

Homework 3: Spark, RDDs, DF

E-63 Big Data Analytics
Harvard University, Autumn 2017

Tim Hagmann

September 23, 2017

Contents

| | |
|---|-----------|
| Introduction | 1 |
| Problem 1 (20%) | 2 |
| Create AWS instance | 2 |
| SCP file transfer | 5 |
| Java, Python, Scala and R | 5 |
| Problem 2 (15%) | 7 |
| Install Spark | 7 |
| Spark config | 7 |
| Check Pyspark | 8 |
| Eliminate warnings | 8 |
| Check Spark-Shell | 8 |
| Problem 3 (15%) | 9 |
| i.) Resilient Distributed Dataset (RDD) | 9 |
| ii) Lambda function (RDD) | 9 |
| iii) Named function (RDD) | 10 |
| Problem 4 (15%) | 10 |
| i) Datasets/Dataframe (DF) | 10 |
| ii) Word count (DF) | 10 |
| Problem 5 (15%) | 11 |
| i.) Benchmark (Bash) | 11 |
| ii.) Total word count (RDD) | 11 |
| Problem 6 (15%) | 12 |
| i) Total word count (DF) | 12 |
| ii) Number of unique words (optional) | 13 |
| iii) SparkR (optional) | 14 |
| References | 16 |

Introduction

The following assignment is concerned with the installation and application of Apache Spark. Spark is an open-source cluster-computing framework which provides programmers with an application programming interface (API). It was developed in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results

on disk. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.

Problem 1 (20%)

Create your own Virtual Machine with a Linux operating system. The lecture notes speak about CentOS. You are welcome to work with another Linux OS. When creating the VM, create an administrative user. Call that user whatever you feel like.

Create AWS instance

As stated on Piazza, instead of using a VM it is also possible to create a AWS instance. Spark is a cluster-computing framework. In order to use the cluster capabilities of Spark it is sensible to use AWS instead of a VM.

Note: There are in principal two ways starting up a AWS instance, one is working with the web based *point and click* interface and the other is to use the *command-line interface (CLI)*. We're using the second option here. The reason for this is, that when working with clusters, using a startup script can be significantly faster and is more fault-tolerant than configuring the cluster by hand.

Step 1: Installing AWS CLI

First, we have to install the AWS CLI on the local bash client (e.g., WSL for Windows, terminal on Linux, etc.)

```
sudo apt install awscli -y
```

Step 2: Configure AWS CLI

Second, we have to configure the AWS CLI, in order to do this, a user has to be created under: <https://console.aws.amazon.com/iam/home#/users>. With the information of the user data the AWS CLI can be configured.

```
# configuring AWSCLI
aws configure

# 1. enter your Access Key ID
# 2. enter your Secret Access Key
# 3. choose region close to you [*] (e.g., "eu-central-1")
# 4. enter "json"
```

Step 3: Creating a SSH Key pair

In order to ssh into the AWS instance, a SSH Key has to be created and downloaded:

```
# Creat SSH Key pair
aws ec2 create-key-pair --region eu-central-1 --key-name aws-instance \
  --query "KeyMaterial" --output text > SSH/aws-instance.pem
```

Step 4: Network & Security

Next we're setting up the network and security settings. Out of convenience, we're allowing all IP addresses to access our AWS Server. We're also opening the port 22 (ssh), 80 (http), 443 (https), 8787 (rstudio) and 4040, 4041, 7077 (Spark).

```
# set MYIP to external IP address
MYIP=$(curl -s http://myip.dnsomatic.com | grep -P '[\d.]')

# set ALLIP to 0.0.0.0/0
ALLIP="0.0.0.0/0"

echo $MYIP
echo $ALLIP

# create a new security group and save its returned ID to SGID
SGID=$(aws ec2 create-security-group --group-name aws-sec-group \
    --description "aws security group" --region eu-central-1)

# allow all IP's access to ports
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 22 --cidr $ALLIP --region eu-central-1
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 80 --cidr $ALLIP --region eu-central-1
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 8787 --cidr $ALLIP --region eu-central-1
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 443 --cidr $ALLIP --region eu-central-1
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 4040 --cidr $ALLIP --region eu-central-1
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 4041 --cidr $ALLIP --region eu-central-1
aws ec2 authorize-security-group-ingress --group-name aws-sec-group \
    --protocol tcp --port 7077 --cidr $ALLIP --region eu-central-1
```

Step 5: Launch EC2 Instance

With the above network and security settings we're launching a free-tier Ubuntu 16.04 Ubuntu instance. The only change that is done to the default settings is to increase the root disk space from 8GB to 32GB.

```
# Launch Instance (Ubuntu 16.04)
aws ec2 run-instances \
    --image-id ami-1e339e71 \
    --count 1 \
    --instance-type t2.micro \
    --key-name aws-instance \
    --security-group-ids aws-sec-group \
    --region eu-central-1 \
    --block-device-mapping \
    "[ { \"DeviceName\": \"/dev/sda1\", \"Ebs\": { \"VolumeSize\": 32 } } ]"
```

Step 6: Associate IP Address (optional)

The started instance comes by default with a *flexible* ip address. This means, that when restarting the instance, a new ip address gets associated to the server. In order to avoid this, we can associate a elastic ip

address to the server. First we're reading the id of our instance as well as the from AWS bought elastic ip address (Buying an IP address has to be done separately on AWS).

```
# Get instances id
aws ec2 describe-instances --query \
  'Reservations[].Instances[].[InstanceId, State.Name]' --output text
```

```
## i-0d52c859a139da090  running
```

```
# Get elastic IP addresses
x=$(aws ec2 describe-addresses)
y=$(echo $x | awk '{print $2}')
echo $y
```

```
## eipalloc-a26c62cb
```

With the above instance id (i-0d52c859a139da090) and elastic ip id (eipalloc-a26c62cb) we can associate the IP address.

```
# Associate IP address
aws ec2 associate-address --instance-id i-0d52c859a139da090 \
  --allocation-id eipalloc-a26c62cb
```

Step 7: SSH into EC2 Instance

Having associated instance **ip_address** (xx.xx.xxx.xx) we can now ssh into the server.

```
# SSH into instance
ssh -i SSH/aws-instance.pem ubuntu@<ip_address>
```

Step 8: Add user

We're next adding a new user and are copying the ssh key to that user.

```
# Add new user
sudo adduser tim
sudo adduser tim sudo

# Copy ssh key
sudo mkdir /home/tim/.ssh
sudo cp /home/ubuntu/.ssh/authorized_keys \
  /home/tim/.ssh/authorized_keys
sudo cp /home/ubuntu/.ssh/authorized_keys \
  /home/tim/.ssh/authorized_keys
sudo chown tim -R /home/tim/.ssh
sudo chmod 700 /home/tim/.ssh
sudo chmod 600 /home/tim/.ssh/authorized_keys
```

Step 9: Enable Swapping (optional)

Because of the limited amount of RAM it might be necessary to enable swapping. This isn't necessary the case when using bigger instances.

```
sudo /bin/dd if=/dev/zero of=/var/swap.1 bs=1M count=2048
sudo /sbin/mkswap /var/swap.1
sudo chmod 600 /var/swap.1
sudo /sbin/swapon /var/swap.1
```

Step 10: Install RStudio Server (optional)

In order to facilitate working with the spark instance, the RStudio Server IDE can be installed. The following bash script facilitates the installation.

```
wget https://cdn.rawgit.com/greenore/linux-setup/7d25ec0c/setup_rstudio.sh
chmod +x setup_rstudio.sh
sudo ./setup_rstudio.sh
```

Step 10: Install Java and Scala

Next we have to install java as spark is dependend on a correct java installation. We're also installing the sbt client

```
# Install OpenJDK and scala
sudo apt install default-jre default-jdk scala -y

# Install sbt
echo "deb https://dl.bintray.com/sbt/debian /" \
| sudo tee -a /etc/apt/sources.list.d/sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
--recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
sudo apt-get update
sudo apt install sbt -y
```

SCP file transfer

Once the VM is created, transfer the attached text file Ulysses10.txt to the home of new user. You can do it using scp (secure copy command) or email.

In order to transfer the file trough SCP from the machine to the server we can open a second terminal on the local computer and the following command is transferring the file to the server.

```
scp -i SSH/aws-instance.pem \
/mnt/c/Local/Education/e63-coursework/hw3/data/ulysses10.txt \
tim@<ip_address>:~/.
```

Note: In this project all the files are transfer trough git over a github account. That means, that the above code is not actually executed.

Java, Python, Scala and R

Examine the version of Java, Python and Scala on your VM. If any of those versions is below requirements for Spark 2.2 install proper version. Set JAVA_HOME environmental variable. Set your PATH environmental variable properly, so that you can invoke: java, sbt and python commands from any directory on your system.

Spark version check

Spark runs on Java 8+, Python 2.7+/3.4+ and R 3.1+. For the Scala API, Spark 2.2.0 uses Scala 2.11.

Get Java version

```
java -version
```

```
## openjdk version "1.8.0_131"  
## OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11)  
## OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)
```

Get Python version

```
python -V
```

```
## Python 2.7.12
```

Get R version

```
R --version
```

```
## R version 3.4.1 (2017-06-30) -- "Single Candle"  
## Copyright (C) 2017 The R Foundation for Statistical Computing  
## Platform: x86_64-pc-linux-gnu (64-bit)  
##  
## R is free software and comes with ABSOLUTELY NO WARRANTY.  
## You are welcome to redistribute it under the terms of the  
## GNU General Public License versions 2 or 3.  
## For more information about these matters see  
## http://www.gnu.org/licenses/.
```

Get Scala version

```
scala -version
```

```
## Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
```

Get Sbt version

```
sbt sbtVersion
```

```
## [info] Loading project definition from /home/tim/project  
## [info] Set current project to tim (in build file:/home/tim/  
## [info] 1.0.1
```

The above outputs show, that the necessary software is up to date. Next we're setting the necessary path variables for JAVA_HOME. In order to do this we're creating a `bash_profile` file with the following content:

```
# Get the aliases and functions  
if [ -f ~/.bashrc ]; then  
    . ~/.bashrc  
fi  
  
# JAVA_HOME environment  
JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre/  
export JAVA_HOME  
  
PATH=$PATH:$JAVA_HOME/bin  
export PATH
```

That file has to be sourced in order to be effective.

```
source ~/.bash_profile
```

Problem 2 (15%)

Install Spark 2.2 on your VM. Make sure that pyspark is also installed. Demonstrate that you can successfully open spark-shell and that you can eliminate most of WARNing messages.

Install Spark

There are multiple ways to install spark on the EC2 instance. We're using the default installation process through downloading the tgz file from apache (Another easy way would be to use the spark_install function inside R (sparklyr package)).

```
# Download Spark
sudo wget https://d3kbcqa49mib13.cloudfront.net/spark-2.2.0-bin-hadoop2.7.tgz

# Unpack Spark in the /opt directory
sudo tar xzvf spark-2.2.0-bin-hadoop2.7.tgz -C /opt
```

Next we have to add SPARK_HOME to the bash_profile file. The new file is going to look like this:

```
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# JAVA_HOME environment
JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre/
export JAVA_HOME

SPARK_HOME=/opt/spark-2.2.0-bin-hadoop2.7/
export SPARK_HOME

PATH=$PATH:$JAVA_HOME/bin:$SPARK_HOME/bin
export PATH
```

That file has to be sourced in order to be effective.

```
source ~/.bash_profile
```

Spark config

** Note: ** When launching spark a metastore_db directory and the derby.log file are being created at the start-up location. Working with a version control system this behavior is rather annoying (i.e., version controlling a database doesn't make a lot of sense). However, the default location of the folder can be specified in the spark-defaults.conf file. This is what we're going to do:

```
# Rename spark-defaults
sudo cp /opt/spark-2.2.0-bin-hadoop2.7/conf/spark-defaults.conf.template \
/opt/spark-2.2.0-bin-hadoop2.7/conf/spark-defaults.conf

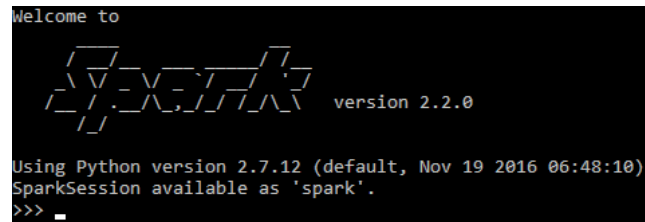
# Append line to the file
```

```
echo 'spark.driver.extraJavaOptions -Dderby.system.home=/tmp/derby' | \
sudo tee --append /opt/spark-2.2.0-bin-hadoop2.7/conf/spark-defaults.conf
```

Check Pyspark

In order to check if pyspark is installed we can run the pyspark command

```
pyspark
```



```
Welcome to
  ____      __
 / _ \_____/ /_  _  _
/_  \_  _/ __ \| \| \
 \___\__\|___/___/|_|_|
version 2.2.0

Using Python version 2.7.12 (default, Nov 19 2016 06:48:10)
SparkSession available as 'spark'.
>>> █
```

The image above shows, that pyspark does work.

Eliminate warnings

Demonstrate that you can successfully open spark-shell and that you can eliminate most of WARNING messages.

In order to eliminate the error messages in the spark-shell we're first creating the log4j.properties file from the template file and setting the log level to ERROR and log4j.rootCategory=ERROR.

```
# Create log4j.properties
sudo mv /opt/spark-2.2.0-bin-hadoop2.7/conf/log4j.properties.template \
/opt/spark-2.2.0-bin-hadoop2.7/conf/log4j.properties

# Edit file
sudo nano /opt/spark-2.2.0-bin-hadoop2.7/conf/log4j.properties
```

Check Spark-Shell

In order to check if the error messages disappeared from spark-shell we can run the following command:

```
spark-shell
```



```
Setting default log level to "ERROR".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://172.31.18.124:4040
Spark context available as 'sc' (master = local[*], app id = local-1505751026849).
Spark session available as 'spark'.
Welcome to
  ____      __
 / _ \_____/ /_  _  _
/_  \_  _/ __ \| \| \
 \___\__\|___/___/|_|_|
version 2.2.0

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```


The image above shows, that spark-shell runs correctly.

Problem 3 (15%)

Find the number of lines in the text file ulysses10.txt that contain word “*afternoon*” or “*night*” or “*morning*”. In this problem use RDD API. Do this in two ways, first create a lambda function which will test whether a line contains any one of those 3 words.

i.) Resilient Distributed Dataset (RDD)

RDD was the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

ii) Lambda function (RDD)

Next we’re using a lambda function to count the number of *night*, *morning* and *afternoon* occurrences in the text.

Install findspark

We’re going to use the standard python 2.7 interpreter to solve the above problem. In order to do this we’re first installing findspark with pip

```
sudo -H pip install findspark
```

Use the Lambda function

```
# Import libraries
import findspark
findspark.init("/opt/spark-2.2.0-bin-hadoop2.7")
from pyspark import SparkContext, SparkConf

# Start session
conf = SparkConf().setMaster("local").setAppName("p3_rdd_lambda")
sc = SparkContext(conf = conf)

# Read data
rdd_ulysses = sc.textFile("/home/tim/e63-coursework/hw3/data/ulysses10.txt")

# Filter values
rdd_linematch = rdd_ulysses.filter(lambda line: "night" in line or "morning" in
                                         line or "afternoon" in line)

# Print data
print "Number of lines with 'morning', 'afternoon', 'night':"
print rdd_linematch.count()
sc.stop()

## Number of lines with 'morning', 'afternoon', 'night':
## 418
```

iii) Named function (RDD)

Second, create a named function in the language of choice that returns TRUE if a line passed to it contains any one of those three words. Demonstrate that the count is the same. Use pyspark and Spark Python API.

```
# Import findspark
import findspark
findspark.init("/opt/spark-2.2.0-bin-hadoop2.7")
from pyspark import SparkConf, SparkContext

# Start session
conf = SparkConf().setMaster("local").setAppName("p3_rdd_function")
sc = SparkContext(conf = conf)

# Define function
def has_word_in(line):
    return ("night" in line or "morning" in line or "afternoon" in line)

# Read data
rdd_ulysses = sc.textFile("/home/tim/e63-coursework/hw3/data/ulysses10.txt")

# Filter values
rdd_linematch = rdd_ulysses.filter(has_word_in)

# Print data
print "Number of lines with 'morning', 'afternoon', 'night':"
print rdd_linematch.count()
sc.stop()
```

```
## Number of lines with 'morning', 'afternoon', 'night':
## 418
```

As can be seen, both *functions* do count 418 morning, afternoon and night words in the ulysses text file.

Problem 4 (15%)

Implement the above task, finding the number of lines with one of those three words in file ulysses10.txt using Dataset/DataFrame API.

i) Datasets/Dataframe (DF)

Starting in Spark 2.0 a DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

ii) Word count (DF)

Using the dataframe API requires to use different syntax. Furthermore, the session is also created in a different fashion.

```

# Import libraries
import findspark
findspark.init("/opt/spark-2.2.0-bin-hadoop2.7")
from pyspark.sql import SparkSession

# Create Session
spark = SparkSession.builder.master("local") \
    .appName("p4_df_filter_count").getOrCreate()

# Read data
df_ulysses = spark.read.text("/home/tim/e63-coursework/hw3/data/ulysses10.txt")

# Filter values
df_linematch = df_ulysses.filter(df_ulysses.value.contains('afternoon') |
                                df_ulysses.value.contains('night') |
                                df_ulysses.value.contains('morning'))

# Print data
print "Number of lines with 'morning', 'afternoon', 'night':"
print df_linematch.count()

## Number of lines with 'morning', 'afternoon', 'night':
## 418

```

We're again counting 418 times morning, afternoon and night. That means we're getting the same count as with the RDD example.

Problem 5 (15%)

Create a standalone Python script that will count all words in file ulysses10.txt. You are expected to produce a single number. Do it using RDD API.

i.) Benchmark (Bash)

Before starting we're defining the bash `wc` function as the benchmark. The bash native wordcount function is a quick and easy way of counting the total number of words inside a text document.

```
wc -w /home/tim/e63-coursework/hw3/data/ulysses10.txt
```

```
## 267832 /home/tim/e63-coursework/hw3/data/ulysses10.txt
```

As can be seen above, the `wc` function counts 267'832. That is the number we're aiming for when counting the words in problem 5 (RDD) and 6 (DF).

ii.) Total word count (RDD)

Counting words inside a RDD can be done with the code below. It follows the classical map/reduce logic. This is the same content as can be found in the `rdd_total_word_count.py` script.

```

# Import findspark
import findspark
findspark.init("/opt/spark-2.2.0-bin-hadoop2.7")
from pyspark import SparkConf, SparkContext

```

```

# Load Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("p5_rdd_count")
sc = SparkContext(conf = conf)

# Load data
rdd_ulysses = sc.textFile("/home/tim/e63-coursework/hw3/data/ulysses10.txt")

# Map and reduce data
rdd_word_count = rdd_ulysses.flatMap(lambda x: x.split()) \
                             .map(lambda x: (x, 1)) \
                             .reduceByKey(lambda x, y : x + y) \
                             .map(lambda x : x[1])

# Print data
print "Total word count:"
print rdd_word_count.sum()
sc.stop()

```

With the below bash command we can run the above python script:

```

# Execute python script
/opt/spark-2.2.0-bin-hadoop2.7/bin/spark-submit \
  /home/tim/e63-coursework/hw3/scripts/p5_rdd_word_count.py

```

```

## Total word count:
## 267832

```

It shows that we count 267'832 words. This matches the results from the bash wc example.

Problem 6 (15%)

Create a standalone Python script that will count all words in file ulysses10.txt. You are expected to produce a single number. Do it using Dataset/DataFrame API.

i) Total word count (DF)

In order to solve the same problem it is possible to apply the *rdd* function on a dataframe and use the same code building blocks as before. However, using the dataframe gives new ways of manipulating the data. Therefore, in order to leverage that we're rewriting the code into the dataframe concept.

Note: Through the `collect()` function the data can be loaded into local memory. This would be another easy solution to solve the problem. However, if the data is too big to fit into the memory of one machine, this is no longer feasible. That is why this option is not done.

```

# Import libraries
import findspark
findspark.init("/opt/spark-2.2.0-bin-hadoop2.7")
from pyspark.sql import SparkSession
from pyspark.sql.functions import split          # Function to split data
from pyspark.sql.functions import explode        # Equivalent to flatMap

# Create Session

```

```

spark = SparkSession.builder.master("local").appName("p6_df_count").getOrCreate()

# Read data
df_ulysses = spark.read.text("/home/tim/e63-coursework/hw3/data/ulysses10.txt")

# Split data
df_words = df_ulysses.select(split(df_ulysses.value, " ").alias("words"))

# Create one row per word
df_word = df_words.select(explode(df_words.words).alias("words"))

# Remove empty lines
df_word = df_word.filter('words != Null or words != ""')

# Output
print("Number of words: ")
print(df_word.count())

# Execute python script
/opt/spark-2.2.0-bin-hadoop2.7/bin/spark-submit \
  /home/tim/e63-coursework/hw3/scripts/p6_df_word_count.py

## Number of words:
## 267832

```

The results are the same as before, we're counting 267'832 words. With dataframes we can easily dig deeper into the data and look at the number of unique words in the text.

ii) Number of unique words (optional)

Having the total number of words we can next have a look at the amount of unique words. This is relatively easy as we just have to perform a `groupBy` and then count the number of lines.

```

# Get unique words
df_word_count = df_word.groupBy("words").count()

print("Number of unique words: ")
print(df_word_count.count())

## Number of unique words:
##
[Stage 1:> (0 + 0) / 200]
[Stage 1:=====> (72 + 1) / 200]
[Stage 1:=====> (98 + 1) / 200]
[Stage 1:=====> (142 + 1) / 200]
[Stage 1:=====> (187 + 1) / 200]

52982

```

There are 52'982 unique word in the dataset. Having that number we can have a look at the data at hand. We're going to do that here with SparkR in order to show, that the API can easily also be accessed from R.

iii) SparkR (optional)

Instead of using the Python API we can also use the R API accessing Spark. In order to have an comparable example out of R we're first replicating the above answer with the R API.

Start session

```
if (nchar(Sys.getenv("SPARK_HOME")) < 1) {  
  Sys.setenv(SPARK_HOME = "/opt/spark-2.2.0-bin-hadoop2.7/")  
}  
  
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"),  
                                       "R", "lib")))  
  
sparkR.session(master="local", sparkConfig=list(app_name = "p6_df_count"))  
  
## Launching java with spark-submit command /opt/spark-2.2.0-bin-hadoop2.7//bin/spark-submit sparkr-s  
## Java ref type org.apache.spark.sql.SparkSession id 1
```

Prepare data and count

```
# Read data  
df_ulysses <- read.text("/home/tim/e63-coursework/hw3/data/ulysses10.txt")  
  
# Create one row per word  
df_words <- selectExpr(df_ulysses, "split(value, ' ') AS value")  
  
# Explode  
df_words <- select(df_words, explode(df_words$value))  
  
# Remove empty lines  
df_word <- filter(df_words, 'col != Null or col != ""')  
  
# Collect  
count(df_word)  
  
## [1] 267832
```

We can see that we're getting the same count of words as with the python example. However, what we didn't do before was having a closer look at the counted word. With the great graphing capabilities of R this is an easy task. However, in order to do this we have to load the data into R, i.e., to collect them.

Transform data

```
# Get the data  
df_word <- collect(df_word)  
  
# Transform into a usable dataframe  
df_word <- magrittr::extract2(df_word, 1)  
df_words <- data.frame(table(df_word))  
names(df_words) <- c("words", "freq")  
  
# Order the vector according to frequency  
df_words <- df_words[order(df_words$freq, decreasing=TRUE), ]
```

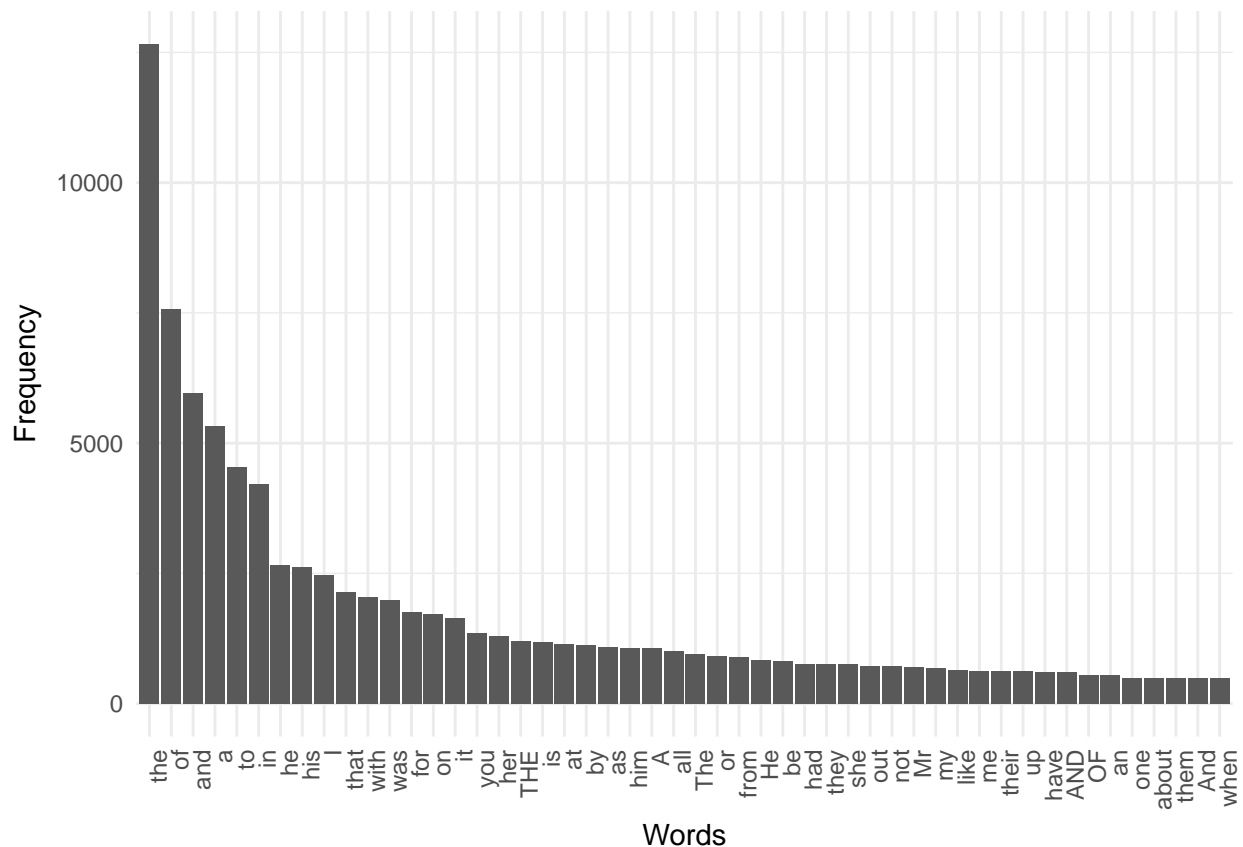
```
df_words$words <- factor(df_words$words,
                        levels=df_words$words[
                          order(df_words$freq, decreasing=TRUE)])
```

Having a dataframe with unique words lets us visualize the frequency of those words.

Plot I: Frequency of words in the Ulysses text

```
library(ggplot2)

ggplot(df_words[1:50, ], aes(words, freq)) +
  geom_bar(stat="identity") +
  theme_minimal() +
  ylab("Frequency") +
  xlab("Words") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

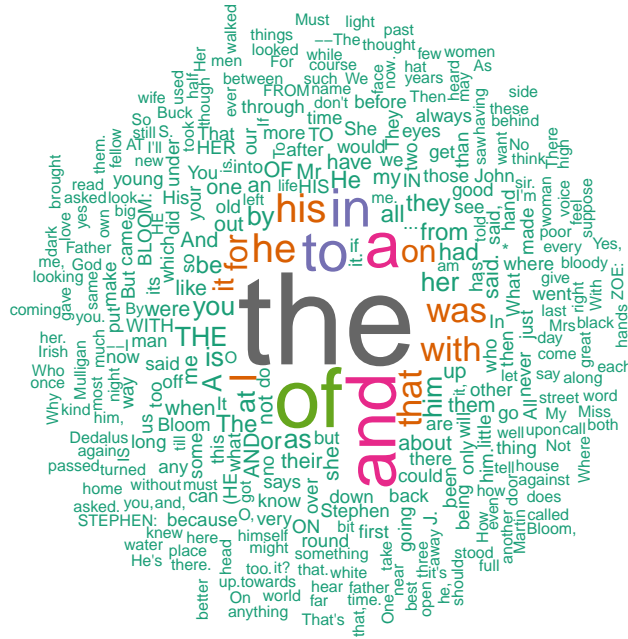


The plot I above shows, that the most used words are as expected filler words. However, we can also see that the cleanup wasn't performed properly as lower and upper case words are still present. In order to cleanup the words and punctuation marks we could use the *lower* and *regex_replace* command. We're not doing that here as we wanna keep the results congruent with the *wc* benchmark. As a last step we're visualizing the words in a wordcloud.

Plot II: Wordcloud

```
set.seed(1234)
wordcloud::wordcloud(words=df_words$words, freq=df_words$freq, min.freq=1,
```

```
max.words=350, random.order=FALSE, rot.per=0.35,  
colors=RColorBrewer::brewer.pal(8, "Dark2"))
```



References

In order to solve some of the above problems I used the following sources.

- Source: <http://www.sparktutorials.net/Getting+Started+with+Apache+Spark+RDDs>
- Source: <https://datamize.wordpress.com/2015/02/08/visualizing-basic-rdd-operations-through-wordcount-in-pyspark/>
- Source: <http://www.mccarroll.net/blog/pyspark2/index.html>