Big Data Analytics

Lecture 8:

Cloud Computing: Introduction/Overview, Distributed Systems

Prof. Dr. Ulrich Matter
29/04/2021

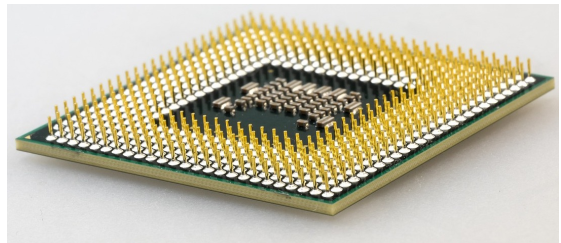
Updates

Schedule

1. Introduction: Big Data, Data Economy. Walkowiak (2016): Chapter 1.
2. Computation and Memory in Applied Econometrics.
3. **Computation and Memory in Applied Econometrics II.**
4. **Advanced R Programming. Wickham (2019): Chapters 2, 3, 17,23, 24.**
5. Import, Cleaning and Transformation of Big Data. Walkowiak (2016): Chapter 3: p. 74-118.
6. Aggregation and Visualization. Walkowiak (2016): Chapter 3: p. 118-127; Wickham et al.(2015); Schwabish (2014).
7. Data Storage, Databases Interaction with R. Walkowiak (2016): Chapter 5.
8. **Cloud Computing: Introduction/Overview, Distributed Systems, Walkowiak (2016): Chapter 4.**
9. Applied Econometrics with Spark; Machine Learning and GPUs.
10. **Q&A in Zoom**
11. **Q&A in Zoom, (hand in presentations)**

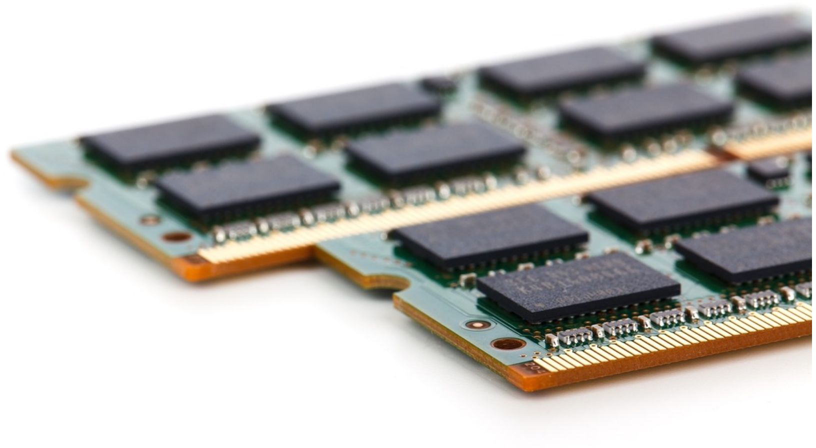
Cloud Services for Big Data Analytics

Wrap-up: efficient use of CPU, RAM, Mass Storage



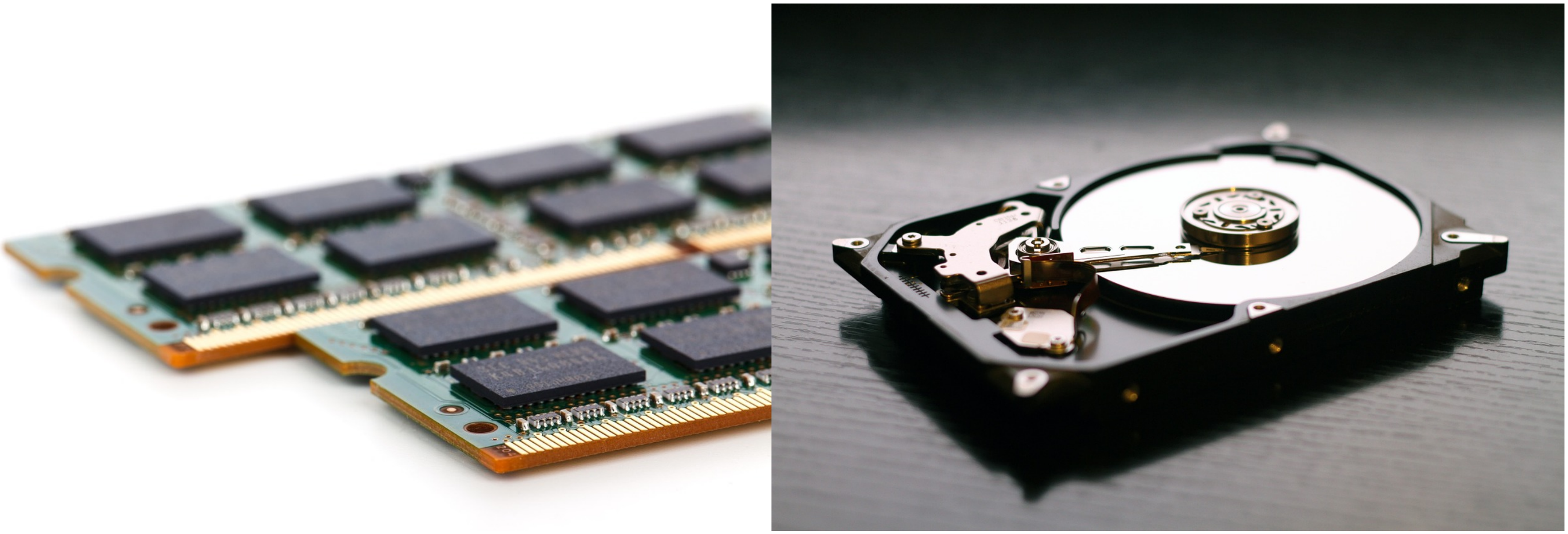
Computationally intense tasks: parallelization, using several CPU cores (nodes) in parallel.

Wrap-up: efficient use of CPU, RAM, Mass Storage



Memory-intense tasks (data still fits into RAM): efficient memory allocation (`data.table`-package).

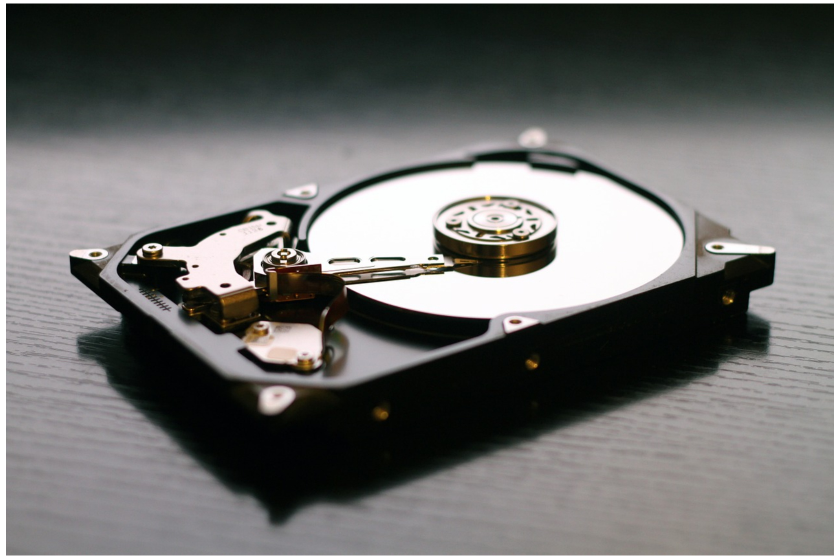
Wrap-up: efficient use of CPU, RAM, Mass Storage



RAM and Harddrive

Memory-intense tasks (data does not fit into RAM): efficient use of virtual memory (use parts of mass storage device as virtual memory).

Wrap-up: efficient use of CPU, RAM, Mass Storage

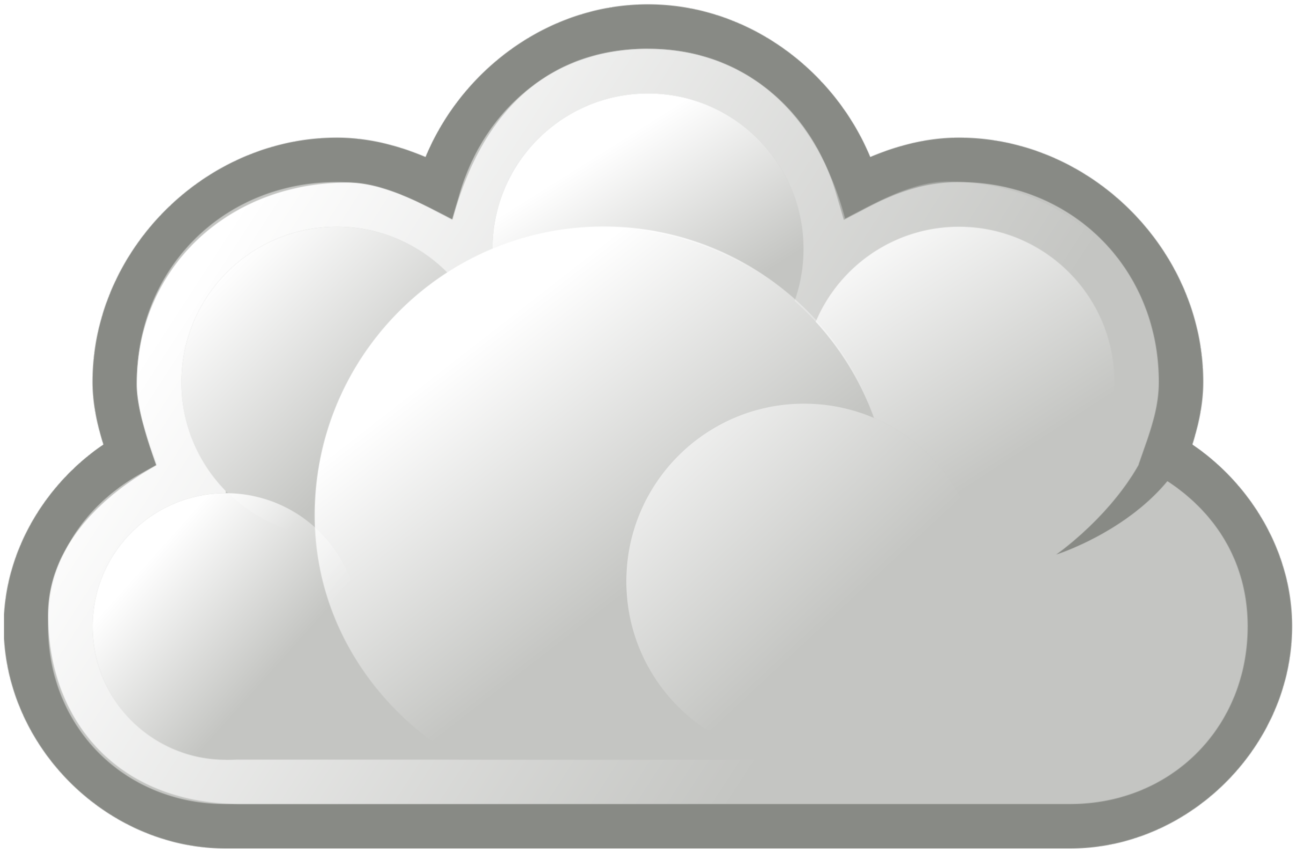


(Big) Data storage: efficient storage (avoid redundancies) and efficient access (speed) with RDBMSs (here: SQLite).

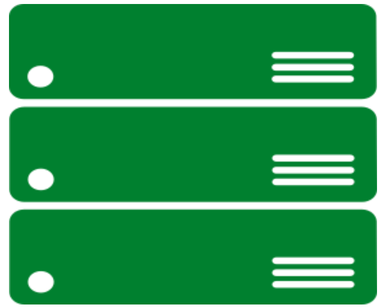
Already using all components most efficiently?

- Scale up ('vertical scaling')
- Scale out ('horizontal scaling')

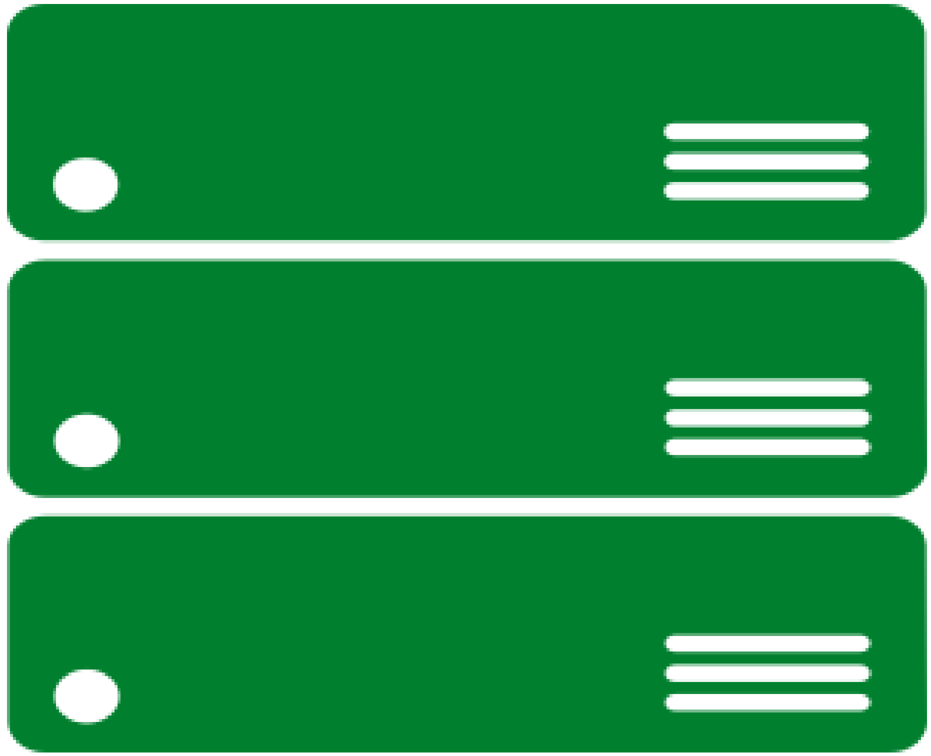
‘The Cloud’



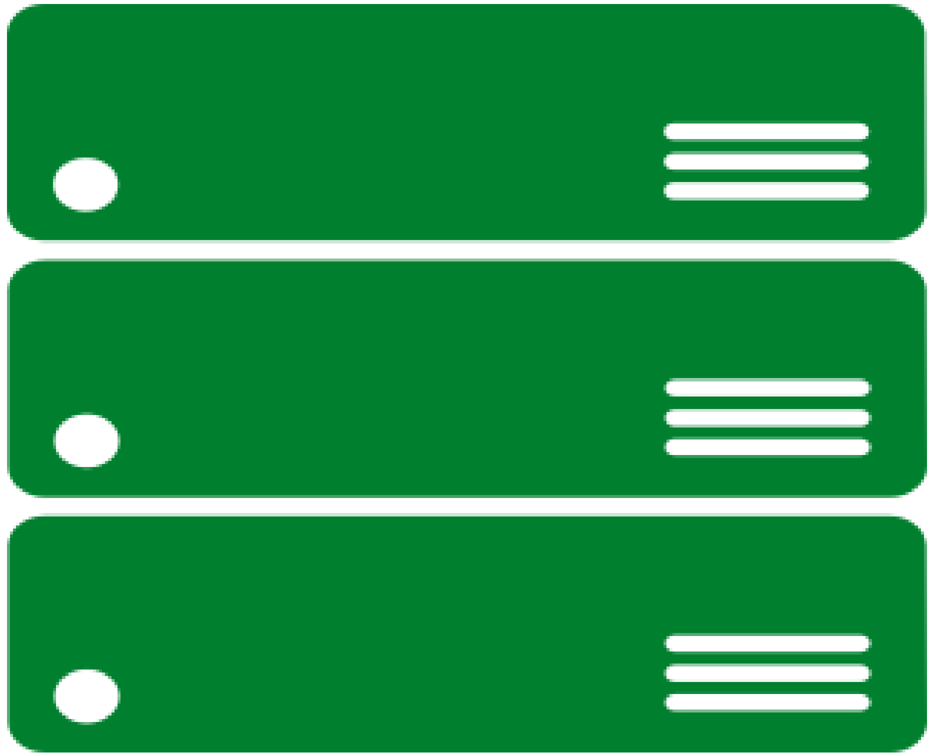
The Cloud: Scaling Up



The Cloud: Scaling Up

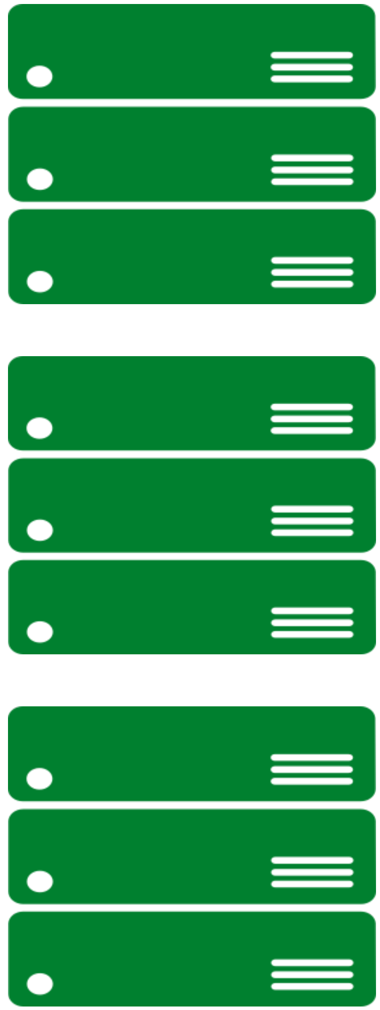


The Cloud: Scaling Up



- Parallel computing, large in-memory computation, SQL/NoSQL databases, etc.
- Common in scientific computing.

The Cloud: Scaling Out



- Distributed Systems: MapReduce/Hadoop etc.
- Rather rare in an applied econometrics setting.

The Cloud in Practice

Rent (virtual) machines on a flexible basis (hourly rate, etc.) from a cloud computing provider.

- Amazon Web Services (AWS)
- Microsoft Azure
- Google Cloud Platform
- IBM Cloud
- Alibaba Cloud（阿里云)
- Tencent Cloud (腾讯云)
- …

Scaling up in the Cloud

Set up

- 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.

Parallelization with an EC2 instance

Run non-parallelized implementation in the cloud.

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


# 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

## ------------------------------------------------------------------------
```

Parallelization with an EC2 instance

Scaling up: rent a machine with more CPU cores.

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

- EC2 instances of type `t2.micro` (free tier) only have one core.
- However, there are many options to scale this up (rent a machine with more CPU cores).

Parallelization with an EC2 instance

Run the parallelized implementation on an EC2 instance.

```
# 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% {
```

Mass Storage: SQL on an EC2 instance

- SQLite: already there!
- However, for the cloud a more sophisticated (client/server) SQL version makes more sense.

Mass Storage: MariaDB on an EC2 instance

- For most of the installation steps, see Walkowiak (2016) (Chapter 5: 'MariaDB with R on a Amazon EC2 instance, pages 255ff).
- `economics.csv` used in the local SQLite examples of Lecture 7.

Data upload (server-side)

```
# 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.c
```

Data import (server-side)

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

Data import (server-side)

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

Connect to MariaDB from RStudio Server (client-side)

```r
# load packages
library(RMySQL)

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

Query the database (client-side)

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

```
# 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
```

Query the database (client-side)

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

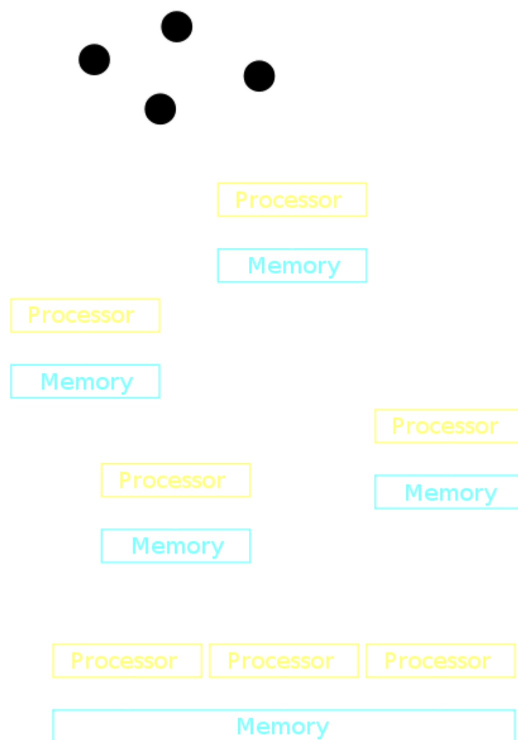
```
query2 <-
"
SELECT date FROM econ
WHERE unemploy > 15000
ORDER BY date;
"

# 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
```

Distributed Systems/MapReduce

Distributed systems



(a), (b): a distributed system. (c): a parallel system.I llustration by [Miym](). [CC BY-SA 3.0]()

MapReduce: Word Count Example

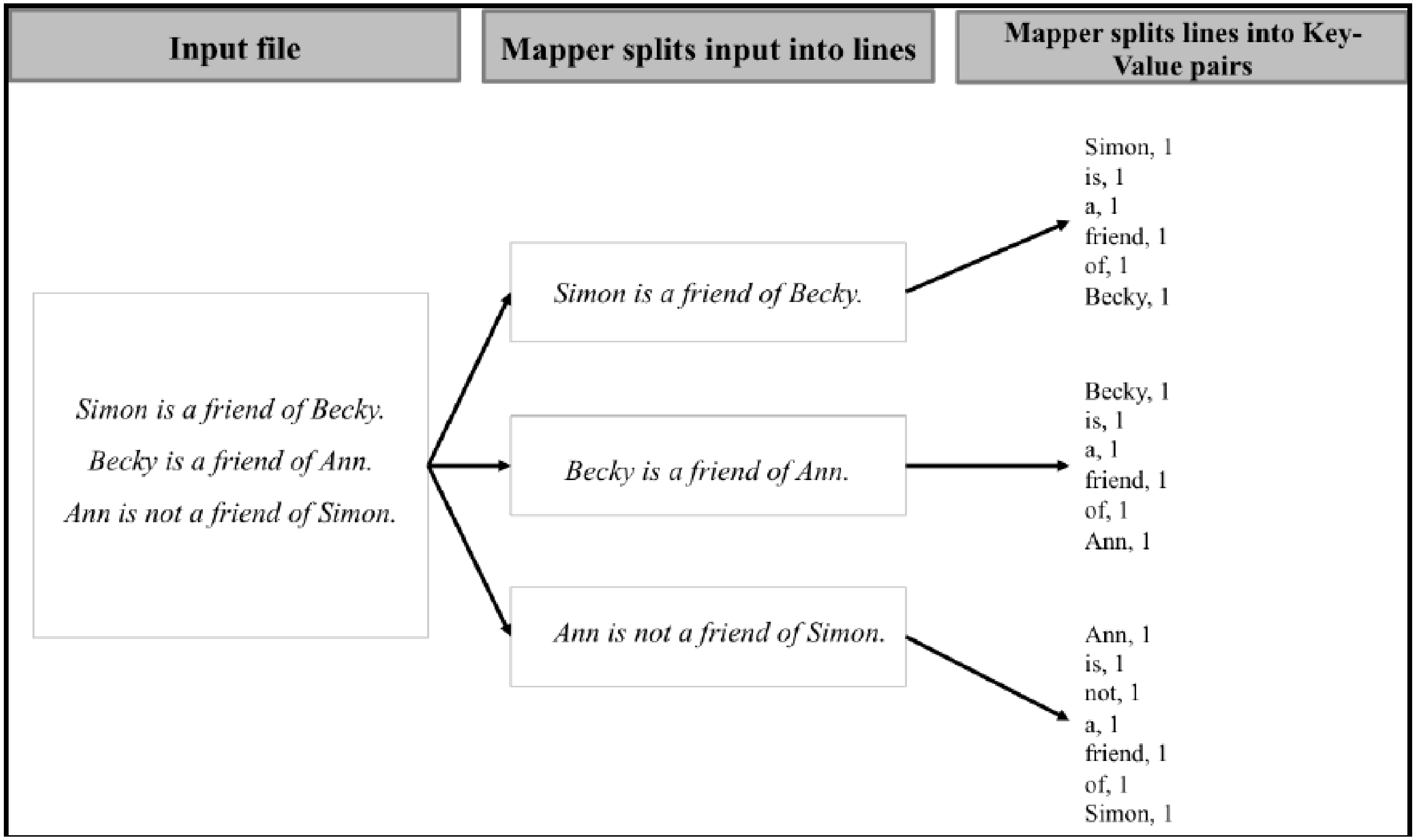
From Walkowiak (2016), Chapter 4:

Simon is a friend of Becky.

Becky is a friend of Ann.

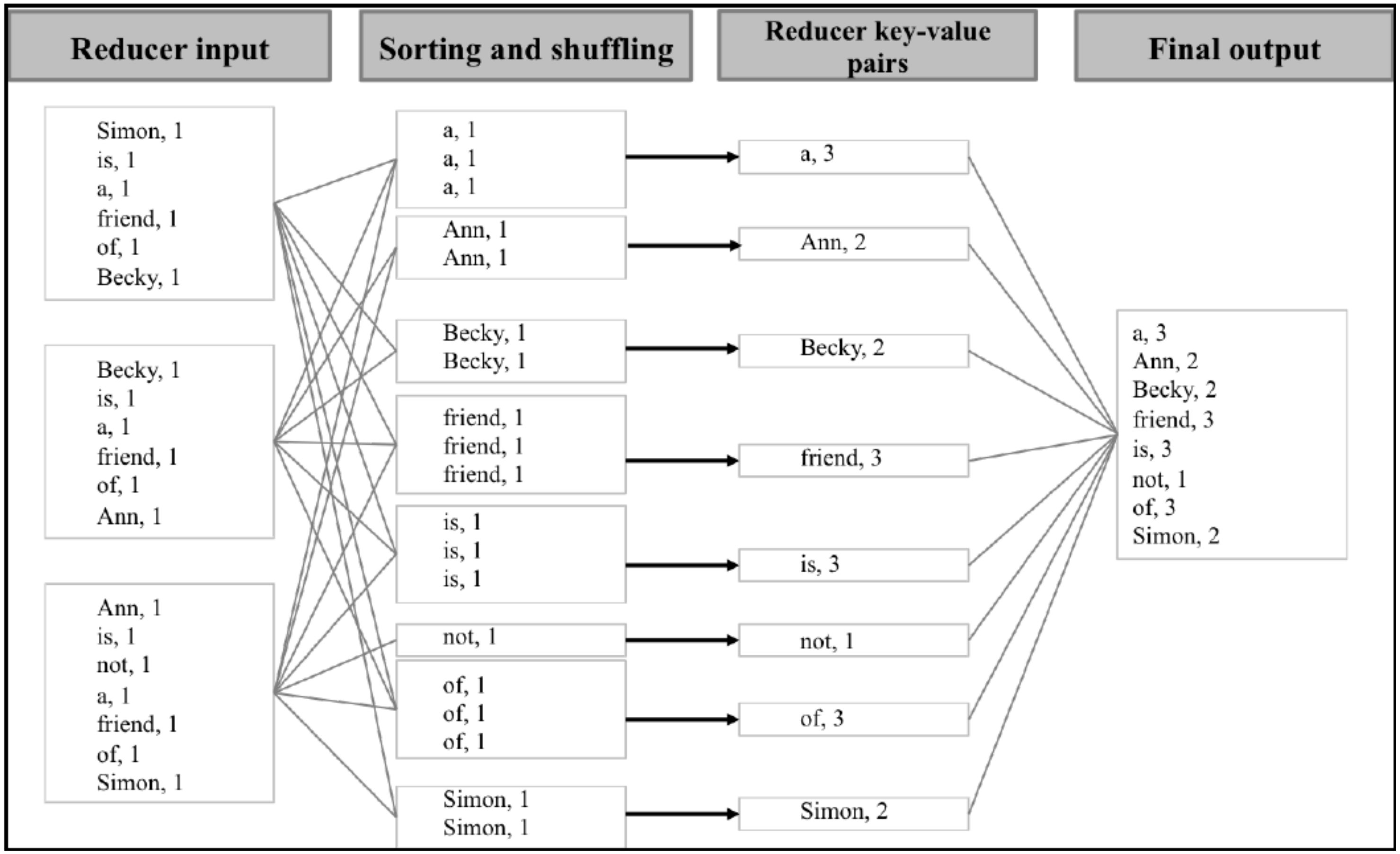
Ann is not a friend of Simon.

MapReduce: Word Count Example



Source: Walkowiak (2016), Chapter 4

MapReduce: Word Count Example



Source: Walkowiak (2016), Chapter 4

Map/Reduce Concept: illustration in R

- `map()`
- `reduce()`

Note: this code example serves to illustrate the underlying idea of MapReduce and how it is related to the idea of `map` and `reduce` functions. It does **not** suggest that MapReduce actually is simply an application of the classical `map` and `reduce (fold)` functions.

Map/Reduce Concept: illustration in R

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

Mapper

```
# 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."
```

Mapper

```r
# 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
```

Reducer

```
# 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
```

Reducer

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
```

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
```

Map/Reduce with Hadoop

APACHE

hadoop™

Hadoop locally (on Ubuntu/Pop_os Linux)

The following example and installation instructions are in part adapted from [this tutorial] by Melissa Anderson and Hanif Jetha.

```
# download binary
wget https://downloads.apache.org/hadoop/common/hadoop-2.10.0/hadoop-2.10.0.tar.gz
# download checksum
wget https://www.apache.org/dist/hadoop/common/hadoop-2.10.0/hadoop-2.10.0.tar.gz.sha512

# run the verification
shasum -a 512 hadoop-2.10.0.tar.gz
# compare with value in mds file
cat hadoop-2.10.0.tar.gz.sha512

# if all is fine, unpack
tar -xzvf hadoop-2.10.0.tar.gz
# move to proper place
sudo mv hadoop-2.10.0 /usr/local/hadoop

# point hadoop to the right java version
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")

# clean up after installation/configuration
rm hadoop-2.10.0.tar.gz
rm hadoop-2.10.0.tar.gz.sha512
```

Check the installation

```
# check installation
/usr/local/hadoop/bin/hadoop version
```

Run Hadoop example

- Basic Hadoop installation comes with a few examples for very typical map/reduce programs.
- More sophisticated programs need to be custom made, written in Java.

Below we replicate the same word-count example as shown in simple R code above.

Run Hadoop example

In a first step, we create an input directory where we store the input file(s) to feed to Hadoop.

```
# create directory for input files (typically text files)
mkdir ~/input
```

Run Hadoop example

Then we add a textfile containing the same text as in the example as above (to make things simpler, we already remove special characters).

```
echo "Simon is a friend of Becky
Becky is a friend of Ann
Ann is not a friend of Simon" >>  ~/input/text.txt
```

Run Hadoop example

Now we can run the MapReduce/Hadoop word count as follows, storing the results in a new directory called `wordcount_example`.

```
# run mapreduce word count
# the specific hadoop program with the implemented wordcount is stored in a java-archive applicatior
/usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-
```

Run Hadoop example

Inspect the results

```
cat ~/wc_example/*
```

```
## Ann  2
## Becky    2
## Simon    2
## a    3
## friend   3
## is   3
## not  1
## of   3
```

MapReduce/Hadoop summary

- Motivation: need to scale out storage/memory.
- Hadoop:MapReduce principle implemented to run on distributed systems.
- Can be run locally (on one node): develop/test code.
- Allows for massive horizontal scaling.
- RAM/storage distributed across machines.

References

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