets you perform statistical and predictive analytics, data mining, and visualization functions on your data. It contains built-in as well as user-created packages that can be accessed via CRAN, its package-distribution system; see the list of CRAN contributed packages at http://cran.r-project.org/web/packages/.

1 R contains built-in as well as user-created packages that can be accessed via CRAN, its package-distribution system; see the list of CRAN contributed packages at http://cran.r-project.org/web/packages/.
A data scientist who’s working with R usually has an existing arsenal of R packages that they use for their work. Rewriting these packages in Java (or any other high-level MapReduce language) would be onerous and would be the antithesis of rapid development (especially in Java). What you need is a way to use R in conjunction with Hadoop and bridge the gap between Hadoop and the huge database of information that exists in R.

Much of the data you work with exists in text form, such as tweets from Twitter, logs, and stock records. In this chapter we’ll look at how you can use R to calculate simple average-based calculations on text-based stock data. In doing so, you’ll learn how you can use R with two different integration approaches: R with streaming and RHadoop. By the end of the chapter you’ll understand the various ways that R can be integrated with Hadoop and how to pick the best approach for your application.

### 11.1 Comparing R and MapReduce integrations

In this section we’ll evaluate the two different methods you’ll use in this chapter to integrate R with MapReduce. I picked these two approaches due to their popularity and differing approaches to solving the same problem—that of combining R and Hadoop together:

- **R and streaming**—With this approach, you use MapReduce to execute R scripts in the map and reduce phases.
- **RHadoop**—RHadoop provides an R wrapper around MapReduce so that they can be seamlessly integrated on the client side.

Table 11.1 compares these two options.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>R and streaming</th>
<th>RHadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>License</td>
<td>R is a combination of GPL-2 and GPL-3. Streaming is integrated into Hadoop, which is Apache 2.0.</td>
<td>Apache 2.0.</td>
</tr>
<tr>
<td>Installation complexity</td>
<td>Easy. The R package needs to be installed on each DataNode, but packages are available on publicly available Yum repositories for easy installation.</td>
<td>Moderate. R must be installed on each DataNode, and RHadoop has dependencies on other R packages. But these packages can be installed with CRAN, and the RHadoop installation, while not via CRAN, is straightforward.</td>
</tr>
</tbody>
</table>
Table 11.1 Comparing R and MapReduce integration options (continued)

<table>
<thead>
<tr>
<th>Criteria</th>
<th>R and streaming</th>
<th>RHadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-side integration with R</td>
<td>None. You need to use the Hadoop command line to launch a streaming job, and specify as arguments the map-side and reduce-side R scripts.</td>
<td>High. RHadoop is also an R library, where users define their map and reduce functions in R.</td>
</tr>
<tr>
<td>Underlying technologies used</td>
<td>Streaming.</td>
<td>RHadoop is a simple, thin wrapper on top of Hadoop and streaming. Therefore, it has no proprietary MapReduce code, and has a simple wrapper R script that’s called from streaming and in turn calls the user’s map and reduce R functions.</td>
</tr>
</tbody>
</table>

Which tool should you pick? As we go through the techniques, you may find that one approach lends itself more to your situation than others. Table 11.2 presents my opinions on which tools perform best in different situations.

Table 11.2 Benefits of R and MapReduce approaches in different situations

<table>
<thead>
<tr>
<th>Approach</th>
<th>Works well in these situations</th>
<th>Things to bear in mind</th>
</tr>
</thead>
<tbody>
<tr>
<td>R and streaming</td>
<td>You want advanced control over your MapReduce functions, such as partitioning and sorting.</td>
<td>Hard to invoke directly from existing R scripts, as compared to the other approach.</td>
</tr>
<tr>
<td>RHadoop</td>
<td>You want access to R and MapReduce without leaving R. You can also work with existing MapReduce input and output format classes.</td>
<td>There needs to be sufficient memory to store all the reducer values for a unique key in memory; values aren’t streamed to the reducer function.</td>
</tr>
</tbody>
</table>

Before we dive into these technologies, let’s take a few minutes to look at some R basics so that your techniques don’t look too alien.

11.2 R fundamentals

In this section we’ll quickly look at some R basics so you understand R language constructs and types.

**Installation**  Follow the instructions in the appendix to install R. Care should be taken to install R in the same directory on all the nodes. Also make sure that all nodes are running the same version of R.

**Starting R and running your first command**  Starting R and running a command couldn’t be simpler:

\[
\text{Calculate the minimum number in a sequence between 1 and 5.}\quad \text{min(1:5)} \quad \text{[1] 1}
\]
Variables

```r
> x <- 2
> print(x)
[1] 2
> y = 5
> y <- toString(mean(c(1:5)))
> z <- c(3:5)
```

```r
> ls()
[1] "x" "y" "z"
> ls.str()
  x : num 2
  y : chr "3"
  z : int [1:3] 3 4 5
> rm(x)
> ls()
[1] "y" "z"
```

**Figure 11.1** Examples of R variables in use

Quick examples

Figure 11.1 illustrates a handful of R basics to help you understand the techniques in this chapter.

Figure 11.2 presents some R vector basics. R supports other data structures such as matrices, arrays, data frames, and factors. This chapter focuses on using vectors (because the examples in this chapter only use vectors), so we won’t get into the details of these other data structures. To sink your teeth into R, take a look at *R in Action, Second Edition* from Robert Kabacoff (Manning, 2013, www.manning.com/kabacoff2/).

You’ve seen some simple examples of how to get up and running with R. It’s time to introduce Hadoop into the equation and look at how you can use R in combination with Hadoop streaming.

### 11.3 R and streaming

With Hadoop streaming, you can write map and reduce functions in any programming or scripting language that supports reading data from standard input and writing to standard output. In this section we’ll look at how you can get streaming to work directly with R in two steps, first in a map-only job, and then in a full MapReduce job. You’ll work with stock data and perform simple calculations. The goal is to show how the integration of R and Hadoop can be made with streaming.

#### 11.3.1 Streaming and map-only R

Just like with regular MapReduce, you can have a map-only job in streaming and R. Map-only jobs make sense in situations where you don’t want to join or group your data together in the reducer.
Vectors

Create a vector of strings.

```r
> x <- c(1, 2*4, "5", TRUE)
>
Because we're mixing types, R coerces all the data into strings.

> x <- c(x, 10, 4:6)
Add elements to the end of the vector.

> x[2:4]
[1] "8" "5" "TRUE"
Extract a range of elements from index 2 to 4.

> x[-(2:4)]
[1] "1" "10" "4" "5" "6"
Extract all elements from the vector excluding elements 2 to 4.

> x <- as.numeric(x)
Convert the string vector into a numerical vector and remove non-numerics.

> print(x <- x[!is.na(x)])
[1] 1 8 5 10 4 5 6
The is.na returns a vector of TRUE or FALSE elements that you use to remove NAs from x.

> summary(x)
Calculate summary statistics on vector x.

   Min. 1st Qu.  Median    Mean 3rd Qu.   Max.  
  1.000  4.500  5.000    5.571  7.000  10.000

> unlist(lapply(c(1:3), function(y) y+1))
Generate a new vector with all the values incremented by one.

The same as lapply, but conveniently returns a vector rather than a list.

> sapply(c(1:3), function(y) y+1)
[1] 2 3 4

Figure 11.2 Examples of R vectors and functions

In this technique you'll look at how Hadoop streaming and R can be used on your stock data to calculate the daily means for each stock symbol.
CHAPTER 11  Integrating R and Hadoop for statistics and more

Problem
You want to integrate R and MapReduce, and you don’t need to join your data or sort your outputs.

Solution
Use R and Hadoop streaming to process data in a map-only job.

Discussion
In this technique you’ll work on the stock CSV file, which contains the following elements for each stock:

Symbol,Date,Open,High,Low,Close,Volume,Adj Close

A subset of the contents of the file can be viewed here:

```
$ head -6 test-data/stocks.txt
AAPL,2009-01-02,85.88,91.04,85.16,90.75,26643400,90.75
AAPL,2008-01-02,199.27,200.26,192.55,194.84,38542100,194.84
AAPL,2007-01-03,86.29,86.58,81.90,83.80,44225700,83.80
AAPL,2006-01-03,72.38,74.75,72.25,74.75,28829800,74.75
AAPL,2005-01-03,64.78,65.11,62.60,63.29,24714000,31.65
AAPL,2004-01-02,21.55,21.75,21.18,21.28,5165800,10.64
```

In your job, you’ll calculate the daily mean for each line using the open and close prices. The R script to perform that task is shown here: 2

```r
#! /usr/bin/env Rscript
options(warn=-1)
sink("/dev/null")
input <- file("stdin", "r")

while(length(currentLine <- readLines(input, n=1, warn=FALSE)) > 0) {

  fields <- unlist(strsplit(currentLine, ","))
  lowHigh <- c(as.double(fields[3]), as.double(fields[6]))
  mean <- mean(lowHigh)
  sink()
  cat(fields[1], fields[2], mean, "\n", sep="\t")
  sink("/dev/null")
}
close(input)
```


---

TECHNIQUE 105  R and streaming

Figure 11.3 shows how streaming and R work together in a map-only job.

Any MapReduce code can be challenging to test, but the great thing about using Hadoop streaming code is that it’s very easy to test on the command line without having to involve MapReduce at all. The following command shows how the Linux `cat` utility (a simple utility for writing the contents of a file to standard output) can be used to quickly test your R script to make sure the output is what you expect:

```
$ cat test-data/stocks.txt | R/stock_day_avg.R
AAPL 2009-01-02 88.315
AAPL 2008-01-02 197.055
AAPL 2007-01-03 85.045
AAPL 2006-01-03 73.565
...
```

That output looks good, so you’re ready to run this in a Hadoop job:

```
$ export HADOOP_HOME=/usr/local/hadoop
$ export HADOOP_HOME=/usr/local/hadoop
$ ${HADOOP_HOME}/bin/hadoop fs -rmr output
$ ${HADOOP_HOME}/bin/hadoop fs -put test-data/stocks.txt

$ hadoop \n  jar ${HADOOP_HOME}/share/hadoop/tools/lib/hadoop-streaming*.jar \n```
You can perform a simple cat that shows you that the output is identical to what you produced when calling the R script directly:

```
$ hadoop fs -cat output/part*  
AAPL 2009-01-02   88.315  
AAPL 2008-01-02   197.055  
AAPL 2007-01-03   85.045  
AAPL 2006-01-03   73.565  
...
```

You may have noticed that you used TextInputFormat, which emits a key/value tuple where the key is the byte offset in the file, and the value contains the contents of a line. But in your R script, you were only supplied the value part of the tuple. This is an optimization in Hadoop streaming, where if it detects you’re using TextInputFormat, it ignores the key from the TextInputFormat. If you want the key supplied to your script, you can set the stream.map.input.ignoreKey Hadoop configuration to true.

Figure 11.4 shows some streaming configuration settings that can be used to customize map inputs and outputs.

Now that we’ve covered how to use R and streaming for a map-only job, let’s see how to get R working with a full map and reduce job.

### 11.3.2 Streaming, R, and full MapReduce

We’ll now look at how you can integrate R with a full-blown MapReduce job. You’ll build upon what you’ve learned about using streaming and a map-side R function, and we’ll introduce a reduce-side function. In doing so, you’ll learn how Hadoop streaming supplies map output keys and the list of map output value tuples to the standard input of the R function, and how the R function outputs are collected.

#### TECHNIQUE 106 Calculate the cumulative moving average for stocks

The previous technique calculated the daily mean for each stock symbol. In this technique you’ll use the MapReduce framework to group together all of the daily means for each stock symbol across multiple days, and then calculate a cumulative moving average (CMA) over that data.
By default, the input keys and values are separated by the tab character. To override this value, use the following configuration key, which in this example tells streaming to use the comma as the separator string:

-D stream.map.input.field.separator=","

To extract the key/value pair from a line of output from a script, streaming will split using the tab character. This can be overridden with the following configuration key:

-D stream.map.output.field.separator=","

Streaming will split the output line based on the first occurrence of stream.map.output.field.separator to determine which part is the key and value. If instead you wanted to split on the third instance of the separator character, you'd specify the following setting in your job:

-D stream.num.map.output.key.fields=3

---

**Problem**

You want to integrate R and streaming in both the map and reduce sides.

**Solution**

Use R and Hadoop streaming to process data in mappers and reducers.

**Discussion**

Recall that in the map-side technique, the map R script emitted tab-separated output with the following fields:

Symbol  Date  Mean

MapReduce will sort and group together the output keys of your map script, which is the stock symbol. For each unique stock symbol, MapReduce will feed your reduce R script with all the map output values for that stock symbol. Your script will sum the means together and emit a single output containing the CMA, as shown in the following listing.³

---

**Listing 11.1 R script emits a single output**

```r
#!/usr/bin/env Rscript
options(warn=-1)
sink("/dev/null")
```

A simple R function that takes as input the stock symbol and a vector of means. It calculates the CMA and writes the symbol and CMA to standard output.

```r
outputMean <- function(stock, means) {
  stock_mean <- mean(means)
  sink()
  cat(stock, stock_mean, "\n", sep="\t")
  sink("/dev/null")
}

input <- file("stdin", "r")
prevKey <- "
means <- numeric(0)

while(length(currentLine <- readLines(input, n=1, warn=FALSE)) > 0) {
  fields <- unlist(strsplit(currentLine, "\t"))
  key <- fields[1]
  mean <- as.double(fields[3])
  if( identical(prevKey, ") || identical(prevKey, key)) {
    prevKey <- key
    means <- c(means, mean)
  } else {
    outputMean(prevKey, means)
    prevKey <- key
    means <- c(means, mean)
  }
}

if(!identical(prevKey, ")) {
  outputMean(prevKey, means)
}

close(input)
```

**Summary**

Figure 11.5 shows how streaming and your R script work together on the reduce side. Again, the beauty of streaming is that you can easily test it with streaming Linux commands:

```
$ cat test-data/stocks.txt | R/test_day_avg.R | \n  sort --key 1,1 | R/test_cma.R
AAPL 68.997
CSCO 49.94775
GOOG 123.9468
MSFT 101.297
YHOO 94.55789
```

That output looks good, so you’re ready to run this in a Hadoop job:

```
$ export HADOOP_HOME=/usr/lib/hadoop
$ ${HADOOP_HOME}/bin/hadoop fs -rmdir output
```
```bash
$ {HADOOP_HOME}/bin/hadoop fs -put test-data/stocks.txt stocks.txt

$ {HADOOP_HOME}/bin/hadoop \
  jar ${HADOOP_HOME}/share/hadoop/tools/lib/hadoop-streaming*.jar \
  -inputformat org.apache.hadoop.mapred.TextInputFormat \
  -input stocks.txt \
  -output output \
```

Figure 11.5 The R and streaming MapReduce data flow
You can perform a simple `cat` that shows you that the output is identical to what you produced when calling the R script directly:

```bash
$ hadoop fs -cat output/part*
AAPL  68.997
CSCO  49.94775
GOOG  123.9468
MSFT  101.297
YHOO  94.55789
```

Figure 11.6 shows streaming configuration settings that can be used to customize map inputs and outputs.

To set a custom input key/value separator string, use the following configuration key. The default is the tab character:

```
-D stream.reduce.input.field.separator=","
```

To set a custom output key/value separator string, use the following configuration key. The default is the tab character:

```
-D stream.reduce.output.field.separator=","
```

Set the number of stream.reduce.output.field.separator separators, which delimit the output key from the output value. The default is 1:

```
-D stream.num.reduce.output.field.separator=3
```

What if the map output values need to be supplied to the reducer in a specific order for each map output key (called a *secondary sort*)? You encountered secondary sorts in chapters 6 and 7 when performing joins and graph operations. Secondary sort in streaming can be achieved by using the `KeyFieldBasedPartitioner`, as shown here:

```bash
$ export HADOOP_HOME=/usr/lib/hadoop
$ ${HADOOP_HOME}/bin/hadoop fs -rmr output
$ ${HADOOP_HOME}/bin/hadoop fs -put test-data/stocks.txt stocks.txt
$ ${HADOOP_HOME}/bin/hadoop \
  jar ${HADOOP_HOME}/share/hadoop/tools/lib/hadoop-streaming*.jar \
  -D stream.num.map.output.key.fields=2 \ 
  -D mapred.text.key.partitioner.options=-k1,1 \ 
  -inputformat org.apache.hadoop.mapred.TextInputFormat \
  -outputformat org.apache.hadoop.mapred.TextOutputFormat \
```
TECHNIQUE 107  RHadoop—a simple integration of client-side R and Hadoop

For additional streaming features, such as greater control over sorts, please look at the Hadoop streaming documentation (http://hadoop.apache.org/docs/r0.18.3/streaming.html).

We’ve looked at how you can use R in combination with streaming to calculate the means over your stock data. One of the disadvantages of this approach is that this can’t be easily integrated into client-side R scripts. This is the problem that RHadoop solves.

11.4 RHadoop—a simple integration of client-side R and Hadoop

RHadoop is an open source project created by Revolution Analytics that provides another approach to integrating R and Hadoop. RHadoop allows MapReduce interactions directly from within your R code.

RHadoop contains a number of components, including these two:

- *rmr*—The integration of R and MapReduce
- *rdfs*—An R interface to HDFS

We’ll focus on using rmr in this chapter because we’re mostly interested in R and MapReduce integration, but rdfs is worth a look for a completely integrated R and Hadoop experience.

**RHadoop installation** To set up RHadoop and its dependencies, follow the instructions in the appendix.

TECHNIQUE 107  Calculating CMA with RHadoop

In this technique we’ll look at how you can use RHadoop to calculate the CMA for your stock data.

- **Problem**
  You want a simpler R and Hadoop client-side integrated solution.

- **Solution**
  This technique looks at how RHadoop can be used with R to launch a MapReduce job inside R, and it also looks at how RHadoop works with Hadoop streaming.

- **Discussion**
  In RHadoop you define your map and reduce operations, which RHadoop invokes as part of the MapReduce job.⁴

---

```r
#define a map function, which takes a key/value pair as input. The keyval function is called for each key/value output tuple that the map emits.

map <- function(k, v) {
  keyval(v[[1]], mean(as.double(c(v[[3]], v[[6]]))))
}

#reduce function, which is called once for each unique map key, where k is the key, and v is a list of values.
reduce <- function(k, vv) {
  keyval(k, mean(as.numeric(unlist(vv))))
}

# define your own reduce output key/value separator.
kvtextoutputformat = function(k, v) {
  paste(c(k, v, "\n"), collapse = "\t")
}

# run a MapReduce job.
mapreduce(
  input = "stocks.txt",
  input.format = make.input.format("csv", sep = ","),
  output = "output",
  output.format = "text",
  map = map,
  reduce = reduce)
```

**Summary**

To execute the code in this technique, you’d run the following commands:

```bash
$ HADOOP_HOME=<Hadoop installation directory>

$ $HADOOP_HOME/bin/hadoop fs -put test-data/stocks.txt stocks.txt

$ R/stock_cma_rmr.R

$ hadoop fs -cat output/part*
```

rmr uses Hadoop streaming. Figure 11.7 shows how your code correlates to the MapReduce job execution.

One of the interesting features of rmr is that it makes the R client-side environment available to the map and reduce R functions executed in MapReduce. This means that the map and reduce functions can reference variables outside of the scope of their respective functions, which is a huge boon for R developers.

rmr has another neat trick up its sleeve in that it can work seamlessly with MapReduce inputs and outputs. In this technique, the input to your job was already in HDFS, and you didn’t interact with the output of your job in R. But rmr has support for writing R variables directly to HDFS, using them as inputs to the MapReduce job, and, after the job has completed, loading them back into an R data structure. This isn’t the approach...
R script using RHadoop rmr to run an identity map and reduce job.

```r
# /usr/bin/env Rscript
library(rmr)

map <- function(k, v) {
  keyval(k, v)
}

reduce <- function(k, vv) {
  lapply(vv, function(v) keyval(k, v))
}

mapreduce(
  input = "stocks.txt",
  output = "output",
  textinputformat = rawtextinputformat,
  map = map,
  reduce = reduce)
```

Figure 11.7  rmr and client-side interactions

you'll want to take when working with large volumes of data, but it's great for prototyping and testing with smaller datasets:

```r
$ R
> library(rmr)
> small.ints = to.dfs(1:10)
> out = mapreduce(
  input = small.ints,
  map = function(k,v) keyval(v, v^2))
...
> result = from.dfs(out)
> print(result)
[[1]]
```
If you’re looking for additional rmr examples, the RHadoop wiki has an excellent tutorial containing examples of logical regression, K-means, and more at https://github.com/RevolutionAnalytics/RHadoop/wiki/Tutorial.

11.5 Chapter summary

The fusion of R and Hadoop allows for large-scale statistical computation, which becomes all the more compelling as both your data sizes and analysis needs grow. In this chapter we focused on two approaches you can use to combine R and Hadoop together. R and streaming provided a basic level of integration, and we also looked at RHadoop, which allows client-side R and Hadoop integration.

You should now have enough information to choose the right level of R and Hadoop integration appropriate for your project.
It’s always a good time to upgrade your Hadoop skills! *Hadoop in Practice, Second Edition* provides a collection of 104 tested, instantly useful techniques for analyzing real-time streams, moving data securely, machine learning, managing large-scale clusters, and taming big data using Hadoop.

This completely revised second edition covers changes and new features in Hadoop core, including MapReduce 2 and YARN. You’ll pick up hands-on best practices for integrating Spark, Kafka, and Impala with Hadoop, and get new and updated techniques for the latest versions of Flume, Sqoop, and Mahout. In short, this is the most practical, up-to-date coverage of Hadoop available.

**What’s Inside**
- Thoroughly updated for Hadoop 2
- How to write YARN applications
- Integrate real-time technologies like Storm, Impala, and Spark
- Predictive analytics using Mahout and RR

Readers need to know a programming language like Java and have basic familiarity with Hadoop.

**Alex Holmes** works on tough big data problems. He is a software engineer, author, speaker, and blogger specializing in large-scale Hadoop projects.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/HadoopinPracticeSecondEdition