



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, 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](https://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 contains only XML files, which it means values to different attributes have a lot of ascii characters, punctuation, numbers and HTML tags. We also needed a function to parse the XML files to Hadoop and to create a mapper for the data. Therefor we created three different main functions thats being reused through the project. :

### 2.1 CleanBody

In cleanBody function it formats strings for mapper function. The function makes non-case sensitivity, removes ascii characters , HTML formatting, and treats anything separated by blanked space or as separated words. Most of the words after separating/splitting will be counted as separate words, and this will effect the results of some tasks. For instance the name "Jens-Petter" will be counted as two words; "Jens" and "Petter". The input for the function is a string body which will be formatted and it returns a list of formatted string.

```
1 def cleanBody(body):
2     body = body.lower()
3     body = ascii(body)
4     body = sub("<.+?>", "", body)
5     body = body.replace("/", " ")
6     body = body.strip()
7
8     for i in ignore_char:
9         body = body.replace(i, "")
10
11     body = body.split(" ")
12
13     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.

```
1 def mapper_core(words, mode="single"):
2     if mode == "single":
3         for word in words:
4             if word not in ["", " "]:
5                 print("%s %s" % (word, 1)) #Emit the word
6
```

```

7 elif mode == "double":
8     in1, in2 = words
9     for word, count in zip(in1,in2):
10         if word not in ["", " "]:
11             print("%s %s" %(word,count)) #emit the words
12
13 elif mode == "triple":
14     in1, in2,in3 = words
15     for id, score, title in zip(in1,in2,in3):
16         print("%s %s %s" %(id,score, title)) #emit the words

```

**Listing 2.** mapper\_core function

## 2.3 Xmlparser

The chosen method to interpret the dataset is to parse to the mapper function. 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

```

1 def xmlparser(infile):
2     if not isinstance(infile, str):
3         infile = infile.detach()
4     mytree = ET.parse(infile)
5     myroot = mytree.getroot()
6     return myroot

```

**Listing 3.** xmlparser function

## 3 Task 1 Warmup

### 3.1 WordCount

**Assumption:** Count the words in body of questions PostTypeID="1" in the *Posts.xml* file. The result should be how many times a word occur in the body 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 loops through the output. 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 example; a<br>, an html line break element, has been turned into the not-word abr.

```

1 abovennthanks 1
2 abovennthanksn 2
3 abovenwhat 1
4 abovenwill 1
5 aboventhe 1
6 aboventrying 1
7 abr 1
8 abra 4
9 abras 1
10 abreast 1
11 abridge 1

```

1a\_output

### 3.2 Unique words

**Assumption:** Write a MapReduce job where the result should be unique words in the titles PostTypeID="1" in the *Posts.xml* File.

**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.

```
1 asm
2 asn
3 aspect
4 aspects
5 aspnet
6 assemble
7 assembled
8 assembles
9 assembling
10 assert
11 assertion
```

1b\_output

### 3.3 MoreThan10

**Assumption:** Write a simple python code to check the title length in *Posts.xml*. The result should output 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.

```
1 7600
```

1c\_output

### 3.4 Stopwords

**Assumption:** Write a simple python code based on task 1a to exclude [stopwords](#) from body of questions PostTypeID="1" in the *Posts.xml*. The output should be text file without any stopwords.

**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"

```
1 ati 8
2 atiradeon 1
3 atlcoin 1
4 atm 18
5 atms 5
6 atom 1
7 atomic 17
8 atomically 1
9 attach 8
10 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 print out top 10 listed words and the corresponding occurrence rate.

**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

```
1 bitcoin 5918
2 transaction 2376
3 wallet 2320
4 address 1550
5 block 1222
6 mining 1172
7 blockchain 1169
8 the 1148
9 transactions 1127
10 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 print the 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.

```
1 moneylaundering
2 moneysupply
3 moneytransfer
4 movecmd
5 msigna
6 mtgox
7 multibit
8 multibithd
9 multifactor
10 multigateway
11 multipartycomputation
```

1f\_output

## 4 Task 2 Discover

### 4.1 Counting

**Assumption:** We chose to write a MapReduce job to count the total unique users there are in *Users.xml*. The result will print out 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.

```
1 56451
```

2a\_output

### 4.2 Unique users

**Assumption:** Write a MapReduce job based on 2a Counting to create a mapper and a reducer functions in *Users.xml*. The result should contain 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.

```
1 10597 frabcus
2 10599 mediatorMatthews
3 105 dh bitcoinse
4 10600 lucifirius
5 10601 adithya
6 10602 user
7 10603 amb
8 10609 ivan malyshev
9 1060 nlovric
10 10610 ajeet ganga
11 10611 michael merickel
```

2b\_output

### 4.3 Top miners

**Assumption:** Write a MapReduce job to find top 10 users based on attribute Reputation="x" in *Users.xml*. The result will print out top 10 users based on their reputation.

**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.

```
1 david 87354
2 user 64135
3 pieter 51598
4 andrew 46149
5 murch 36524
6 thepiachu 31489
7 nick 29072
8 stephen 27509
9 chris 24736
10 nate 21620
```

2c\_output

### 4.4 Top questions

**Assumption:** Write a MapReduce job to find top 10 title questions PostTypeID="1" based on attribute Score="x" in *Posts.xml*. The result lists top 10 questions using id, question and the 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 python's 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.

```
1 336, detecting dishonest mergedmining pool, 186
2 148, what bitcoin mixing laundry services are available today, 176
3 845, change address after payment, 164
4 114, how can i make my own pool, 151
5 3536, blockchain for internetofthings or is there any mining algorithm that can be utilized in
lowcomputational power devices preserving security, 142
6 8031, what should i do if i change my bitcoin wallet, 136
7 9046, how does bitcoin prevent someone from sending more money than he she has, 133
8 658, is it possible for a country to control bitcoin usage, 132
9 91, how does the mining process support the currency, 119
```

```
10 876, is there a ripple whitepaper, 118
```

2d\_output

#### 4.5 Favorite questions

**Assumption:** Write a MapReduce job to find top 10 title questions PostTypeID="1" based on attribute FavoriteCount="x" in *Posts.xml*. The result lists top 10 questions like id, question and the score.

**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.

```
1 148, what bitcoin mixing laundry services are availble today, 103
2 336, detecting dishonest mergedmining pool, 95
3 8031, is this calculation of mining probability using the bernoulli trial formula accurate, 83
4 12427, can i force my wallet to only have news keys postencryption, 60
5 9046, how does a client know it is connected to the right pp network, 60
6 3374, can a mobile be protected against the linode problem, 52
7 3536, how does shapeshift operate on confirmations, 52
8 12670, what is a mining pool what is it good for, 48
9 658, is it possible for a country to control bitcoin usage, 47
10 118, is there a mtgox live alternative which supports euro, 46
```

2e\_output

#### 4.6 Average answers

**Assumption:**

**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 1.4316527838667252
```

2f\_output

#### 4.7 Countries

**Assumption:** We chose to write a MapReduce job to discover users by countries in *Users.xml*. The result lists 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.

**Notes/Reflection** We have elected to separate by location and not country, since by our judgement these locations are typed in by users. Places like "California, USA", "California, United States", and "Washington DC, USA" will thus be counted as different locations.

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

2g\_output



## 4.8 Names

**Assumption** We chose to write a MapReduce job to find the most popular names in *Users.xml*. The result lists top 10 common names and how many.

**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.

```
1 user 3584
2 alex 102
3 john 99
4 david 84
5 mike 74
6 chris 70
7 mark 68
8 michael 65
9 peter 55
10 sam 53
```

2h\_output

## 4.9 Answers

**Assumption:** Write a simple python code to find how many titles of questions PostTypeId="x" have at least one answers based on attribute AnswerCount in *Users.xml*. The result prints out 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

```
1 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 *Posts.xml*. For instance, "big data" or "Fast car" are examples of bigram. The result print the most common bigram

**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 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.

```
1 how to , 1825
```

3a\_output

### 5.2 Trigram

**Assumption:** This task is based on 3a, to find three words that appear consecutively in *Posts.xml*. The result print 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].

```
1 what is the , 603
```

3b\_output

### 5.3 Combiner

**Assumption:** Write a MapReduce job and add a combiner before the data sent to the reducer. The result should be the same as a reducer, but the reducer receive a smaller volume of data.

**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.

```
1 a 47804
2 aa 16
3 aaa 7
4 aaaa 3
5 aaaaaaaaaaaaaaaaaaaaaanconvert 1
6 aaaaaaaaaaaaaaaaaaaaaunntrying 1
7 aaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbxxxxxxxxxxxxxxxxxxxxxtime 1
8 aaaaaaaaaaaaazuioarpxpqbzcnnaltvmjjswobclbn 1
9 aaaaannn 1
10 aaaabedbeffdefdebdafebdebnnout 1
```

3c\_output

### 5.4 Useless

**Assumption:** We chose write a MapReduce job to find how many times the word "useless" in *Posts.xml* occurs in the body of questions PostTypeId="1". The result print 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.

**Notes/Reflection**

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

3d\_output

## 6 Task 4 Search engine

### 6.1 Title index

**Assumption:** We chose to write MapReduce job to create index over titles, bodies and answers of questions in *Posts.xml*. We are after having a simple index that lists publications in which a search term/s and occur/s. The result is a list of words and their index appearance in posts LOL

**Implementation** This mapper function parses through all rows in posts.xml. It then forks over whether PostTypeId is 1 or 2, 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

**Notes/Reflection**

```
1 angel, ,11695,8519,5197
2 angeles, ,5197,11695
3 angellist, ,11695
4 angels, ,11695,687
5 anger, ,2749
6 angered, ,11912
7 angle, ,2937,4110,1224
8 angles, ,1859
```

9 angry, ,2240,10044

4a\_output

## Conclusion

MapReduce in Hadoop map a large dataset, then perform a reduce, on the specific output from the mapper.

The idea behind MapReduce is that Hadoop can first map a large data set, and then perform a reduction on that content for specific results. A reduce function can be thought of as a kind of filter for raw data. The HDFS system then acts to distribute data across a network or migrate it as necessary.