Big Data Statistics for R and Python

Lecture 7: Cloud Computing

Prof. Dr. Ulrich Matter
(University of St. Gallen)

15/04/2019

1 Cloud Services for Big Data Analytics

So far we have focused on how to deal with large amounts of data and/or computationally demanding tasks on our local machine (desktop/laptop). A key aspect of this course has thus been in a first step to understand why our local machine is struggling with a data analysis task when there is a large amount of data to be processed. In a second step we have looked into practical solutions to these challenges. These solutions are in essence tools (in this course particularly tools provided in the R environment) to use the key components of our computing environment (CPU, RAM, mass storage) most efficiently:

- Computationally intense tasks (but not pushing RAM to the limit): parallelization, using several CPU cores (nodes) in parallel.
- Memory-intense tasks (data still fits into RAM): efficient memory allocation (`data.table`-package).
- Memory-intense tasks (data does not fit into RAM): efficient use of virtual memory (use parts of mass storage device as virtual memory).
- Big Data storage: efficient storage (avoid redundancies) and efficient access (speed) with RDBMSs (here: SQLite).

In practice, data sets might be too large for our local machine even if we take all of the techniques listed above into account. That is, a parallelized task might still take ages to complete because our local machine has too few cores available, a task involving virtual memory would use up way too much space on our hard-disk, running a large SQL database locally would use up too much resources, etc.

In such situations, we have to think about horizontal and vertical scaling beyond our local machine. That is, we outsource tasks to a bigger machine (or a cluster of machines) to which our local computer is connected over the Internet (or over a local network). While a decade or two ago most organizations had their own large centrally hosted machines (database servers, cluster computers) for such tasks, today they often rely on third-party solutions *'in the cloud'*. That is, specialized companies provide flexible access to computing resources that can be easily accessed via a broadband Internet-connection and rented on an hourly (or even minutes and seconds) basis. Given the obvious economies of scale in this line of business, a few large players have emerged who practically dominate most of the global market:

- Amazon Web Services (AWS).
- Microsoft Azure
- Google Cloud Platform
- IBM Cloud
- Alibaba Cloud
- Tencent Cloud
- ...

For details on what some of the largest platforms provide, see the overview in the online chapter to Walkowiak (2016) 'Pushing R Further'.

When we use such cloud services to *scale up* (vertical scaling) the computing resources, the transition from our local implementation of a data analytics task to the cloud implementation is often rather simple. Once we have set up a cloud instance and figured out how to communicate with it, we typically can even run the exact same R-script locally or in the cloud. This is typically the case for parallelized tasks (simply run the

same script on a machine with more cores), in-memory tasks (rent a machine with more RAM but still use `data.table()` etc.), and working with an SQL database (simply set up the database in the cloud instead of locally).

However, for really memory-intense tasks, the cloud provides options to *scale out* (horizontal scaling). Meaning, a task is distributed among a cluster of computing instances/servers. The implementation of such data analytics tasks is based on a paradigm that we rather do not encounter when working locally: *Map/Reduce* (implemented in the *Hadoop* framework).

In the following we look first at scaling up more familiar approaches with the help of the cloud and then look at the Map/Reduce concept and how it can be applied in *Hadoop* running on cloud instances.

2 Scaling up in the Cloud

In the following examples we use different cloud services provided by AWS. See the online chapter to Walkowiak (2016) 'Pushing R Further' for how to set up an AWS account and the basics for how to set up AWS instances. The examples below are based on the assumption that the EC2 instance and RStudio Server have been set up exactly as explained in 'Pushing R Further', pages 22-38.

2.1 Parallelization with an EC2 instance

In this short tutorial we scale the computation of clustered standard errors shown in Lecture 3 up by running it on an AWS EC2 instance. Below we use the same source code as in the original example (see `03_computation_memory.Rmd`). Note that there are a few things that we need to keep in mind in order to make the script run on an AWS EC2 instance in RStudio Server.

First, our EC2 instance is a Linux machine. Most of you are probably rather used to running R on a Mac or Windows PC. When running R on a Linux machine, there is an additional step to install R packages (at least for most of the packages): R packages need to be compiled before they can be installed. The command to install packages is exactly the same (`install.packages()`) and normally you only notice a slight difference in the output shown in the R console during installation (and the installation process takes a little longer than what you are used to). However, in some cases the compilation of a package fails (and thus the installation fails) because of some configuration on your Linux machine or because a lower-level program is not installed. Currently, this is what happens when you attempt to install the `data.table` package on your EC2 instance (set up for this example). By default, you won't be able to install `data.table` right away because the EC2 instance uses a new compiler that `data.table` doesn't like. In order to resolve this issue we first have to switch to the terminal window in our current RStudio Server session and run the following two lines of code (source: https://stackoverflow.com/a/48724665):

```
mkdir ~/.R
echo "CC=gcc64" >> ~/.R/Makevars
```

This will point the R package installer to an older compiler which resolves the issue. Obviously, this is just one small example which does not generalize and might not be an issue at all in the near future (with a new `data.table` package version). However, it is a nice example of an additional step needed to make your locally (Mac/Windows) tested R-script work on a Linux machine in the cloud.

Now, let's go through the bootstrap example. First, let's run the non-parallel implementation of the script. When executing the code below line-by-line, you will notice that essentially all parts of the script work exactly as on your local machine. This is one of the great advantages of running R/RStudio Server in the cloud. You can implement your entire data analysis locally (based on a small sample), test it locally, and then move it to the cloud and run it on a larger scale in exactly the same way (even with the same GUI).

```
# CASE STUDY: PARALLEL ---------------------------
```

```r
# NOTE: the default EC2 AMI instance uses a newer compiler which data.table does not like,
# before you can install data.table, switch to the terminal in your current RStudio Server session and
# type the following:
# mkdir ~/.R
# echo "CC=gcc64" >> ~/.R/Makevars
# this sets the default to an older C compiler.
# See https://stackoverflow.com/questions/48576682/r-and-data-table-on-aws for details.

# install packages
install.packages("data.table")
install.packages("doSNOW")

# load packages
library(data.table)


## ------------------------------------------------------------------------
stopdata <- read.csv("https://vincentarelbundock.github.io/Rdatasets/csv/carData/MplsStops.csv")

## ------------------------------------------------------------------------
# remove incomplete obs
stopdata <- na.omit(stopdata)
# code dependent var
stopdata$vsearch <- 0
stopdata$vsearch[stopdata$vehicleSearch=="YES"] <- 1
# code explanatory var
stopdata$white <- 0
stopdata$white[stopdata$race=="White"] <- 1

## ------------------------------------------------------------------------
model <- vsearch ~ white + factor(policePrecinct)

## ------------------------------------------------------------------------
fit <- lm(model, stopdata)
summary(fit)


# bootstrapping: normal approach

## ----message=FALSE-------------------------------------------------------

# set the 'seed' for random numbers (makes the example reproducible)
set.seed(2)

# set number of bootstrap iterations
B <- 50
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)
# draw bootstrap samples, estimate model for each sample
for (i in 1:B) {
```

```
  # draw sample of precincts (cluster level)
  precincts_i <- sample(precincts, size = 5, replace = TRUE)
  # get observations
  bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
  bs_i <- rbindlist(bs_i)

  # estimate model and record coefficients
  boot_coefs[i,] <- coef(lm(model, bs_i))[1:2] # ignore FE-coefficients
}

## ------------------------------------------------------------------------
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

So far, we have only demonstrated that the simple implementation (non-parallel) works both locally and in the cloud. The real purpose of using an EC2 instance in this example is to make use of the fact that we can scale up our instance to have more CPU cores available for the parallel implementation of our bootstrap procedure. Recall that running the script below on our local machine will employ all cores available to an compute the bootstrap resampling in parallel on all these cores. Exactly the same thing happens when running the code below on our simple `t2.micro` instance. However this type of EC2 instance only has one core. You can check this when running the following line of code in RStudio Server (assuming the `doSNOW` package is installed and loaded):

```
parallel::detectCores()
```

When running the entire parallel implementation below, you will thus notice that it won't compute the bootstrap SE any faster than with the non-parallel version above. However, by simply initiating another EC2 type with more cores, we can distribute the workload across many CPU cores, using exactly the same R-script.

```
# bootstrapping: parallel approaach

## ----message=FALSE-------------------------------------------------------
# install.packages("doSNOW", "parallel")
# load packages for parallel processing
library(doSNOW)

# get the number of cores available
ncores <- parallel::detectCores()
# set cores for parallel processing
ctemp <- makeCluster(ncores) #
registerDoSNOW(ctemp)


# set number of bootstrap iterations
B <- 50
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)

# bootstrapping in parallel
boot_coefs <-
```

```
  foreach(i = 1:B, .combine = rbind, .packages="data.table") %dopar% {

    # draw sample of precincts (cluster level)
    precincts_i <- sample(precincts, size = 5, replace = TRUE)
    # get observations
    bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
    bs_i <- rbindlist(bs_i)

    # estimate model and record coefficients
    coef(lm(model, bs_i))[1:2] # ignore FE-coefficients

  }


# be a good citizen and stop the snow clusters
stopCluster(cl = ctemp)



## ------------------------------------------------------------------------
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

2.2 Mass Storage: MariaDB on a EC2 instance

Once we have set up RStudio Server on an EC2 instance, we can run the SQLite examples demonstrated locally in Lecture 6 on it. There are no additional steps needed to install SQLite. However, when using RDBMSs in the cloud, we typically have a more sophisticated implementation than SQLite in mind. Particularly, we want to set up an actual RDBMS-server running in the cloud to which several clients can connect (via RStudio Server). The following example, based on Walkowiak (2016), guides you through the first step to set up such a database in the cloud. To keep things simple, the example sets up a database of the same data set as shown in the first SQLite example in Lecture 6, but this time with MariaDB on an EC2 instance in the cloud. For most of the installation steps you are referred to the respective pages in Walkowiak (2016) (Chapter 5: 'MariaDB with R on a Amazon EC2 instance, pages 255ff). However, since some of the steps shown in the book are outdated, the example below hints to some alternative/additional steps needed to make the database run on an Ubuntu 18.04 machine.

2.2.1 Data import

We import the same simple data set `economics.csv` used in the local SQLite examples of Lecture 6. Following the Instructions of Walkowiak (2016), pages 252 to 254, we upload the `economics.csv` file (instead of the example data used in Walkowiak (2016)).[1]

```
# from the directory where the key-file is stored...
scp -r -i "mariadb_ec2.pem" ~/Desktop/economics.csv umatter@ec2-184-72-202-166.compute-1.amazonaws.com:~
```

After installing MariaDB and starting a new database called `data1` (following Walkowiak (2016), pages 255-259), we now create the first table of our database and import data to it. Note that we only have to

[1] Note that in all the code examples below, the username is `umatter`, and the IP-address will have to be replaced with the public IP-address of your EC2 instance.

slightly adjust the former SQLite syntax to make this work (remove double quotes for field names). In addition, note that we can use the same field types as in the SQLite DB.[2]

```
-- Create the new table
CREATE TABLE econ(
date DATE,
pce REAL,
pop INTEGER,
psavert REAL,
uempmed REAL,
unemploy INTEGER
);
```

After following the steps in Walkowiak (2016), pages 259-262, we can import the `economics.csv`-file to the `econ` table in MariaDB (again, assuming the username is `umatter`). Note that the syntax to import data to a table is quite different from the SQLite example in Lecture 6.

```
LOAD DATA LOCAL INFILE
'/home/umatter/economics.csv'
INTO TABLE econ
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Now we can start using the newly created database from within RStudio Server running on our EC2 instance (following Walkowiak (2016), pages 263ff). Note that with the current version of Ubuntu Linux running on EC2, you need to install a newer version of the MariaDB client. The following solution suggested here should work.

```
sudo apt-get install software-properties-common
sudo apt-key adv --recv-keys --keyserver hkp://keyserver.ubuntu.com:80 0xF1656F24C74CD1D8
sudo add-apt-repository 'deb [arch=amd64,arm64,ppc64el] http://sfo1.mirrors.digitalocean.com/mariadb/re
sudo apt update
sudo apt install mariadb-server
sudo apt install mariadb-client
sudo apt install libmariadb-dev
sudo apt install libmariadb-dev-compat
sudo apt-get install libmariadbclient18
```

As in the SQLite examples in Lecture 6, we can now query the database from within the R console (this time using `RMySQL` instead of `RSQLite`, and using R from within RStudio Server in the cloud!).

First, we need to connect to the newly created MariaDB database.

```
# load packages
library(RMySQL)

# connect to the db
con <- dbConnect(RMySQL::MySQL(),
                 user = "umatter",
                 password = "Password1",
                 host = "localhost",
                 dbname = "data1")
```

In our first query, we select all (`*`) variable values of the observation of January 1968.

[2]However, MariaDB is a much more sophisticated RDBMS than SQLite and comes with many more field types, see the official list of supported data types.

```
# define the query
query1 <-
"
SELECT * FROM econ
WHERE date = '1968-01-01';
"
# send the query to the db and get the result
jan <- dbGetQuery(con, query1)
jan
```

```
#        date   pce    pop psavert uempmed unemploy
# 1 1968-01-01 531.5 199808    11.7     5.1     2878
```

Now let's select all year/months in which there were more than 15 million unemployed, ordered by date.

```
query2 <-
"
SELECT date FROM econ
WHERE unemploy > 15000
ORDER BY date;
"

# send the query to the db and get the result
unemp <- dbGetQuery(con, query2)
head(unemp)
```

```
#         date
# 1 2009-09-01
# 2 2009-10-01
# 3 2009-11-01
# 4 2009-12-01
# 5 2010-01-01
# 6 2010-02-01
```

When done working with the database, we close the connection to the MariaDB database with `dbDisconnect(con)`.

3 Distributed Systems/Map Reduce

3.1 Map/Reduce Concept: Illustration in R

In order to better understand the basic concept behind the MapReduce-Framework on a distributed system, let's look at how we can combine the basic functions `map()` and `reduce()` in R to implement the basic MapReduce example shown in Walkowiak (2016), Chapter 4, pages 132-134 (this is just to illustrate the underlying idea, not to suggest that MapReduce actually is just an application of the classical `map` and `reduce (fold)` functions in functional programming).[3] The overall aim of the program is to count the number of times each word is repeated in a given text. The input to the program is thus a text, the output is a list of key-value pairs with the unique words occurring in the text as keys and their respective number of occurrences as values.

In the code example, we will use the following text as input.

[3] For a more detailed discussion of what `map` and `reduce` have *actually* to do with MapReduce see this post.

```
input_text <-
"Simon is a friend of Becky.
Becky is a friend of Ann.
Ann is not a friend of Simon."
```

3.1.1 Mapper

The Mapper first splits the text into lines, and then splits the lines into key-value pairs, assigning to each key the value `1`. For the first step we use `strsplit()` that takes a character string as input and splits it into a list of substrings according to the matches of a substring (here `"\n"`, indicating the end of a line).

```
# Mapper splits input into lines
lines <- as.list(strsplit(input_text, "\n")[[1]])
lines
```

```
## [[1]]
## [1] "Simon is a friend of Becky."
##
## [[2]]
## [1] "Becky is a friend of Ann."
##
## [[3]]
## [1] "Ann is not a friend of Simon."
```

In a second step, we apply our own function (`map_fun()`) to each line of text via `Map()`. `map_fun()` splits each line into words (keys) and assigns a value of `1` to each key.

```
# Mapper splits lines into Key-Value pairs
map_fun <-
    function(x){

        # remove special characters
        x_clean <- gsub("[[:punct:]]", "", x)
        # split line into words
        keys <- unlist(strsplit(x_clean, " "))
        # initiate key-value pairs
        key_values <- rep(1, length(keys))
        names(key_values) <- keys

        return(key_values)
    }

kv_pairs <- Map(map_fun, lines)

# look at the result
kv_pairs
```

```
## [[1]]
##  Simon     is      a friend     of  Becky
##      1      1      1      1      1      1
##
## [[2]]
##  Becky     is      a friend     of    Ann
##      1      1      1      1      1      1
##
```

```
## [[3]]
##    Ann     is    not      a friend     of  Simon
##      1      1      1      1      1      1      1
```

3.1.2 Reducer

The Reducer first sorts and shuffles the input from the Mapper and then reduces the key-value pairs by summing up the values for each key.

```
# order and shuffle
kv_pairs <- unlist(kv_pairs)
keys <- unique(names(kv_pairs))
keys <- keys[order(keys)]
shuffled <- lapply(keys,
                   function(x) kv_pairs[x == names(kv_pairs)])
shuffled
```

```
## [[1]]
## a a a
## 1 1 1
##
## [[2]]
## Ann Ann
##   1   1
##
## [[3]]
## Becky Becky
##     1     1
##
## [[4]]
## friend friend friend
##      1      1      1
##
## [[5]]
## is is is
##  1  1  1
##
## [[6]]
## not
##   1
##
## [[7]]
## of of of
##  1  1  1
##
## [[8]]
## Simon Simon
##     1     1
```

Now we can sum up the keys in order to the the word count for the entire input.

```
sums <- sapply(shuffled, sum)
names(sums) <- keys
sums
```

```
##      a    Ann  Becky friend     is    not     of  Simon
```

```
##      3      2      2      3      3      1      3      2
```

3.1.3 Simpler example: Compute the total number of words

```
# assigns the number of words per line as value
map_fun2 <-
     function(x){
          # remove special characters
          x_clean <- gsub("[[:punct:]]", "", x)
          # split line into words, count no. of words per line
          values <- length(unlist(strsplit(x_clean, " ")))
          return(values)
     }
# Mapper
mapped <- Map(map_fun2, lines)
mapped
```

```
## [[1]]
## [1] 6
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 7
```

```
# Reducer
reduced <- Reduce(sum, mapped)
reduced
```

```
## [1] 19
```

3.2 Notes on Deploying Hortonworks Sandbox on Azure

The instructions in Walkowiak (2016) Chapter 4, pages 138ff are partly outdated. You will notice that the screenshots do not anymore correspond with the current Azure layout. The Hortonworks tutorial 'Deploying Hortonworks Sandbox on Microsoft Azure' is more up to date.[4] The layout of Microsoft Azure's dashboard has slightly changed. The settings are essentially the same but might occur an a different order than what you see in the Hortonworks tutorial. There are also small changes in the default settings. You might have to add an additional inbound rule in the 'Networking' settings (either when setting up the virtual machine or after starting it) that allows ssh (port 22) and HTTP (port 80) inbound traffic.

Once you've gone through all the steps in the 'Deploying Hortonworks Sandbox on Microsoft Azure' tutorial, open a terminal window and connect to the virtual machine as suggested in the tutorial:

```
ssh azureSandbox
```

Open another terminal window and run the following command to connect with the hadoop sandbox. You will be asked for the password. The default password is `hadoop`. After entering the password you will be prompted to change the password (`Changing password for root. (current) UNIX password:`). In order to change it you have to re-enter the current password (`hadoop`) and then enter your new password for root.

```
ssh root@localhost -p 2222
```

[4] For a short description of how to set up Hortonworks Sandbox on AWS, see here.

3.3 Notes on working with Hortonworks Sandbox/Hadoop on Azure

(Walkowiak (2016): A word count example in Hadoop using Java)

Now everything should be set up and running and you can continue with the tutorial in Walkowiak (2016), pages 152 (unlike in the example in the book, the prompt will read `[root@sandbox-hdp ~]`) with the line `hadoop version`. If you do not want to go through the tutorial as `root`, you can add a user as follows (replacing `<username>` with a user name of your choice).

```
useradd <username>
passwd <username>
```

Note that you will have to grant the user additional rights for some of the commands in the book's tutorial. Follow these instructions to do so.

In the following I use the user name `umatter` (with `sudo` permission). In order to connect to the sandbox with the new user, open a new terminal window and type the following command.

```
ssh umatter@localhost -p 2222
```

To transfer the data set as instructed on page 160 in Walkowiak (2016), use the following command in a new Terminal

```
scp -P 2222 -r ~/Desktop/twain_data.txt umatter@localhost:~/data
```

Similarly, for uploading the Java classes as instructed on page 162:

```
 scp -P 2222 -r ~/Desktop/wordcount umatter@localhost:~/wordcount
```

And finally to download the final output of the word count exercise:

```
scp -P 2222 -r umatter@localhost:~/wordcount/wordcount.txt ~/Desktop/wordcount_final.txt
```

References

Walkowiak, Simkon. 2016. *Big Data Analytics with R*. Birmingham, UK: PACKT Publishing.