Course Assignment

Master of

Applied Computer Science

Department of Technology

Assignment title	Exploring Bitcoin
Course code	MA120
Course name	Big Data
Due date	18.October 2019

Declaration:

Through the submission of this assignment, we hereby declare that this report is the result of our own work, and that all sources have been properly cited to throughout the text.

Names of students	Theodor Midtbø Alstad ; Howie Chen
Student ID numbers	865317 ; 866354

Exploring Bitcoin

ABSTRACT

In this assignment we parsed a dataset, in the form of XMLs, through a Hadoop- and Pig apache. We learned how to write MapReduce jobs and Pig scripts, and how effective it is on our dataset based on XML files from Bitcoin stack exchange.

Contents

1 Introduction

This report is of a structure where each task is separately reported, frequently referring to each other due to their similarity. Each contains our assumptions about what the task is asking for, implementation, and any notes/reflections if necessary, opting to add notes/reflection more than necessary rather than less than necessary.
We chose to work together because both of us have python background, choosing python as main programming language. We explored a Bitcoin StackExchange dataset based in XML-files as given from archive.org/download/stackexchange/bitcoin.stackexchange.com.7z (accessed 16.10.2019) through Hadoop and Pig.
Hadoop is used to store and process big data, it has enormous processing power potential and has the ability to handle virtually any number of tasks and jobs. Hadoop provides a framework which allow users to write and test distributed systems and does not rely on hardware to provide fault tolerance, making it less apt for our small-scale applications, yet we use it for the experience. Pig is a MapReduce abstraction, working on Hadoop methods, that we use in some tasks due to its efficiency. We have elected to use both pig-scripts and pure MapReduce for very similar tasks (see 2c and 2d) to both show our ability to employ both methods and get experience with the two methods.

2 Main functions

The datasets consists only XML files which, being of a document-oriented database, is structured into attributes attributed to each element of the file. Being oriented towards use in HTML, i.e. a website, the attributes contain ascii characters, punctuation, numbers and HTML tags that we don't want to be there. For this we designed the function cleanBody, meant to clean the text gained from the XML-documents. Additionally a function to parse the XML files was required, for which we created XmlParser. Lastly, we created a general function that would perform the duties of putting out the results of the mapper function; mapper_core. This section explains their implementations, details the modules imported and describes the bash-script used to run our python-scripts:

2.1 cleanBody

The cleanBody function formats strings to be parseable by python interpreters. The function removes case sensitivity, ascii characters, HTML formatting, and treats anything separated by blanked space or / as separated words. This simple implementation of text formatting will affect the results of some tasks. For instances the name "Jens-Petter" will be interpreted as the word "jenspetter" and the filepath /documents/folder/file.extension will be interpreted as the words "documents", "folder" and "fileextension".
The function takes a string as input, formats it and outputs a list of all the words split by the string.split(" ") function. It uses the sub-function of the open source re module to remove HTML-tags and ascii function that is inherent starting with in python 3 to convert all characters to parse-able ascii text. The symbols in ignore_char, which are characters we are not considering is defined by the builtin string functions string.punctuation and string.digits.

```
def cleanBody(body):
    body = body.lower()
    body = ascii(body)
    body = sub("<.+?>","",body)
    body = body.replace("/", " ")
    body = body.strip()

    for i in ignore_char:
          body = body.replace(i, "")

    body = body.split(" ")

    return body
```

Listing 1. cleanBody function

2.2 Mapper Core

mapper_core is the core of the mapper function; it prints out the relevant data in a format parseable by Hadoop. It functions through three modes, as determined by the parameter mode: "single", "double", and "triple".

- Single: Assumes input "words" is a list of words to print. Prints word in words as (word, 1). Ignores empty strings and spaces.
- Double: Assumes input "words" is two nested lists to print. Prints word, count in words as (word, count). Ignores empty strings and spaces.

- Triple: Assumes input "words" is three nested lists to print. Prints id, score, title in words as (id, score, title). does not ignore empty strings and spaces.

```
def mapper_core(words, mode="single"):
    if mode == "single":
        for word in words:
            if word not in ["", " "]:
              print("%s %s" %(word,1)) #Emit the word

    elif mode == "double":
        in1, in2 = words
        for word, count in zip(in1,in2):
            if word not in ["", " "]:
                print("%s %s" %(word,count)) #emit the words

    elif mode == "triple":
        in1, in2,in3  = words
        for id, score, title in zip(in1,in2,in3):
            print("%s %s %s" %(id,score, title)) #emit the words
```

Listing 2. mapper_core function

2.3 xmlparser

The xmlparser function parses an xml using the open-source xml.etree.ElementTree. For normal use it uses sys.stdin as input, but for debugging purposes it also contains a clause that lets it accept a string (see the isinstance(infile, str) check) as a path to the location of an XML-file.
Although parsing an entire XML-file takes up significant memory, this method fits our dataset. It has been separated out as a function so it may be easily replaced by other methods more fit for large files. The input for this function is a XML-file and the output is a parsed XML-file.

```
def xmlparser(infile):
    if not isinstance(infile, str):
        infile = infile.detach()
    mytree = ET.parse(infile)
    myroot = mytree.getroot()
    return myroot
```

Listing 3. xmlparser function

2.4 Import sys

The sys module is an open-source, built in python function that allows us to add to the module search path with sys.path.append(*../*), enabling finding ProjectFunctions in a parallel folder, and read in the input for our mapper function and reduce function using sys.stdin.

```
import sys
sys.path.append(\'../\') #allows access functions in parallel folder
import ProjectFunctions.functions as proj
```

Listing 4. Import for sys

2.5 Import xml.etree.ElementTree

ElementTree is a part of the xml open-source library. XML is an inherently hierarchical data format, being document-oriented, and is naturally represented using a tree-like structure. By using the library ElementTree will represent the XML document as a tree, and elements of the tree is represented as single nodes.

```
import xml.etree.ElementTree as ET
```

Listing 5. Import for ElementTree

2.6 Import re.sub

re.sub is an open-source module we use in cleanBody to remove HTML-tags by removing anything that appears between two chevrons.

```
from re import sub
```

Listing 6. Import for re.sub

2.7 Bash-script

As an example of files being run, this bash-script that was used while testing out scripts is provided. It was used on a windows PC, meaning scripts had to be converted to unix code. It automatically names, cleans, and saves outputs locally. In this example task 4 index is being run. Other examples are 2c using task=2c and taskname=TopMiners and xmlsource=users.

```
# Assumptions:
# 1. The current directory contains files mapper.py and reducer.py for mapper and reducer code
    respectively
# 2. The tasks are named according to the schema of mapper being named as [tasknumber][taskname].py and
    the reducer being named as [tasknumber]reducer.py

#Simplify task change further
task=4
taskname=index
xmlsource=posts

#Simplify task change
mapperfile=$task$taskname.py
reducerfile="$task"reducer.py
outfile=output$task

#Automatic removal
hadoop fs -rm -r $outfile

#For Windows-compatibility
apt-get install dos2unix
dos2unix $mapperfile
dos2unix $reducerfile

#Main MapReduce function
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
-files $mapperfile,$reducerfile \
-mapper $mapperfile \
-reducer $reducerfile \
-input $xmlsource.xml \
-output $outfile

#Read file and save it locally
hadoop fs -cat $outfile/*
hadoop fs -copyToLocal $outfile

#Automatic cleanup
hadoop fs -rm -r $outfile
```

bash-script for a windows-based environment

3 Task 1 Warmup

3.1 WordCount

Assumption: Count the words in body of questions, identified by PostTypeID="1", in *Posts.xml*. The result should be a list of each word and how many times it occurs in the bodies of questions.

Implementation: The mapper uses the xmlparser function to parse an XML-file, iterates through all rows in parsed file, cleans text using the cleanBody function and prints output for the reducer script by using the mapper_core function. The reducer receives the mapper output, parses it and iterates through the parsed lines. A counter variable is used to track repeated words, outputting the word with the counter when the word stops repeating.

Notes/Reflection: Here the choices made building the cleanBody function affects the output, interpreting some things as words that clearly are not. See line 7 in 1a_output for examples; what is possibly a
, an html line break element, is interpreted as the word abr.

```
abovennthanks 1
abovennthanksn 2
```

```
abovennwhat 1
abovennwill 1
aboventhe 1
aboventrying 1
abr 1
abra 4
abras 1
abreast 1
abridge 1
```

1a_output

3.2 Unique words

Assumption: Write a MapReduce job which outputs unique words in the titles of questions, identified by PostTypeID="1", in *Posts.xml*. The result should be a list of each word as it appears in the titles of questions.

Implementation: This MapReduce job functions as 1a WordCount does, except it does not count the words and it looks through the titles instead of the bodies.

Notes/Reflection: Here again the choices made building the cleanBody function affects the output.

```
asm
asn
aspect
aspects
aspnet
assemble
assembled
assembles
assembling
assert
assertion
```

1b_output

3.3 MoreThan10

Assumption: Write a simple python script to check the amount of words in titles of posts in *Posts.xml*. The result should be a count of how many titles have more than 10 words.

Implementation: The mapper functions as the mapper in 1a WordCount, except it keeps a counter for the titles of more than 10 words and it outputs the final counter instead of using mapper_core. The reducer receives the mapper outputs and sums them up.

Notes/Reflection: This MapReduce job is simpler, the reducer is unnecessary outside an environment with clusters.

```
7600
```

1c_output

3.4 Stopwords

Assumption: Write a simple python script based on task 1a to exclude stopwords: from body of questions, identified by PostTypeID="1", in *Posts.xml*. The output should be a word count excluding any stop words.

Implementation: This MapReduce job functions as 1a WordCount does, except it has a section for the iterative removal of stopwords, as defined in StopWords.txt.

Notes/Reflection: Given how we built cleanBody, we chose to remove the apostrophes (') from the stopwords, turning words like "we're" to "were".

```
ati 8
atiradeon 1
atlcoin 1
```

```
atm 18
atms 5
atom 1
atomic 17
atomically 1
attach 8
attached 6
```

1d_output

3.5 Pig top 10

Assumption: Write a pig script to select top 10 listed words after removing the stopwords from *Posts.xml*. The output should be a list of the top 10 listed words and the corresponding occurrence count.

Implementation: Given this task being based on 1d StopWords, the mapper and reducer are the same for both tasks. Our pig loaded the output from the MapReduce job, ordered it by count, limited it to 10 elements and stored it into a text file: 1ePig_Output.txt.

Notes/Reflection: We chose for this pig script to load its dataset from an hdfs cluster as opposed to a local storage location.

```
bitcoin 5918
transaction 2376
wallet 2320
address 1550
block 1222
mining 1172
blockchain 1169
the 1148
transactions 1127
bitcoins 1050
```

1ePig_output

3.6 Tags

Assumption: Write a MapReduce job to create a dictionary over unique tags in *Posts.xml*. The result should be a list of unique tags.

Implementation: This MapReduce job functions similarly to 1a WordCount, except it has to be aware of whether the post has a tag at all or not, being implemented when tags are extracted from the row.

Notes/Reflection: The tags are separated by chevrons, and would be fused by cleanBody, so we elected to replace them with spaces. Some of the posts lacked any tag attribute, so we elected to pass over those.

```
moneylaundering
moneysupply
moneytransfer
movecmd
msigna
mtgox
multibit
multibithd
multifactor
multigateway
multipartycomputation
```

1f_output

4 Task 2 Discover

4.1 Counting

Assumption: Write a MapReduce job to count the total unique users there are in *Users.xml*. The result should be a count of how many unique users there are.

Implementation: This MapReduce job functions similarly to 1c MoreThan10, except it counts all unique users instead of titles with more than 10 words.

Notes/Reflection: We interpreted "unique user" as a user with a unique Id attribute.

```
56451
```

2a_output

4.2 Unique users

Assumption: Write a MapReduce job based on 2a Counting to list the unique users in *Users.xml*. The result should be a list containing unique users in the dataset.

Implementation: This MapReduce job functions as 2a Counting does, except it outputs all unique users instead of a count of them

Notes/Reflection: Here the implementation of cleanBody is modified with a join function as cleanBody returns a list where we want a string. Since usernames can contain spaces, we elected to instead use | as a separator in hadoop.

```
10597 frabcus
10599 mediatormatthews
105 dh  bitcoinse
10600 lucifirius
10601 adithya
10602 user
10603 amb
10609 ivan malyshev
1060 nlovric
10610 ajeet ganga
10611 michael merickel
```

2b_output

4.3 Top miners

Assumption:Write a pig script or MapReduce job to find top 10 users based on Reputation attribute in *Users.xml*. The result should be a list of the 10 users based with the highest reputation value.

Implementation: This mapper job is based on 2b Unique users, except instead of putting out name and id, it puts out name and the reputation attribute. Here, reputation is given to mapper_core as a one-element list (mapper_core called with arguments (name, [reputation])) due to it looping through its inputs. The reducer is the same as the reducer of 2b Unique users. The pig script functions the same way as in 1e Pig Top 10

Notes/Reflection: We elected here to use pig to order the MapReduce output in order to get experience.

```
david 87354
user 64135
pieter 51598
andrew 46149
murch 36524
thepiachu 31489
nick 29072
stephen 27509
chris 24736
nate 21620
```

2c_output

4.4 Top questions

Assumption: Write a pig script or MapReduce job to find top 10 title questions, identified by PostTypeID="1", in *Posts.xml* based on the Score attribute. The result should be a list containing the top 10 questions, showing id, question, and score.

Implementation: This mapper job functions similarly to 2c Top miners except it puts out 3 parsed attributes using the

"triple"-mode of mapper_core. The reducer performs the reduction and ordering, finding the 10 highest score values. It does this by inserting every input into a list instead of printing it, using pythons standard function string.sort to order it. It then iterates through the 10 highest values to give an output.

Notes/Reflection: The reducer could potentially be improved by only keeping 10 elements in the list at once, changing them out as higher scores were encountered.

```
336,    detecting dishonest mergedmining pool, 186
148,    what bitcoin mixing laundry services are availble today, 176
845,    change address after payment, 164
114,    how can i make my own pool, 151
3536,   blockchain for internetofthings or is there any mining algorithm that can be utilized in
    lowcomputational power devices preserving security, 142
8031,   what should i do if i change my bitcoin wallet, 136
9046,   how does bitcoin prevent someone from sending more money than he she has, 133
658,    is it possible for a country to control bitcoin usage, 132
91,     how does the mining process support the currency, 119
876,    is there a ripple whitepaper, 118
```

2d_output

4.5 Favorite questions

Assumption: Write a pig script or MapReduce job to find top 10 title questions, identified by PostTypeID="1", in *Posts.xml* based on FavouriteCount attribute. The result should be a list containing the top 10 questions, showing id, question and the favourite count.

Implementation: This MapReduce job is the same as 2d Top questions, except it extracts the FavoriteCount-attribute instead of the score

Notes/Reflection: Here, like 1f Tags, some posts had to be skipped due to their lacking any FavoriteCount-attribute.

```
148,    what bitcoin mixing laundry services are availble today, 103
336,    detecting dishonest mergedmining pool, 95
8031,   is this calculation of mining probability using the bernoulli trial formula accurate, 83
12427,  can i force my wallet to only have news keys postencryption, 60
9046,   how does a client know it is connected to the right pp network, 60
3374,   can a mobile be protected against the linode problem, 52
3536,   how does shapeshift operate on  confirmations, 52
12670,  what is a mining pool what is it good for, 48
658,    is it possible for a country to control bitcoin usage, 47
118,    is there a mtgox live alternative which supports euro, 46
```

2e_output

4.6 Average answers

Assumption: Write a MapReduce job that calculates the average number of answers per question. The result should be a calculated average of all AnswerCount attributes.

Implementation: This mapper functions similarly to 1c MoreThan10, except for having two counters, one for score and the other for users parsed, which are averaged at the end. The reducer is the same as the reducer in 2a Counting

Notes/Reflection: We have elected to not limit the amount of decimals.

```
1.4316527838667252
```

2f_output

4.7 Countries

Assumption: Write a MapReduce job to count the amount of users attributed to a country by the Location attribute in *Users.xml*. The result should be a list of different countries and corresponding users.

Implementation: This mapper functions similarly to 1f tags, except instead of cleaning the tags, it uses "|" as a separator, since spaces are part of location. If Location is separated by a ¸ıthe last part is treated as the country.

Notes/Reflection: Because by our judgement locations are typed in by users, we have elected to separate the country out of Location very simply by using ¸ıas a separator. Anything after the last ¸ıis treated as a country. A location with no ¸ıis treated as a country. Locations like "California, USA", "California, United States", and "US" will be interpreted as "usa", "united states", and "us", respectively.

```
f starworld hotel avenida da amizade macau  1
godavripark soc yogichowk surat gujarat india pin  1
arakawa arakawa city uecueacufdueuc  1
bay dr bay point ca usa  1
beulah way se conyersga united states  1
canada  1
candiles mxexico  1
corners  1
dev null  4
florabunda lane lefevres corner nsw  1
```

2g_output

4.8 Names

Assumption: Write a pig script or MapReduce job to find the 10 most popular names in *Users.xml*. The result should be a list top 10 common names.

Implementation: This mapper functions similarly to 2g Countries, except it extracts the DisplayName-attribute in place of the Location-attribute. The reducer functions similarly to 2d Top questions, except it parses differently and has different text formatting for the output due to the decrease from 3 to 2 arguments

Notes/Reflection: Here the most common name is User. This is due to our choice of removing numbers in cleanBody, combining User1, User2, User3 etc. The task suggested explicitly that we should split up names by spaces, perhaps wanting to separate out last names from first names. Due to the nature of Displaynames usually not being full legal names, we have elected to interpret a "name" as the full name given in DisplayName.

```
user  3584
alex  102
john  99
david  84
mike  74
chris  70
mark  68
michael  65
peter  55
sam  53
```

2h_output

4.9 Answers

Assumption: Write a python script to find how many questions, identified by PostTypeID="1", in *Posts.xml* have at least one answer based on attribute AnswerCount. The result should be a count of how many questions have been answered.

Implementation: This mapper functions in the same way as 1c MoreThan10 except it checks whether AnswerCount is larger than or equal to 1 instead of whether the title word length is larger than 10.

```
20492
```

2i_output

5 Task 3 Numbers

5.1 Bigram

Assumption: We chose to write a MapReduce job to find the most common pair of adjacent words in the titles of questions, identified by PostTypeID="1", in *Posts.xml*. A bigram is a combination of two words in a sequence. For instance the sentence

"Big data is big" contains the bigrams "Big data", "data is", and "is big". The result should be the most common bigram along with a count of how many times it occurs.

Implementation: This MapReduce is exactly the same as 1a WordCount, except we output each bigram in the body instead of each word. Because of this we he had to forego the mapper_core for a print function inside a loop looping through every bigram. The reducer finds the most common bigram by iteratively comparing each elements second element (the bigrams count) against a variable declared maximum.

Notes/Reflection: In the print-loop we had to limit the loop to i in range(len(words)-1), since we printed both words[i] and words[i+1], and doing otherwise would result in and out of range error.

```
how to  , 1825
```

3a_output

5.2 Trigram

Assumption: Perform the same task as in 3a Bigram, except operate with trigrams (sequences of 3 words as opposed to 2). The result should be the most common trigram.

Implementation: This is exactly the same as 3a Bigram, except we output trigrams instead of bigrams.

Notes/Reflection: The mapper, as in 3a Bigram, limited the print-loop to range(len(words)-2) and printed words[i], words[i+1], and words[i+2].

```
what is the  , 603
```

3b_output

5.3 Combiner

Assumption: Perform the same task as in 1a WordCount, except add a combiner before the mapper sends information to the reducer. The result should be the same as in 1a WordCount, but the volume of the data transfer should be smaller.

Implementation The MapReduce function is the same as in 1a WordCount, with a combiner ahead of the print function in the mapper.

Notes/Reflection: Having a combiner in a MapReduce job saves bandwidth and computational strain by decreasing the volume of data sent from the mapper. This is less relevant in our case, working with relatively small datasets.

```
a 47804
aa 16
aaa 7
aaaa 3
aaaaaaaaaaaaaaaaaaaaanconvert 1
aaaaaaaaaaaaaaaaaaaaaunntrying 1
aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbxxxxxxxxxxxxxxxxxxxxntime 1
aaaaaaaaaaaazuioarpxpqbcznnaltvmjjswobclbn 1
aaaaannn 1
aaaabedbeffdefdebdafbedebnnout 1
```

3c_output

5.4 Useless

Assumption: Write a MapReduce job to find how many times the word "useless" in occurs in the bodies of questions, identified by PostTypeID="1", in *Posts.xml*. The result should be a count of how many times the word useless occurs.

Implementation: This mapper functions similarly to 1c MoreThan10 and 2i Answers, except instead of checking for titles with more words than 10 or whether AnswerCount is larger than or equal to 1, it counts whether the word "useless" occurs in the body.

```
The word useless occurs 62 times
```

3d_output

6 Task 4 Search engine

6.1 Title index

Assumption: Write MapReduce job to create index over titles, bodies and answers of questions in *Posts.xml*. The result should be a simple index that lists posts words appear in by their Id's.

Implementation: This mapper function parses through all rows in posts.xml. It then forks over whether PostTypeId is 1 or 2, i.e. whether it is an answer or question, extracting the relevant id, body and title from the row. It contains a combiner to speed up the process, removing duplicates. The reducer parses the input and iterates through all the lines. It uses two variables to know what the last word and associated Id was, skipping the word if both are the same, adding the Id to a list if *only* the word is the same, and putting out the word with an associated list of Id's if neither is the same, i.e. it has finished working with the word.

Notes/Reflection: We elected to have a combiner in the mapper although it is not strictly necessary, since we are working with small file-sizes.

```
angel, ,11695,8519,5197
angeles, ,5197,11695
angellist, ,11695
angels, ,11695,687
anger, ,2749
angered, ,11912
angle, ,2937,4110,1224
angles, ,1859
angry, ,2240,10044
```

4a_output

7 Conclusion

Through this assignment we gained experience in using MapReduce jobs and pig scripts. We parsed XML-documents and extracted specific information we were looking for to be formatted to an output to each tasks.