RHadoop

Jim Harner

1/6/2020

Load the libraries for the RHadoop interface to Hadoop.

```
library(rhdfs)
```

```
## Loading required package: rJava
##
## HADOOP_CMD=/opt/hadoop/bin/hadoop
##
## Be sure to run hdfs.init()
```

```
library(rmr2)
```

```
## Please review your hadoop settings. See help(hadoop.settings)
```

```
hdfs.init()
```

4.7 RHadoop

RHadoop is a collection of R packages developed by Revolution Analytics (now owned by Microsoft) that allow users to manage and analyze data with Hadoop.

MapReduce is a powerful programming framework for efficiently processing very large amounts of data stored in the Hadoop distributed filesystem (HDFS). RHadoop is tuned to the needs of data analysts who typically work in the R environment as opposed to general-purpose languages like Java.

RHadoop provides an R package called rmr2, whose goals are:

- To provide map-reduce programmers an easy, productive, and elegant way to write MapReduce jobs. Programs written using the `rmr2` package may need one-two orders of magnitude less code than Java, while being written in a readable, reusable and extensible language.

- To give R programmers a way to access the map-reduce programming paradigm and to work on big data sets in a natural way for data analysts working in R.
  Together with its companion packages `rhdfs` and `rhbase` (for working with HDFS and HBase datastores, respectively, in R) the `rmr2` package provides a way for data analysts to access massive, fault-tolerant parallelism without needing to master distributed programming. By providing an abstraction layer on top of all of the Hadoop implementation details, the `rmr2` package lets the R programmer focus on the data analysis of very large data sets.

Initially, we must put data into HDFS (or HBase) for analysis. This can be done in R (rather than the UNIX command line with `hadoop` or `hbase`) using the RHadoop package `rhdfs`. Once the library is loaded, it must be initiated.

You must have an account on our installed HDFS. The directories and files in my home directory are listed by the `hdfs.ls` function.

```
hdfs.ls(path="/user/rstudio")
```

```
## NULL
```

For a full list of hdfs functions see the help pages for the `rhdfs` package.

4.7.1 My first mapreduce job

`mapreduce` is not very different than a combination of `lapply` and `tapply`: transform elements of a list, compute an index (key in mapreduce jargon) and process the resulting groups.

```
# Using sapply:
small.ints = 1:100
sapply(small.ints, function(x) x^2)
```

```
##   [1]     1     4     9    16    25    36    49    64    81   100   121   144
##  [13]   169   196   225   256   289   324   361   400   441   484   529   576
##  [25]   625   676   729   784   841   900   961  1024  1089  1156  1225  1296
##  [37]  1369  1444  1521  1600  1681  1764  1849  1936  2025  2116  2209  2304
##  [49]  2401  2500  2601  2704  2809  2916  3025  3136  3249  3364  3481  3600
##  [61]  3721  3844  3969  4096  4225  4356  4489  4624  4761  4900  5041  5184
##  [73]  5329  5476  5625  5776  5929  6084  6241  6400  6561  6724  6889  7056
##  [85]  7225  7396  7569  7744  7921  8100  8281  8464  8649  8836  9025  9216
##  [97]  9409  9604  9801 10000
```

```
# Using mapreduce: note that rmr2 has its own interface with hdfs
small.ints = to.dfs(1:100)

out.data <- mapreduce(
  input = small.ints,
  map = function(k, v) cbind(v, v^2))

out.data <- from.dfs(out.data)
str(out.data)
```

```
## List of 2
##  $ key: NULL
##  $ val: num [1:100, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "v" ""
```

```
head(out.data$val)
```

```
##      v
## [1,] 1  1
## [2,] 2  4
## [3,] 3  9
## [4,] 4 16
## [5,] 5 25
## [6,] 6 36
```

The first line puts the data into HDFS, where the bulk of the data has to reside for `mapreduce` to operate on. Don't use `to.dfs` to write out big data since it is not scalable. `to.dfs` is nonetheless very useful for a variety of uses like writing test cases, learning and debugging. `to.dfs` can put the data in a file of your own choosing, but if you don't specify one it will create temp files and clean them up when done. The return value is something we call a `big.data.object`. You can assign it to variables, pass it to other `rmr` functions, `mapreduce` jobs, or read it back in. It is a stub, i.e., the data is not in memory, only some information that helps finding and managing the data.

The second line, i.e., `mapreduce`, replaces `lapply`. We prefer named arguments with mapreduce because there's quite a few possible arguments, but it's not mandatory. The input is the variable `small.ints` which contains the output of `to.dfs`. This is a stub for our small number data set in its HDFS version, but it could be a file path or a list containing a mix of both. The function to apply, which is called a `map` function as opposed to the `reduce` function, which we are not using here, is a regular R function with a few constraints:

1. It's a function of two arguments, a collection of keys and one of values.

2. It returns key-value pairs using the function `keyval`, which can have vectors, lists, matrices or data.frames as arguments; you can also return `NULL`. You can avoid calling `keyval` explicitly but the return value `x` will be converted with a call to `keyval(NULL,x)`. This is not allowed in the `map` function when the `reduce` function is specified and under no circumstance in the `combine` function, since specifying the key is necessary for the shuffle phase.

The return value is `big.data.object`, and you can pass it as input to other jobs or read it into memory with `from.dfs`. `from.dfs` is complementary to `to.dfs` and returns a key-value pair collection. `from.dfs` is useful in defining map reduce algorithms whenever a mapreduce job produces something of reasonable size, like a summary, that can fit in memory and needs to be inspected to decide on the next steps, or to visualize it. It is much more important than `to.dfs` in production work.

4.7.2 My second mapreduce job

We've just created a simple job that was logically equivalent to a `lapply` but can run on big data. That job had only a map. This example has both a map and a reduce phase. The closest equivalent in R is arguably a `tapply`.

```
groups = rbinom(32, n = 200, prob = 0.4)
groups
```

```
##   [1] 12 12 12 15 17 11 10 15 15 15 14 10 12 15 20 19 15  7 16 16 13 14 12 14 11
##  [26] 11 12 12 13  9  9 11 11 16 14 13 11  8 15 15  5 13 13 13 17 12 10 10  9 13
##  [51]  9 14 14 15 13 11 13 12 14 12 14 13  9 13 15 11 13  9 13 11 11  9 12  9 13
##  [76] 14 14 11  8 12 15 14 10 11 12 15 13 20 10 14 17 11 10 12  7 12 11 12 14  9
## [101]  4 13 17  8 10 14 13 10 10 16 13 12 16 12 10 17 12 17 14 15  9 13  9 11 16
## [126] 15 12 15 13 10 11 11 14 11 13 11 11 10 14 12 13 14 12 19 11 13 14  9  8 10
## [151] 16 18 12 13 13 14 16 13 10 11 18  8 14 10 14  7 15 11 16 19 12 14  9 13 19
## [176] 15 11 13 15 10 15 14 16 13 12 11 11 14 16 14 16  7 12 11  8  8 13 13  8  9
```

```
tapply(groups, groups, length)
```

```
##  4  5  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##  1  1  4  8 14 17 27 26 31 26 19 12  6  2  4  2
```

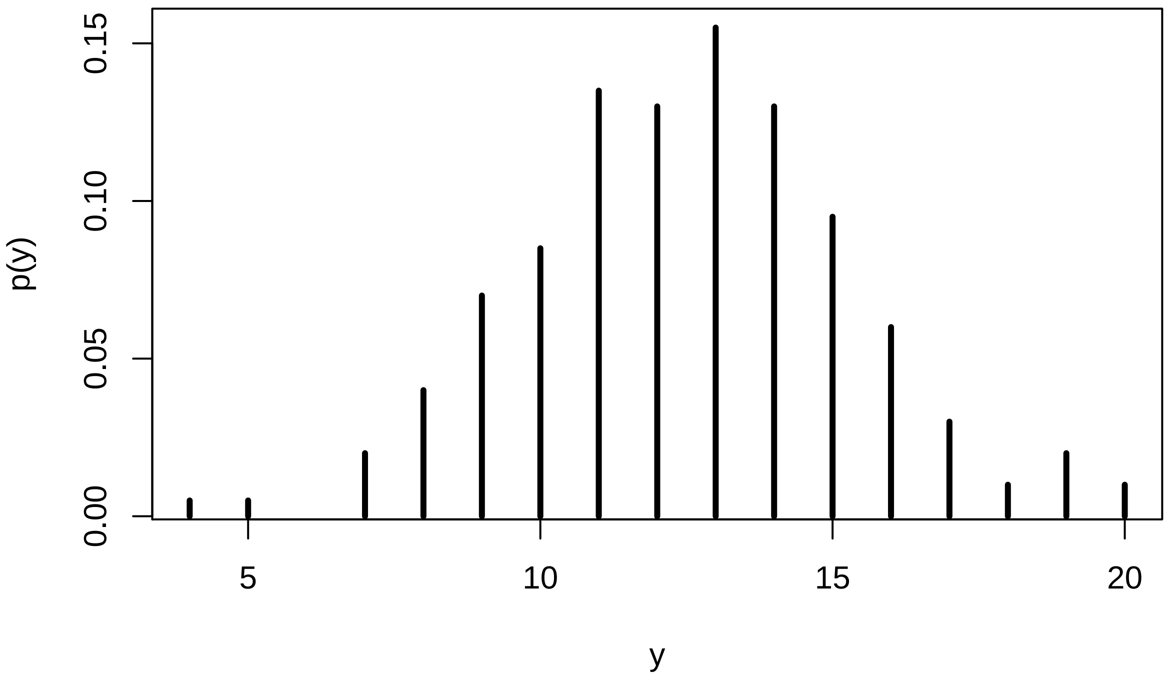
This creates a sample from the binomial and counts how many times each outcome occurred.

```
groups = to.dfs(groups)

out.data <- from.dfs(
  mapreduce(
    input = groups,
    map = function(., v) keyval(v, 1),
    reduce = function(k, vv)  keyval(k, length(vv))))
out.data
```

```
## $key
##  [1]  4  5  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $val
##  [1]  1  1  4  8 14 17 27 26 31 26 19 12  6  2  4  2
```

```r
plot(out.data$key, out.data$val/200, type = "h", lwd=3,
     xlab = "y", ylab = "p(y)")
```



First we move the data into HDFS with `to.dfs`. Normally big data enters HDFS with scalable data collection systems such as Flume or Sqoop. In that case we would just specify the HDFS path to the data as input to `mapreduce`. In this case the input is the variable `groups`, which contains a `big.data.object`, which keeps track of where the data is and does the clean up when the data is no longer needed.

The `map` function is set to the default, which is like an identity but consistent with the `map` requirements, i.e., `function(., v) keyval(k, 1)`.

The `reduce` function takes two arguments, one is a key and the other is a collection of all the values associated with that key. It could be one of vector, list, data frame or matrix depending on what was returned by the `map` function. The idea is that if the user returned values of one class, we should preserve that through the shuffle phase.

As in the `map` case, the `reduce` function can return `NULL`, a key-value pair generated by the function `keyval` or any other object `x` which is equivalent to `keyval(NULL, x)`. The default is no `reduce`, that is the output of the `map` is the output of `mapreduce`. In this case the keys are realizations of the binomial and the values are all 1. Since we want to know how many there are, we count using `length`. Looking back at this second example, there are small differences with `tapply` but the overall complexity is similar.

4.7.3 Word Count

We define a function, `wordcount`, that encapsulates this job. Our main goal was not simply to make it easy to run a mapreduce job but to make mapreduce jobs first class citizens of the R environment and to make it easy to create abstractions based on them. For instance, we wanted to be able to assign the result of a mapreduce job to a variable and to create complex expressions including mapreduce jobs. We take the first step here by creating a function that is itself a job, which can be chained with other jobs, executed in a loop etc.

```r
wordcount =
  function(input, output = NULL, pattern = " "){
    wc.map =
```

```r
      function(., lines) {
        keyval(unlist(strsplit(x = lines, split = pattern)), 1)
        }
    wc.reduce =
      function(word, counts ) {
        keyval(word, sum(counts))
        }
    mapreduce(input = input, output = output,
              map = wc.map, reduce = wc.reduce, combine = TRUE)
  }
```

Capture the R license as text.

```r
text = capture.output(license())
text
```

```
##  [1] ""
##  [2] "This software is distributed under the terms of the GNU General"
##  [3] "Public License, either Version 2, June 1991 or Version 3, June 2007."
##  [4] "The terms of version 2 of the license are in a file called COPYING"
##  [5] "which you should have received with"
##  [6] "this software and which can be displayed by RShowDoc(\"COPYING\")."
##  [7] "Version 3 of the license can be displayed by RShowDoc(\"GPL-3\")."
##  [8] ""
##  [9] "Copies of both versions 2 and 3 of the license can be found"
## [10] "at https://www.R-project.org/Licenses/."
## [11] ""
## [12] "A small number of files (the API header files listed in"
## [13] "R_DOC_DIR/COPYRIGHTS) are distributed under the"
## [14] "LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later."
## [15] "This can be displayed by RShowDoc(\"LGPL-2.1\"),"
## [16] "or obtained at the URI given."
## [17] "Version 3 of the license can be displayed by RShowDoc(\"LGPL-3\")."
## [18] ""
## [19] "'Share and Enjoy.'"
## [20] ""
```

Technically, we should clean up the text file since "3" and "3,", etc. will be counted as separate words, but this can also be done after getting the output from Hadoop.

```r
out = list()
rmr.options(backend = "hadoop")
```

```
## NULL
```

```r
word.df <- to.dfs(keyval(NULL, text))
word.out <- wordcount(word.df, pattern = " ")
out[["hadoop"]] <- from.dfs(word.out)
out[["hadoop"]]
```

```
## $key
##  [1] "2"
##  [2] "3"
##  [3] "A"
##  [4] "a"
##  [5] "2,"
##  [6] "3,"
```

```
##  [7] "at"
##  [8] "be"
##  [9] "by"
## [10] "in"
## [11] "is"
## [12] "of"
## [13] "or"
## [14] "2.1"
## [15] "API"
## [16] "GNU"
## [17] "The"
## [18] "URI"
## [19] "and"
## [20] "are"
## [21] "can"
## [22] "the"
## [23] "you"
## [24] "(the"
## [25] "1991"
## [26] "June"
## [27] "This"
## [28] "both"
## [29] "file"
## [30] "have"
## [31] "this"
## [32] "with"
## [33] "2007."
## [34] "files"
## [35] "found"
## [36] "small"
## [37] "terms"
## [38] "under"
## [39] "which"
## [40] "'Share"
## [41] "Copies"
## [42] "LESSER"
## [43] "PUBLIC"
## [44] "Public"
## [45] "called"
## [46] "either"
## [47] "given."
## [48] "header"
## [49] "later."
## [50] "listed"
## [51] "number"
## [52] "should"
## [53] "COPYING"
## [54] "Enjoy.'"
## [55] "GENERAL"
## [56] "General"
## [57] "Version"
## [58] "license"
## [59] "version"
## [60] "LICENSE,"
```

```
## [61] "License,"
## [62] "obtained"
## [63] "received"
## [64] "software"
## [65] "versions"
## [66] "displayed"
## [67] "distributed"
## [68] "RShowDoc(\"GPL-3\")."
## [69] "RShowDoc(\"LGPL-3\")."
## [70] "RShowDoc(\"COPYING\")."
## [71] "RShowDoc(\"LGPL-2.1\"),"
## [72] "R_DOC_DIR/COPYRIGHTS)"
## [73] "https://www.R-project.org/Licenses/."
##
## $val
##  [1] 2 3 1 1 1 1 2 5 4 2 1 8 3 1 1 2 1 1 3 2 5 8 1 1 1 2 2 1 1 1 1 1 1 2 1 1 2 2
## [39] 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 2 1 1 1 1 1 2 1 4 2 1 1 1 1 1 1
```

The `map` function, as we know already, takes two arguments, a key and a value. The key here is not important, indeed always `NULL`. The value contains several lines of text, which gets split according to a pattern. Here you can see that pattern is accessible in the mapper without work on the programmer side and according to normal R scope rules.

This apparent simplicity hides the fact that the `map` function is executed in a different interpreter and on a different machine than the `mapreduce` function. Behind the scenes the R environment is serialized, broadcast to the cluster and restored on each interpreter running on the nodes. For each word, a key value pair `(w, 1)` is generated with `keyval` and their collection is the return value of the mapper.

The `reduce` function takes a key and a collection of values, in this case a numeric vector, as input and simply sums up all the counts and returns the pair word and count using the same helper function, `keyval`. Finally, specifying the use of a combiner is necessary to guarantee the scalability of this algorithm.

The implementation defines `map` and `reduce` functions and then makes a single call to `mapreduce`. The `map` and `reduce` functions could be anonymous functions as they are used only once, but there is one advantage in naming them. `rmr` offers alternate backends, in particular one can switch off Hadoop altogether with `rmr.options(backend = "local")`.

The `input` can be an HDFS path, the return value of `to.dfs` or another job or a list—potentially, a mix of all three cases, as in `list("a/long/path", to.dfs(...), mapreduce(...), ...)`. The `output` can be an HDFS path but if it is `NULL` a temporary file will be generated and wrapped in a big data object, like the ones generated by `to.dfs`. In either event, the job will return the information about the output, either the path or the big data object.

Therefore, we simply pass along the input and output of the `wordcount` function to the `mapreduce` call and return whatever its return value. That way the new function also behaves like a proper mapreduce job. The `input.format` argument allows us to specify the format of the input. The default is based on R's own serialization functions and supports all R data types. In this case we just want to read a text file, so the "text" format will create key value pairs with a `NULL` key and a line of text as value. You can easily specify your own input and output formats and even accept and produce binary formats with the functions `make.input.format` and `make.output.format`.

This discussion should make it clear that RHadoop is very flexible in the data structures is can take as input, pass from map to reduce, and return as output. In this sense, RHadoop greatly generalizes Hadoop Streaming.

```r
options(warn=0)
```