Business Intelligence in R

– SEMINAR SUMMER SEMESTER 2016 –

How to utilize Hadoop from within the statistical software "R"

– SEMINAR PAPER –

Submitted by:
Axel Perschmann

Date:
22. July 2016

Advisers:
Prof. Dr. Dirk Neumann
Dr. Stefan Feuerriegel
Nicolas Pröllochs

Contents

Abstract

The programming language R is a very popular and powerful tool for statisticians, but by itself not designed for big data exceeding the local machines memory and processing capabilities. When analyzing large amounts of data, this quickly becomes a serious and unacceptable restriction. This paper gives an outline about the large-scale data management system Apache Hadoop, it's underlying data storage and processing concepts and how it can be utilized from within R to overcome this restriction. To demonstrate the usefulness of Hadoop, a real world example from the field of stock trading is discussed in detail.

1 Motivation

In a world of ever growing data, we quickly come to a point where analytic computations on a single, local workstation are no longer feasible. To overcome the limited capabilities of a single machine, specialized frameworks, such as Apache Hadoop, have emerged and paved the way for big data analysis through distributed storage and computation across multiple computers.

One exemplary use case, where statisticians can benefit from a distributed setting, as provided by Hadoop, is the analyzation of stock exchange data. Market place organizers, such as Deutsche Börse, hold millions of historic transaction details for a great number of stocks. The size of such data sets can easily grow into the range of Petabytes, hampering local-only analyzation.

The remainder of this paper is structured as follows: Section 2 gives a general introduction of the programming language R and the Apache Hadoop framework, Section 3 describes how to utilize Hadoop from within R to combine their capabilities and Section 4 covers a real world example from the field of stock trading in detail. Section 5 closes with a short conclusion.

2 Background

After a brief introduction into the programming language R, this section gives an outline about the large-scale data management system Apache Hadoop and its underlying concepts for distributed data storage and data processing.

2.1 R

R [8] is a free programming language with a strong focus on statistical computing and graphics. While a fresh installation of R provides only basic functionality, it's capabilities can easily be extended through extra packages [13]. There is a strong community of R users actively involved in the development and maintenance of additional functionality to the R language. The distributed Comprehensive R Archive Network (CRAN) platform currently (June 2016) lists more than 8500 [12] additional packages, while further packages can be found on repositories as e. g. GitHub.

2.1.1 RStudio

While R comes with a command line interpreter, there exist powerful and productive user interfaces. Probably the most popular one is RStudio [10], which is open source and available for Windows, Mac and Linux. It comes in two editions: RStudio Desktop and RStudio Server, while the latter one runs on a remote Linux server and allows accessing RStudio using a web browser.

2.2 Hadoop

Figure 1: Apache Hadoop [2].

Apache Hadoop is a large-scale data management system maintained and released by the Apache Software Foundation under open source licence. It is a framework designed for the distributed storage and processing of very large data sets (Petabytes) across computer clusters [2] and provides a highly extendable ecosystem as shown in Figure 2. The projects official logo is shown in Figure 1.

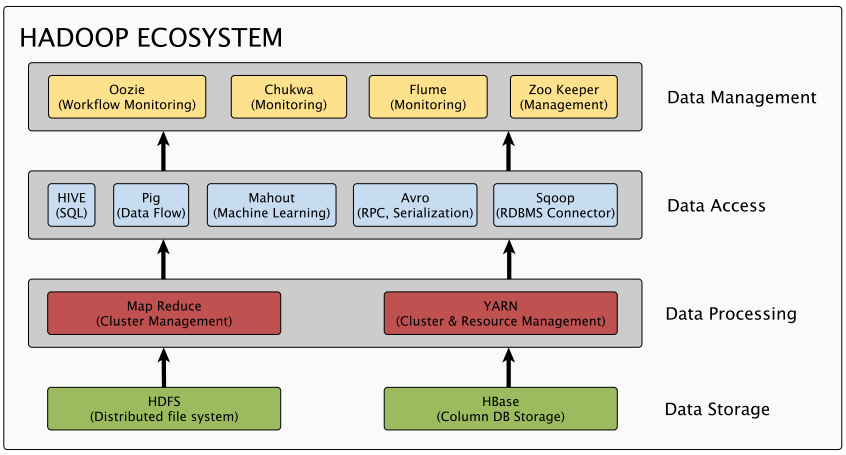

Figure 2: Hadoop Ecosystem.

Built upon simple programming models, Hadoop can be scaled from single nodes to thousands of machines and runs on low-cost commodity hardware which is usually more vulnerable for failures than expensive, specialized hardware. To guarantee a highly-available service, the framework is designed to automatically detect and handle such failures.

What makes Hadoop different from traditional parallelism frameworks is its solution to the annoying I/O bottleneck typically arising in big data computations. Rather than bringing the data to the computing node, it brings the computation to the data. The fundaments of Apache Hadoop basically consists of the storage part Hadoop Distributed File System (HDFS) and the processing part MapReduce, which are in the focus of this paper.

2.2.1 Hadoop Distributed File System (HDFS)

The storage part, HDFS, is a highly fault-tolerant filesystem designed to run on low-cost hardware which is inspired by the Google File System (GFS)[4]. HDFS provides high-throughput access to application data and is tuned to support very large files. The files are split into large blocks (e. g. 64 MB), replicated for reliability and distributed across various nodes in a cluster as illustrated in Figure 3.

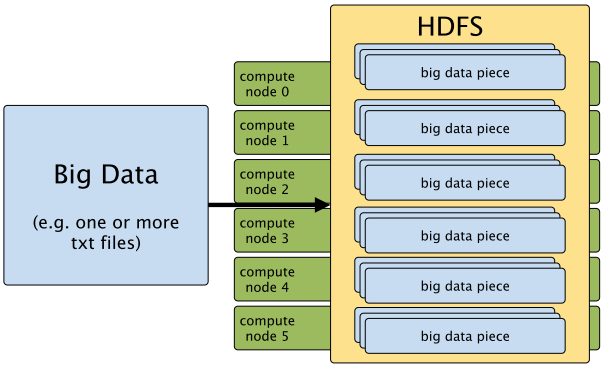

Figure 3: File distribution across multiple physical compute nodes.

Typical file manipulation commands as e.g. `ls, mkdir, rm, mv, cp, cat, tail, chmod, ...` can be used to interact with files, while HDFS takes care of recombining the data appropriately [5]. These commands can be run via the Hadoop command `hadoop dfs [GENERIC_OPTIONS] [COMMAND_OPTIONS]`.

Example
```
hdfs dfs -ls /user/isresearch/
hdfs dfs -put ./1017_01_M_08_E_20090331.csv /user/isresearch/
```

2.2.2 Hadoop MapReduce

The processing part, Hadoop MapReduce, is an implementation of Googles MapReduce programming model [3] and is composed of three phases:

Map() In the first phase, each worker node applies the *Map()* function to it's local part of the data. The data can be filtered and sorted in any desired form and will eventually be converted into zero or more key-value pairs.

Shuffle() In the second phase, the results from the previous *Map()* phase are grouped according to their given key and stored back into the HDFS. The Filesystem takes care of the redistribution of the intermediate results, such that all data of one key is located on the same physical node.

Reduce() In the last phase the worker nodes apply the *Reduce()* function on each group of intermediate results individually and in parallel. The *Reduce()* function performs a summary operation, such as counting the number of respective data points.

In contrast to traditional parallelism models, MapReduce performs the computation where the data resides, rather than transferring the data to the computation. As such, it avoids the previously mentioned I/O bottleneck, which arises from limited bandwidth and transportation cost induced by huge data files. Note that the individual phases must be complete before a succeeding phase can start. The typical MapReduce workflow is illustrated in Figure 4.

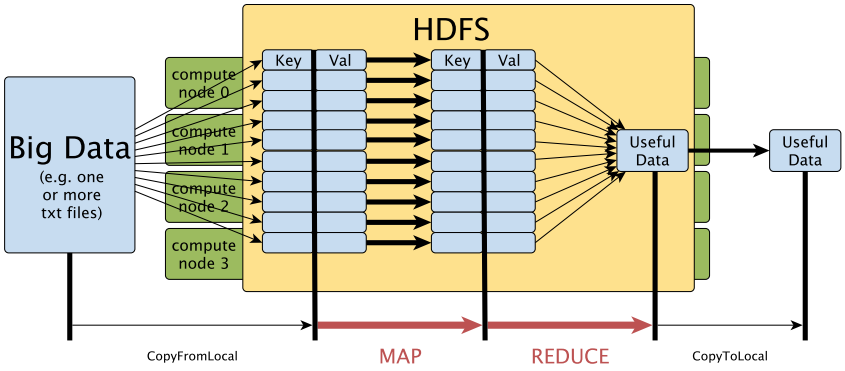

Figure 4: MapReduce Workflow.

2.2.3 Hadoop Streaming

Hadoop Streaming [6] is a generic API which comes with Hadoop. It allows Map and Reduce functions to be written in any language. Both functions read their respective input from stdin (line by line) and pass their output to stdout. The corresponding streaming format is shown in Listing 1. The first part of each line until a delimiter (default: tab character) serves as the *key*, while the rest of the line will be the *value*. The output is always sorted by key.

Listing 1: Default Hadoop Streaming format.

```
key1 \t value1 \n
key2 \t value2 \n
key3 \t value3 \n
```

3 Utilizing Hadoop from within R

When analyzing big data, the capabilities of R and Hadoop fit together very well and should ideally be united. In the following, this paper describes how to utilize Hadoop from within the statistical software "R".

3.1 RHadoop

The open source RHadoop project [9], sponsored by Revolution Analytics, aims to give R programmers powerful tools to analyze data stored in Hadoop. RHadoop is a collection of five R packages as listed in Table 1.

The focus of this paper lies on the RHadoop packages *rhdfs* and *rmr2*, which allow R programmers to access the HDFS and to perform statistical analysis in R via Hadoop MapReduce functionality on a Hadoop cluster. Detailed install instructions can be found on the projects website [9].

rhdfs	This package provides basic connectivity to the Hadoop Distributed File System. R programmers can browse, read, write, and modify files stored in HDFS from within R. Install this package only on the node that will run the R client.
rhbase	This package provides basic connectivity to the HBASE distributed database, using the Thrift server. R programmers can browse, read, write, and modify tables stored in HBASE, Hadoops distributed, scalable big data store database [11]. Install this package only on the node that will run the R client.
plyrmr	This package enables the R user to perform common data manipulation operations, as found in popular packages such as plyr and reshape2, on very large data sets stored on Hadoop. Like rmr, it relies on Hadoop MapReduce to perform its tasks, but it provides a familiar plyr-like interface while hiding many of the MapReduce details. Install this package only every node in the cluster.
rmr2	A package that allows R developer to perform statistical analysis in R via Hadoop MapReduce functionality on a Hadoop cluster. Install this package on every node in the cluster.
ravro	A package that adds the ability to read and write avro files from local and HDFS file system and adds an avro input format for rmr2. Install this package only on the node that will run the R client.

Table 1: RHadoop is a colletion of five R packages [9].

3.2 rhdfs

Once *rhdfs* is installed and the package is loaded, R programmers can browse, read, write and modify files stored in HDFS. Amongst others the package offers functions for file manipulation (`hdfs.delete, hdfs.put, hdfs.get, ...`), read/write (`hdfs.file, hdfs.write, hdfs.seek, ...`) and utility (`hdfs.ls, hdfs.file.info, ...`).

As a prerequisite the environment variable HADOOP_CMD must point to the full path of the *hadoop* binary and the package must be initialized via `hdfs.init()`. The corresponding code to define an environment variable, load the rhdfs library and initialize its functionality is shown in Listing 2.

Listing 2: Initialization of rhdfs.

```
Sys.setenv(HADOOP_CMD="/usr/local/hadoop/bin/hadoop")
library(rhdfs)
hdfs.init()
```

Listing 3 shows exemplary how any R object can be serialized to HDFS. Line 4 establishes a write connection (`"w"`) to a, possibly not yet existing, file called `"my_smart_unique_name"`. The R object `model` is serialized and written into this file in Line 4. Finally the connection to that file must be closed.

Listing 3: Serialization of an exemplary R object [9].

```
model <- lm(...)
modelfilename <- "my_smart_unique_name"
modelfile <- hdfs.file(modelfilename, "w")
hdfs.write(model, modelfile)
hdfs.close(modelfile)
```

In contrast Listing 4 shows the deserialization of an HDFS file to a R object. Here, a read connection (`"r"`) to the desired HDFS file is established to load its content into the R environment. Subsequently the retrieved content must be unserialized (Line 3) and again the connection must be closed.

Listing 4: Deserialization of an exemplary R object [9].

```
modelfile = hdfs.file(modelfilename, "r")
m <- hdfs.read(modelfile)
model <- unserialize(m)
hdfs.close(modelfile)
```

3.3 rmr2

With the help of *rmr2* programmers can utilize the Hadoop MapReduce functionality from within R to perform statistical analysis on huge data sets on a parallel basis.

Similar as for *rhdfs* it is necessary to set the environment variables HADOOP_CMD and HADOOP_STREAMING, the latter one pointing to the Hadoop Streaming Java[1] Library. An example initialization is show in Listing 5.

Listing 5: Initialization of the rmr2 package.

```
Sys.setenv(HADOOP_CMD="/usr/local/hadoop/bin/hadoop")
Sys.setenv(HADOOP_STREAMING
              ="/usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar")
library(rmr2)
rmr.options(backend="hadoop")
```

Via `rmr.options()` it is possible to switch between the distributed Hadoop backend (see Line 5) and a non-distributed local backend. With `backend="local"` rmr2 jobs run single threaded on the local interpreter which helps a lot when debugging. By default, all nodes will receive a copy of the global R environment, i. e. they can read previously initialized variables. To reduce transfer costs the option `rmr.options(exclude.objects=NULL)` can exclude certain objects from this practice.

The most important function provided by rmr2 is `mapreduce()`. Listing 6 shows the functions signature as documented in the R documentation. The function defines and runs a MapReduce job on a given input (either an object or a valid path string). Various parameters allow an individual fine tuning of the MapReduce job. E. g.:

input Paths to the input folder(s) or the return value of another mapreduce or a `to.dfs`[2] call.

input.format `= make.input.format("csv", sep = ";")` defines the format and structure of the input file(s).

output defines the destination path in the HDFS. If absent, `mapreduce()` returns a `big.data.object`[3], otherwise it returns the HDFS filepath.

1 Apache Hadoop is implemented in Java
2 `to.dfs()` and `from.dfs()` allow to move data from RAM to HDFS and back. [R help of rmr2, 9]
3 A `big.data.object` references a data set stored in the HDFS. As such even huge data sets that exceed the nodes memory can be manipulated by other rhadoop functions. [R help of rmr2, 9]

map an R function of two arguments specifying the map phase. It either returns `NULL` or a collection of key-value pairs.

reduce an R function which takes as an input the key-value pairs generated by the preceding map phase grouped by key.

Listing 6: function signature of `mapreduce()` as described in the rmr2 help (`?mapreduce`).

```
mapreduce(
  input,
  output = NULL,
  map = to.map(identity),
  reduce = NULL,
  vectorized.reduce = FALSE,
  combine = NULL,
  in.memory.combine = FALSE,
  input.format = "native",
  output.format = "native",
  backend.parameters = list(),
  verbose = TRUE)
```

3.4 Word Count Example

Listing 7 shows a simple "Hello World!" example for a MapReduce job [1]. The jobs intend is to return the number of word occurrences within a given input text file. The input file `loremipsum.txt` contains 100 words of dummy text and as such `input.format` is simply `"text"`.

Our map function, defined in Line 5, reads the input line by line and splits each line into individual words. Using the individual words as key and integer one as a counter variable, it produces corresponding key-value pairs. In the next phase, the mapper function, defined in Line 8, receives the grouped key-values and simply outputs the sum over all counter 'ones' for each key. In Line 12 the final result is moved into the R environment and can be viewed or further processed.

Listing 7: Hello World example for MapReduce [1].

```
library(rmr2)
rmr.options(backend = "local")
data <- mapreduce(input = "loremipsum.txt",
                  input.format = "text",
                  map = function(.,lines){
                    keyval(key = unlist(strsplit(lines, split = " ")),
                           val = 1) },
                  reduce = function(word, counts){
                    keyval(key = word,
                           val = sum(counts)) }
)
result = from.dfs(data)
result$key
 [1] "Lorem"      "ipsum"      "dolor"      "sit"        "amet,"      "consetetur"  ...
result$val
 [1] 4 4 4 4 2 2 2 2 4 4 2 2 2 2 2 2 8 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

4 Real World Example: Stock Exchange Data

This chapter is about a concrete, real world example. The seminars secondary task was to analyze stock exchange data to find out how fast automated traders respond to published ad hoc messages. The complete code for this example can be found online at [7] and is split into several R scripts:

0_helloWorld.R	Initial hello world example (Section 3.4)
1_dataToHDFS.R	Unzip and transfer stock data into the HDFS (Section 4.1.1)
2_parse_adHocXML.R	Parsing of available adhoc messages (Section 4.1.2)
3a_hadoop_checkTimeZone.R	Little experiment to verify correct timezone (Section 4.1.1)
3_hadoop.R	Defining and running the MapReduce job (Section 4.2)
4_evaluation.R	Evaluation and plotting of results (Section 4.3)

4.1 Input Data

4.1.1 Stock exchange data from the Deutsche Börse

The analyzation is based on 166 files (200+ GB) of tick data, obtained from the universities exclusive access to the Deutsche Börse. Exact to the hundredth of a second they contain millions of transitions, each encoded as follows:

`WKN, ISIN`	Instrument identifiers
`INSTRUMENT_NAME`	Instrument name
`TIMESTAMP, HSEC`	Exact timestamp
`PRICE`	Current price
`UNITS`	Number of units
`BID_ASK_FLAG`	BID or ASK

An excerpt from one of the files is shown in Listing 8.

Listing 8: Excerpt from monthly_bba_aa_20090331.csv.

```
                                                                                BID_ASK
WKN    ;ISIN         ;INSTRUMENT_NAME           ;TIMESTAMP           ;HSEC;PRICE;UNITS;_FLAG
940602;NL0000009538;KON.PHILIPS.ELECT.  EO-20;2009-03-02 12:04:00; 92 ;12.06; 8572;A
843002;DE0008430026;MUENCH.RUECKVERS.VNA O.N.;2009-03-02 12:04:00; 94 ;92.34;   46;A
840400;DE0008404005;ALLIANZ SE VNA O.N.       ;2009-03-02 12:04:00; 94 ;50.89;   90;A
LYX0A0;FR0010344986;LYXOR ETF DJ ST. 600 RET.;2009-03-02 12:04:00; 94 ;18.29;10000;B
```

In the early phase of testing the subsequent MapReduce job on only one of these 166 input files (monthly_bba_aa_20090331.csv) very little trade activity could be observed. As the given timestamps do not include a timezone definition a little experiment was performed to exclude the case of a miss interpretation. *3a_hadoop_checkTimeZone.R* analyzes how trade activity is distributed in terms of daytime. Figure 5 clearly depicts that there is no offset in between observed trades and the trading hours of Xetra which are between 8.50 a.m. CET and 5.30 p.m. CET [14].

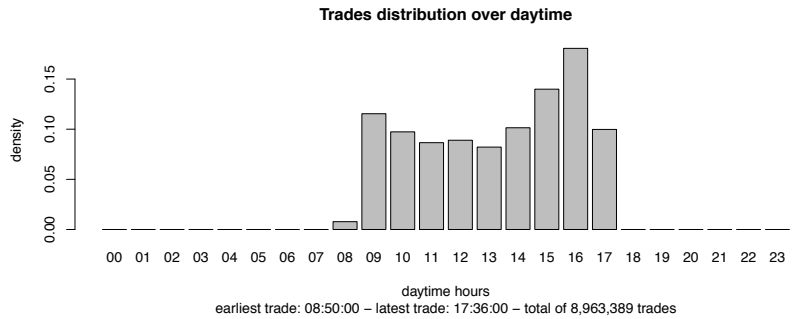

Figure 5: Trade distribution within monthly_bba_aa_20090331.csv.

Transfer into HDFS

All 166 files came as compressed zip files. Listing 9 shows how these files were unzipped and transferred into the HDFS.

As a first step a new directory on the HDFS was created using the `hdfs.mkdir` command (Line 9). Then a for-loop, iterating over all 166 .zip files found in the local directory `DeutscheBoerse/`, unzips the contained csv files into a temporary folder `DeutscheBoerse/tmp/` and finally transfers the data into the HDFS using the `hdfs.put` command (Line 21). Because the zipped files consume only 260 MB while the unzipped versions take 1.35 GB the latter ones were immediately discarded after they had been transferred to the HDFS.

Listing 9: 1_dataToHDFS.R.

```
Sys.setenv(HADOOP_CMD="/usr/local/hadoop/bin/hadoop")
library(rhdfs)
hdfs.init()

setwd("DeutscheBoerse")
input.names = list.files(path = ".", pattern=".*zip")

# create new directory on HDFS
hdfs.mkdir("/user/isresearch/Data")

# push files to HDFS. Only necessary once!
for (name in input.names) {
  print(name)
  # unzip file and read name of extracted file
  unzip(name, exdir = "tmp/")
  source.name = list.files(path = "tmp")

  path.source = paste(getwd(), "/tmp/", source.name, sep="")
  path.dest = paste("/user/isresearch/Data/", source.name, sep="")
  # load to HDFS
  hdfs.put(src=path.source, dest=path.dest)

  # do not keep large unzipped file
  unlink(paste(getwd(), "/tmp", sep=""), recursive=TRUE)
}
```

Input format

As we want to process csv files, we have to specify the input format of our mapreduce job, as briefly mentioned in Section 3.3. In Listing 10 we define the csv file separator, the column names and that we do not want strings to be interpreted as factors. Unfortunately `make.input.format()` does not come with a parameter like `header=TRUE`, such that we have to consider this in our map function where we must discard the headerline.

Listing 10: Input format.

```
inputformat <- make.input.format("csv", sep = ";", stringsAsFactors = FALSE,
                                 col.names=c("WKN", "ISIN", "INSTRUMENT_NAME", "TIMESTAMP",
                                             "HSEC", "PRICE", "UNITS", "BID_ASK_FLAG") )
rmr.options(backend="hadoop")
files = hdfs.ls("Data")[,6]          # all filenames in dir /user/isresearch/Data/
files = files[grep(".*csv", files)] # filter for csv files only
```

4.1.2 Ad hoc messages

An additional collection of seven xml-files ($\approx$ 170 MB in total) contain individual ad hoc messages including their respective date and time of publication. The seminars task included the extraction of corresponding (`TIMESTAMP`, `ISIN`)-pairs using regular expressions. Due to the relatively small size of these xml files this sub task was solved entirely on the local machine and the results stored into a csv file.

To make the xml parsing a little complicated, the structure and content of the files vary. In some files the `ISIN` can directly be read from a field called `"isin"`, whereas other files are lacking this field and therefore enforce the use of a regular expression search over the whole message text. Furthermore some messages include only dates and text and do not mention any `ISIN`. Those messages were discarded and ignored by *2_parse_adHocXML.R*. The final version of the regular expression, utilized to extract an `ISIN` from the raw ad hoc message text, is shown in Listing 11.

Listing 11: Regular Expression to extract an ISIN from text.

```
expr = paste("(XS|AD|AE|AF|AG|AI|AL|AM|AO|AQ|AR|AS|AT|AU|AW|AX|AZ|BA|BB|BD|",
             "BE|BF|BG|BH|BI|BJ|BL|BM|BN|BO|BQ|BR|BS|BT|BV|BW|BY|BZ|CA|CC|",
             "CD|CF|CG|CH|CI|CK|CL|CM|CN|CO|CR|CU|CV|CW|CX|CY|CZ|DE|DJ|DK|",
             "DM|DO|DZ|EC|EE|EG|EH|ER|ES|ET|FI|FJ|FK|FM|FO|FR|GA|GB|GD|GE|",
             "GF|GG|GH|GI|GL|GM|GN|GP|GQ|GR|GS|GT|GU|GW|GY|HK|HM|HN|HR|HT|",
             "HU|ID|IE|IL|IM|^IN|IO|IQ|IR|IS|IT|JE|JM|JO|JP|KE|KG|KH|KI|KM|",
             "KN|KP|KR|KW|KY|KZ|LA|LB|LC|LI|LK|LR|LS|LT|LU|LV|LY|MA|MC|MD|",
             "ME|MF|MG|MH|MK|ML|MM|MN|MO|MP|MQ|MR|MS|MT|MU|MV|MW|MX|MY|MZ|",
             "NA|NC|NE|NF|NG|NI|NL|NO|NP|NR|NU|NZ|OM|PA|PE|PF|PG|PH|PK|PL|",
             "PM|PN|PR|PS|PT|PW|PY|QA|RE|RO|RS|RU|RW|SA|SB|SC|SD|SE|SG|SH|",
             "SI|SJ|SK|SL|SM|SN|SO|SR|SS|ST|SV|SX|SY|SZ|TC|TD|TF|TG|TH|TJ|",
             "TK|TL|TM|TN|TO|TR|TT|TV|TW|TZ|UA|UG|UM|US|UY|UZ|VA|VC|VE|VG|",
             "VI|VN|VU|WF|WS|XF|YE|YT|ZA|ZM|ZW)([\\s]?)",
             "(?=.{0,8}[0-9]+)([0-9A-Z]{9}[0-9]?)",
             "(?![0-9]{1})",
             sep="")
  # 2 characters:   country
  # 9 alphanumeric: NSIN
  # 1 number:       optional checksum
```

The resulting list of 33 617 distinct (TIMESTAMP, ISIN)-pairs `ev` was ordered by ISIN and TIMESTAMP successively and eventually stored into a csv file as shown in Listing 12.

Listing 12: Sorting and storage of events.csv.

```
ev = unique(data.frame(events))

# sort stuff
library(dplyr)
ev = arrange(ev, isin, date)

write.csv(ev, file="events.csv", row.names=FALSE)
```

4.2 MapReduce Job

Before defining and running the MapReduce job in Listing 13, we ensure the previously generated list `ev` is available in the global R environment such that our reduce function can access the ad hoc message dates. We further generate a smaller `isin.list` which aggregates all distinct ISINs contained in `ev`, such that our map function can discard data points not belonging to any of these ISINs.

Listing 13: ev and isin.list are loaded into the global R environment.

```
# load isin_set
ev = read.csv("events.csv", sep=",", as.is=TRUE, col.names=c("date", "ISIN"))
isin.list = as.character(unique(ev[,2]))
```

4.2.1 Map() function

Multiple, parallel running instances of the Map() function as shown in Listing 14 transfer small junks of the whole input into (key, value)-pairs.

Each instance receives it's part of the input as value matrix `v`, where the different values are accessible by their respective column name (e. g. `v$PRICE`). The key vector `k` is empty, except for the case of chaining multiple MapReduce jobs together.

Listing 14: Map() function.

```
map.ticks <- function(k, v) {
  # Skip header lines
  v = v[v$PRICE != "PRICE",]
  v$PRICE = as.double(v$PRICE)
  v = v[v$PRICE != 0.0, ] # erroneous values in data?!

  # only extract ISIN's when we have observed an AdHoc msg
  v = cbind(v, useful=v$ISIN %in% isin.list)
  v = v[v$useful == TRUE,]

  keyval(key=paste(v$ISIN, v$BID_ASK_FLAG, sep="_"),
         val=subset(v, select=c("TIMESTAMP", "PRICE")))
}
```

As mentioned in Section 4.1.1, we need to filter out the header line. In Line 3 we discard all rows whose `v$PRICE` contains the string `"PRICE"` rather than a number. Once the header is gone, we can convert column `v$PRICE` from character to double (Line 4) and discard erroneous values

(Line 5). A major reduction of the data is achieved in Line 8, where we check wether a data points ISIN is present in isin.list (useful=TRUE) or not (useful=FALSE). Only useful values will be kept. Finally we define the key as a concatenation of ISIN and BID_ASK_FLAG (e. g. "DE0007664039_A", "DK0010272632_B") and the value is further reduced to the columns of interest: TIMESTAMP and PRICE.

4.2.2 Reduce() function

After the map phase has finished, the ordered (key, value)-pairs are distributed across multiple, parallel running instances of the Reduce() function shown in Listing 15.

Listing 15: Reduce() function.

```
red.ticks <- function(k, v) {
  v$TIMESTAMP = as.POSIXct(v$TIMESTAMP, "%Y-%m-%d %H:%M:%OS", tz="CET")
  v = v[order(v$TIMESTAMP),]

  adhocs = ev[ev$ISIN == unlist(strsplit(k, "_"))[1], 1]
  adhocs = as.POSIXct(adhocs, "%Y-%m-%d %H:%M:%OS", tz="CET")
  adhocs = adhocs[!is.na(adhocs)]

  # only keep adhoc dates which lay in the interval of available tick data
  adhocs.relevant = adhocs[adhocs > v$TIMESTAMP[1]]
  adhocs.relevant = adhocs.relevant[adhocs.relevant < tail(v$TIMESTAMP, n=1)]

  if (length(adhocs.relevant) > 0) {
    # compute price deltas for fixed intervals
    l = length(adhocs.relevant)
    offset.seconds = c(0, 1, 5, 10, 30, 60, 60*5, 60*10, 60*60)
    timestamps = matrix(rep(adhocs.relevant,length(offset.seconds)), nrow=l)
    offset = matrix(rep(offset.seconds, l), byrow=TRUE, nrow=l)
    timestamps.future = timestamps + offset

    fun = stepfun(v$TIMESTAMP, c(v$PRICE, tail(v$PRICE, n=1)), f=0, right=TRUE)

    # collect corresponding PRICE for each TIMESTAMP
    result = c()
    for (i in 1:l) {
      x = fun(timestamps.future[i,])
      result = rbind(result, x)
    }
    colnames(result) = paste("+", offset.seconds, "s", sep="")

    # calculate PRICE delta over time
    delta = result / result[,1]
    colnames(delta) = paste("s", offset.seconds, sep="")
    rownames(delta) = make.names(adhocs.relevant, unique=TRUE)

    keyval(key=cbind(isin=k, date=as.character(adhocs.relevant)),
           val=delta)
  }
}
```

After an initial character to POSIXct[4] conversion and subsequent order-by-time command, we extract all ad hoc timestamps belonging to the current ISIN from ev. In Line 10f. we discard

4 https://stat.ethz.ch/R-manual/R-devel/library/base/html/DateTimeClasses.html, accessed: 17-June-2016

dates of ad hoc messages that occurred outside the available tick data range.

For the remaining adhoc dates we check the stock price 1s, 5s, 10s, 30s, 60s, 5m, 10m and 1h after publication of the particular message and compute the respective deltas in percent. For this purpose, we convert the available stock data points into a step function (Line 21). Parameter `f=0` disables interpolation between the given x values, such that the functions return value always corresponds to the last price observed before the queried time point.

The function finally returns the computed deltas for every applicable (`ISIN, ad hoc`)-pair.

4.2.3 MapReduce job call

The final call of the MapReduce job is shown Listing 16. The variables `files` and `inputformat` were introduced in chapter 4.1.1, `map.ticks` in Section 4.2.1 and `red.ticks` in Section 4.2.2.

The job's output will be stored into the HDFS as a csv file while the function `mapreduce()` returns the associated filepath. To further view or process the data in RStudio `from.dfs()` loads the output into the R environment.

Listing 16: Running the MapReduce job.

```
library(rmr2)
rmr.options(backend="hadoop")

data<- mapreduce(input=files
                 ,input.format=inputformat
                 ,output="user/isresearch/output.csv"
                 ,output.format=make.output.format("csv", sep=";")
                 ,map = map.ticks
                 ,reduce = red.ticks
)
data.df = from.dfs(data, format=make.input.format("csv", sep=";"))
```

4.3 Evaluation

Once the MapReduce job has finished successfully and the data is loaded into the R environment, we can finally generate some statistics. The examples task was to find out how fast automated traders respond to published ad hoc messages. The evaluation code is available at [7].

Figure 6 shows the ratio of samples where an ad hoc message was shortly after followed by a price fluctuation. In 58 of 8565 cases (0.7%) a trade had happened within the first second after the publication, whereas after 1 minute the activity rate was already at 8.6%. One hour after publication, 29.9% off all examples where affected by a price fluctuation. The Figure also shows that the majority of these price chances had an upward trend.

Figure 7 shows the average stock performance after an ad hoc message was published. The displayed mean values were calculated from all samples that did show trade activity, i. e. the values for '+1s' origin from 58 samples, whereas the values for '+1h' are an aggregation of 2559

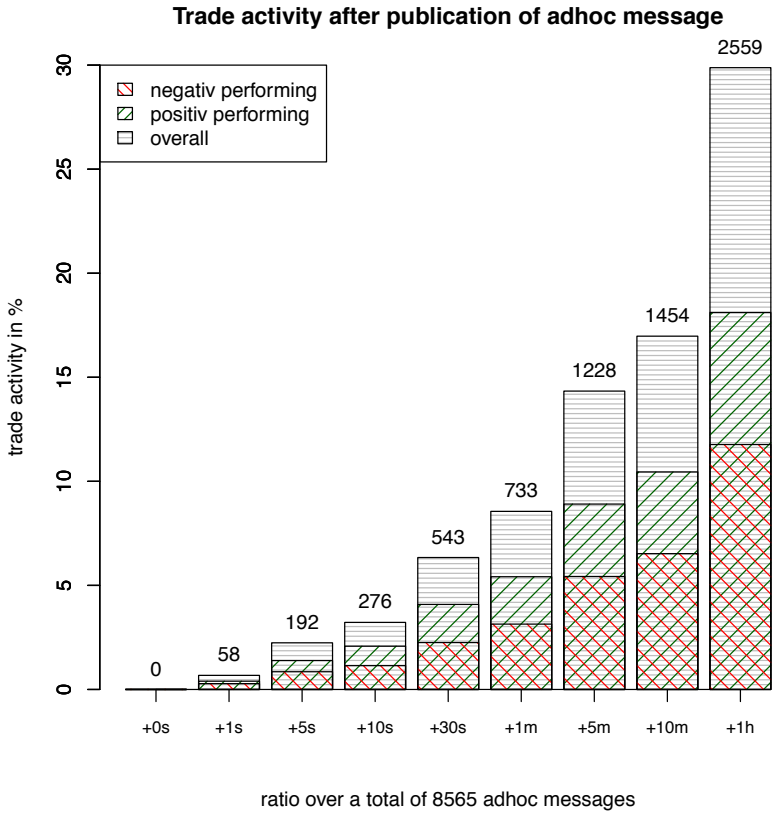

Figure 6: Trade activity after publication of ad hoc message.

values.

Nevertheless, since it seems reasonable that within elongating time intervals more trade activity is observed, we can not proof any causality in between the publication of ad hoc messages and the occurring price fluctuations. In future studies one could analyze preceding points in time as well to compare the trade activity before and after the respective ad hoc message publication.

5 Conclusion

Due to the volume of data, the real world example shown in Section 4 would have been a tough job on any single local workstation or students notebook. When analyzing big data, it is a great relief or even an inevitable thing to use a cluster of computer for distributed storage and data processing.

Hadoop is a great and powerful cluster framework and R is a highly popular and well-advanced programming language for statisticians. In a world of ever growing data, Hadoop and R make a perfect fit. Both combined, the mightful analytic capabilities of R can be applied to big data.

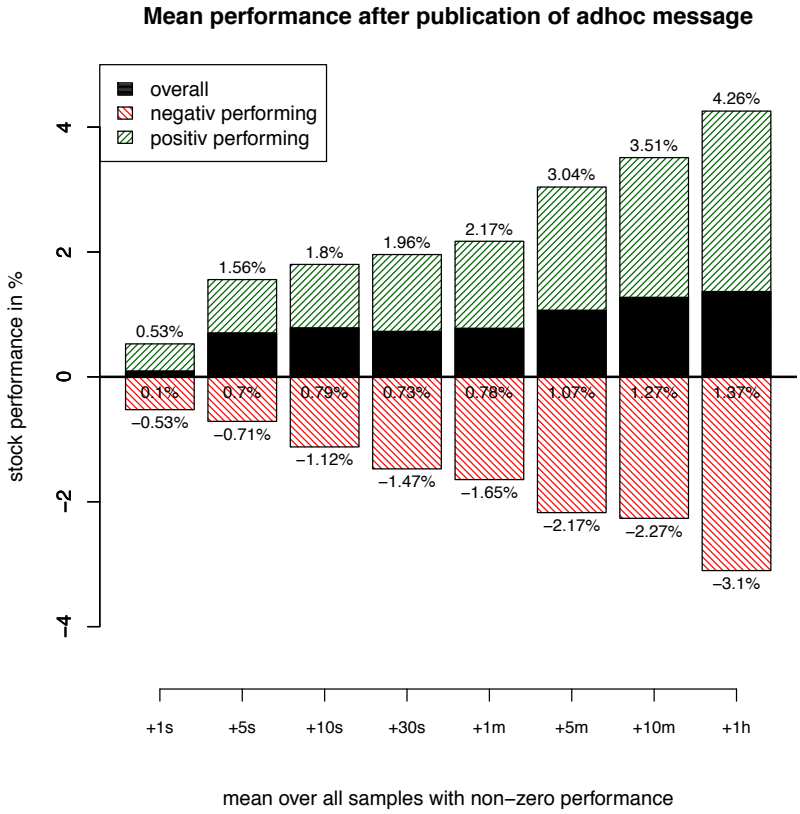

Figure 7: Mean performance after publication of ad hoc message.

A Glossary

CRAN Comprehensive R Archive Network

HDFS Hadoop Distributed File System

GFS Google File System

B References

[1] ANTONIO PICCOLBO. *Using R and Hadoop for Statistical Computation at Scale*. http://strataconf.com/stratany2013/public/schedule/detail/30632. [Online; accessed 11-June-2016].

[2] *Apache Hadoop*. https://hadoop.apache.org.

[3] J. DEAN and S. GHEMAWAT. *MapReduce: Simplified Data Processing on Large Clusters*. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10.

[4] S. GHEMAWAT, H. GOBIOFF, and S.-T. LEUNG. *The Google File System*. In: *SIGOPS Oper. Syst. Rev.* Vol. 37, No. 5 (Oct. 2003), pp. 29–43. ISSN: 0163-5980.

[5] *Glenn K Lockwood*. http://www.glennklockwood.com/data-intensive/hadoop/overview.html. [Online; accessed 08-June-2016].

[6] *Hadoop Streaming*. https://hadoop.apache.org/docs/r1.2.1/streaming.html.

[7] A. PERSCHMANN. *Business Intelligence in R*. https://github.com/axelperschmann/BusinessIntelligenceInR. 2016.

[8] R DEVELOPMENT CORE TEAM. *R: A Language and Environment for Statistical Computing*. Available online at www.R-project.org. R Foundation for Statistical Computing. Vienna, Austria, 2008. ISBN: 3-900051-07-0.

[9] *RevolutionAnalytics - RHadoop Project*. https://github.com/RevolutionAnalytics/RHadoop/wiki. [Online; accessed 09-June-2016].

[10] RSTUDIO TEAM. *RStudio: Integrated Development Environment for R*. RStudio, Inc. Boston, MA, 2015.

[11] THE APACHE SOFTWARE FOUNDATION. *Welcome to HBase!* http://hbase.apache.org. [Online; accessed 15-June-2016]. 2010.

[12] *The Comprehensive R Archive Network*. https://cran.r-project.org/web/packages. [Online; accessed 08-June-2016].

[13] H. WICKHAM. *R Packages*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 1491910593.

[14] XETRA. *Xetra Trading hours*. http://www.xetra.com/xetra-en/trading/trading-information/trading-hours. [Online; accessed 18-June-2016].

C List of Figures

D List of Tables

E Listings