Big Data Portfolio Assignment – Part 1

# 3.1 – Importing data into HDFS

I downloaded the datafile via <https://www.openstreetmap.org/>. The Area I chose for my data was an area in the city Moss. The specific bounding box is: 59.4302 , 10.6557 , 59.4089 , 10.7023

I then created a folder (xmlhadoop) in the hdfs by running the command:

  hdfs dfs -mkdir /xmlhadoop

 I then uploaded the data file into the folder previously created using the following command:

  hdfs dfs -put Downloads/map.osm /xmlhadoop

When inserting a new file to the file system, several things happen:

Firstly, the client asks the namenode to create a file.

The namenode then responds to the request with a list of datanodes to store replica blocks on.

*The reason why hadoop has data replication is due to increased reliability. If for whatever reason a block cannot be accessed, it provides a backup block. This significantly diminishes the chances of a piece of data being lost forever and a piece of data being inaccessible.*

Upon recieving the replication block list, it starts writing data to the first datanode. From there the datanode will write to the next datanode in the list, and so on recursively. Every datanode that has the file discloses that to the namenode, so that the namenode may keep track of where the file and all its replicas are stored. Once writing to all the datanodes in the list has finished, the namenode is informed that the procedure has completed.

HDFS has a method in place that handles errors in the procedure, such as a datanode failing. If a datanode does fail, then the pipeline of datanodes writing to the next datanode is closed. The datanode that failed is given a new id which the namenode is informed of. From there, a new pipeline is created using the good datanodes, and the remaining data is written on that pipeline. The datanode that did fail will consequently be a missing replica. The namenode takes upon the responsibility of arranging for a replication of said missing replica.

The HDFS Replica Placement Policy is what differentiates HDFS from most other distributed file systems. It is a rack-aware replica placement policy that aims to improve data reliability, availability and network bandwidth utilization. In a large HDFS there are many racks. Communication between two racks must go through switches. So, in most cases, communication between computers in the same rack will be quicker. And of course, storing replicas on the same node would give fastest access but is pointless since if that node fails, all the replicas will be lost aswell.

The replication method is as follows :

1. Puts a replica on the node where the client is. If the client is not in the cluster then it just chooses a random node.
2. Another replica is put on a node in a different rack.
3. Another replica is put in the same rack as 2. - but placed in different node of course.
4. For every additional replica they shall be placed at random, while not allowing the number of replicas for a rack to be above a specific limit. The limit is: *(replicas - 1) / (racks + 2)*

Of course, the cluster setup used in the project is a single-node cluster. It will not have the efficiency of a multi-node cluster nor will it utilise the distributed processing. All it does is provide the daemons (Namenode, Datanode, Resource Manager etc) so it may simulate a cluster like-environment to enable testing of hadoop applications. It does not really serve any purpose other than being a useful tool for studying and testing.

In this single-node cluster the replication factor is set to 1. That makes sense of course because storing replicas on the same node is futile (they would not function as a backup).

# 3.2 – Updating data in HDFS

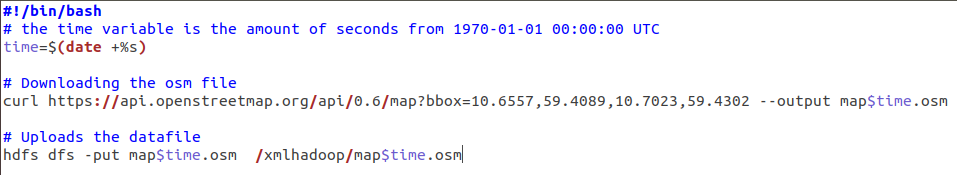
The HDFS does provide a command for overwriting existing files within the hdfs.

  hdfs dfs -put -f <localFile> <pathOnHadoop>

The -f parameter will overwrite the existing file.

However, that approach does not consider if the file being overwritten is currently used in a job. A safer approach would be to simply add the new data as a new seperate file. That means making sure the name of the file is different from an existing file in the same hadoop folder.

One could then simply delete the outdated datafile once it is no longer being used in a job and use the new updated file for any future jobs. The following script file downloads the osm file (by curling the API) and ensures the filename will be unique/different from previous osm files by using a timestamp ($time).

**updateData.sh**

# 3.3.1 – Simple Metrics

## Compiling and running the MapReduce program, and viewing the results

I created Bash files so that running the mapreduce programes would be faster and more user-friendly (on my end).

It is therefore a driving factor behind the structure and naming process for the contents of the hdfs.

For example, the actual programes themselves are named as "XMLHadoop{nr of task it solves}.java" which results in the following Bash scripts being able to compile and run tasks given the task nr as parameter:

**compile.sh**

Machine generated alternative text:
#! Ibin/bash 
CLASS PATH= 'hadoop classpath 
javac -classpath S{CLASS PATH} XMLHadoopS1.java 
jar cf prod/xmlhadoopS1.jar XMLHadoopS1*.class 

**run.sh**

Machine generated alternative text:
#! Ibin/bash 
hadoop jar prod/xmlhadoopS1.jar XMLHadoopS1 

All the mapreduce programs were placed within the "xmlhadoop" folder in the hdfs. Within the xmlhadoop folder is another sub-folder called "result". Each task has the output result folder as "result/opg{nr of task it solves}". Bash script for deleting previous output result, and script for reading the result are made as such:

**delete\_hadoop\_result.sh**

Machine generated alternative text:
#! Ibin/bash 
hadoop fs -rm -r Ixmlhadoop/result/opgS1 

**printResult.sh**

Machine generated alternative text:
#! /btn/bash 
hdfs dfs -cat Ixmlhadoop/result/opgS1/* 

The result from the jobs are folders. They are created by the following code:



The MapReduce job will check that the specified output directory does not already exist. If it does exist an exception will be thrown, and the directory will not be overwritten.

So, in order to re-run the program, the output directory must be deleted before writing to it again.

The resulting directory will be a directory that contains part-r files and a \_SUCCESS file. The \_SUCCESS file will be empty and serves as a tool to let applications check if the result set is complete. The part-r files are the result from the reducers. "-r" means it is the result of a reducer job and the numbers following the -r (f.example -0000) is the reducer task number.

When I have been working on MapReduce, I have used the following bash script:

**full.sh**

Machine generated alternative text:
#! /btn/bash 
sh delete hadoop_result.sh SI 
sh comptle.sh SI 
sh run.sh SI 
echo 
sh printResult.sh SI 

## Code Theory and Concepts Explained

### MapReduce

TODO XML reading

TODO Mapper class

The Reducer has 3 primary phases.

1. Shuffle – copying the sorted output from the Mappers.

2. Sort –

3. Reduce – Does something foreach key and all its values.

### Spark

TODO

## Simple Metrics Pseudocode

### Task 1 - How many buildings is it in the extract you selected?

#### MapReduce

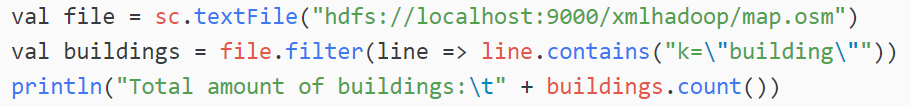
* TODO:: explain :: Input split on <way>
* Mapper: Outputs < ”Buildings”,1 > foreach instance of <tag k="building" .../>
* Combiner: Has a global int variable that initially is set to 0. In the reduce() method the variable adds the Int value of each value to the key “Buildings”.   
  Outputs < ”Buildings”, sum >.
* Reducer: Does the same as the Combiner (they are the same class)

Result:



#### Spark

Firstly, the datafile is loaded in as an RDD. Then I perform a filter() function on the RDD that filters out the lines containing the string ‘k=”building”’. Lastly, in the printout a count() function is run on the filtered RDD which results in the total number of buildings found in the datafile.



Result:



### Task 2 - How many addr:street tags exist for each street?

#### MapReduce

* TODO:: explain :: input split <way>
* Mapper: Outputs foreach instance of <tag k="addr:street»./>. Writes: <Streetname, 1>
* Combiner: An int variable ‘sum’ is declared in the reduce() method scope. It is initially set to 0. When looping the values for the keys, the ‘sum’ variable adds the value.  
  Outputs <Streetname, sum>
* Reducer: Same as the Combiner (is the same class).

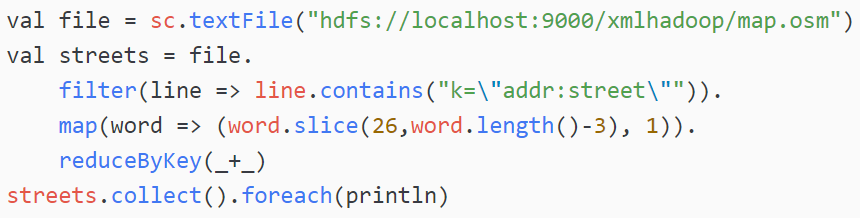
Result:

Machine generated alternative text:
Abborveten 
Bakkeveien 
Ballblomveten 
Betteveten 
Bekkeltveten 
60 
15 
36 
56 
25 
Birkebeinerveten 
Bjerkeveten 
55 
Bjørn Kristensens vet 
19 
12 
Blåbærstten 
Blåbæräsen 
Blåmetsveten 
Bokfinkveten 
17 
9 
20 
23 
Borgermester Aases gate 19 
Brasmeveten 
Bredalbakken 
Bredsandkroken 
20 
61 
6 

Etc…

#### Spark

The RDD is loaded in. Then, I filter out the lines that contain the 'k="addr:street"'. From there I use map() and reduceByKey() functions. The map function writes key as streetname by using slice() to extract the correct substring (static values for the slicing is used but should not be a problem as the format is always the same). It also writes 1 as value to reducer. The reducer then adds all the values for the same key.



Result:

Machine generated alternative text:
(Roseveten,4) 
(Kvtleveten,46) 
(Festeveten , 24) 
(ørekroken terrasse ,48) 
(Teglverksveten, 10) 
(Betteveten,56) 
(Vestre Ekholtvet,l) 
(Rådyrfaret,9) 
(Scharres gate,7) 

Etc…

### Task 3 - Which object in the extract has been updated the most times, and what object is that?

#### MapReduce

* TODO:: explain:: Input Split <osm>
* Mapper: Outputs foreach instance of <way/relation/node version="…"/> tags. Writes: <nodename , VersionNr>
* VersionNr is a class that implements Writable. It has String nodename, int updateAmount & long nodeId
* Combiner: Has a global int ‘highestVersionValue’ which is initially set to 0. When looping through all the values it compares the updateAmount with ‘highestVersionValue’, and if it is bigger than it keeps track of that VersionNr.  
  In the cleanup() method it outputs the VersionNr that had the biggest updateAmount.
* Reducer: Is the same as the Combiner (same class).

Result:



#### Spark

The program filters out all the lines containing "version=". From there it uses the map() function and produces => (*version\_nr\_as\_float*, *node\_name* with id *id*). The reason I converted the version nr to Float instead of Int is because I had some occurences of "1.0" in my datafile, which could not be converted into an Int. Since I put a Float as Key, the max() function can extract the highest key and the value based on the key. Which results in the following:



Result:



### Task 4 - Which 20 highways contains the most nodes?

#### MapReduce

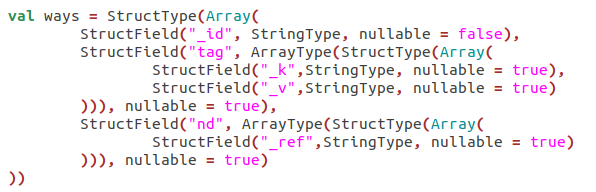
* TODO:: explain:: input slit <way>
* Mapper: A global TreeMap variable tmap is initialized in the setup() method.  
  The TreeMap has key IntWritable, and value List<Highway>. The key is going to be the amount of nd tags in the <way>. The reason the value is a list of highways and not highways themselves is so that different highways of same nr\_of\_nd\_nodes can be stored.   
  The map() method writes to the tmap foreach instance <way> that contains tag k=highway.  
  In the cleanup() method the TreeMap is shrunken to a size of 20. Then proceeds to write those to the output.
* Highway is a custom Writable. It has String wayId, String highwayType & int nrNdNodes.
* Reducer: The reducer also has a global TreeMap variable <IntWritable, List<Highway>> which is initialized in the setup() method. The reduce() method writes all the values to the treemap. In the cleanup() method the treemap is shrunken to a size of 20 and will be the output result.  
  Result:

A screen shot of a computer

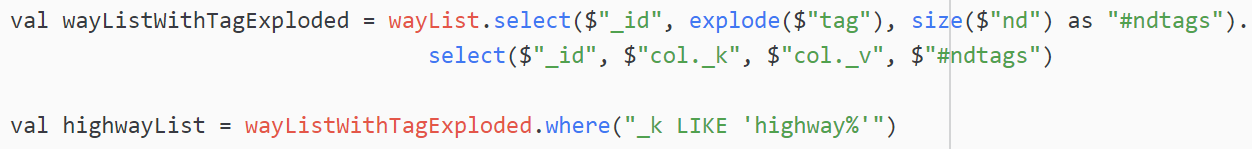
Description automatically generated

#### Spark

The solution is resolved by creating a dataframe which is suited for the wanted data from the osm file.



Upon loading the osm file it imports the data into the dataframe. Since there exists several <nd> and <tag> child elements they are created as arrays in the schema, which is why field “tag” and “nd” are of type ArayType. An aggregate function size() is done on the nd array to get the number of nds a way has. I also explode the tag array so that I can filter them out on the key value using the where() function.



Result:

Machine generated alternative text:
_tdl 
1 48934821 
1 201998981 
11725495891 
11728168281 
Il 799067911 
1 2932375411 
_v I #ndtags I 
terttary I 
residenttall 
cycleway I 
cycleway I 
cycleway I 
tertiaryl 
Il 46932721 unclasstftedl 
11001286351 unclasstftedl 
16590726261 residenttall 
11588619851 residenttall 
1 799134361 
cycleway I 
11586705671 residenttall 
1 3099590711 
moto rway I 
1 201996551 unclasstftedl 
1 48934581 unclasstftedl 
11588510871 residenttall 
11586716151 
servicel 
1 202930521 residenttall 
11588793851 residenttall 
1 3779162371 
cycleway I 
94 | 
64 
481 
391 
371 
371 
361 
361 
341 
331 
331 
331 
30 
271 
271 
271 
261 
261 
261 
261 

The result shows the 20 highways with most nd tags. Structured as: way id (\_id), and what type of highway it is (\_v) and the number of nd tags the way contained (#ndtags).

### Task 5 - What is the average number of nodes used to form the building ways in the extract?

#### MapReduce

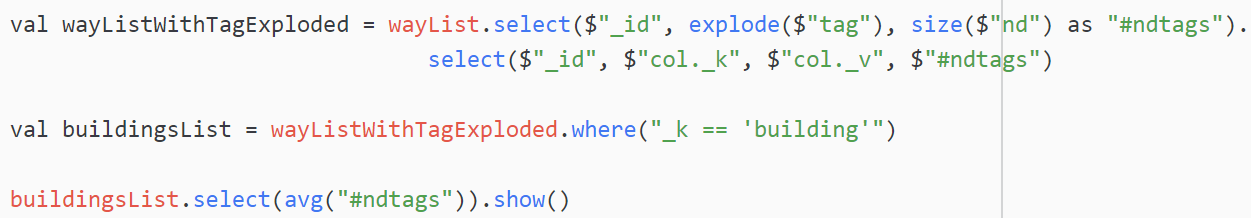
* TODO:: explain:: input split <way>
* Mapper: In the map() method the Mapper outputs <1, nrOfNdNodes> foreach instance of <tag k=building> in the <way> element.
* TODO:: Remove combiner and explaing that averages of averages isn’t accurate.Combiner: Has global int variables totalNdsNodes and totalBildings that are both initially set to 0. In the reduce() method the totalNdsNodes adds the value, and the totalBuildings adds the key.   
  In the cleanup() it writes <totalBuildings, FloatWritable(totalNdsNodes)>
* Reducer: Does the same as the Combiner. However, the global variable ‘totalNdsNodes’ is a float that already is the calculated average from the mappers.

Result:



#### Spark

The program does not use a custom dataframe when loading the <way> elements from the datafile. Instead the default dataframe it produces is used. The array(tag) is exploded to allow for filtering on tag values (key=building). The array(nd) is also converted to size(nd) to keep track of how many nd’s each building has. From there the avg() function is used on size(nd) column.



Result:

Machine generated alternative text:
| 6.1495327102803741 

### Task 6 - How many ways of types ”highway=path”, ”highway=service”, ”highway=road”, ”highway=unclassified” contains a node with the tag ”barrier=liftgate”?

#### MapReduce

* TODO:: explain:: input split <osm>
* The solution uses the ReduceSideJoin method, which means x2 mappers and x1 reducer

TODO: explain how multipleInputs launches multipl mappers

* NodeRefMapper: In the map() method the mapper outputs <node\_id, true> foreach instance of a node that contains <tag k=barrier v=lift\_gate/>
* HighwayWithSpecificValuesMapper: In the map() method outputs < node\_ref, false> foreach instance of <nd> elements in a <way> that contains <tag k=highway v=service/path/road/unclassified/>
* The OutputValue being a BooleanWritable is used to identify wether the map output was from the NodeRefMapper or the HighwayWithSpecificValuesMapper. (true means it was the output from NodeRefMapper, false means it was the output from HighwayWithSpecificValuesMapper).
* Reducer: Has a global int ‘totalOccurencesOfHighwaysThatReferToNode’.   
  In the reduce() method local-scope variables boolean nodeRefferedDoesHaveBarrierLiftGate is initially set to false and int nrOfHighwaysReferToNode set to 0. When looping through the values it sets the local scope boolean value to true if one of the values looped is true. If not, then it iterates the value ‘nrOfHighwaysReferToNode’ by 1.  
  After looping an if statement is used to verify if the key is outputted from both the NodeRefMapper and the HighwayWithSpecificValuesMapper.   
  

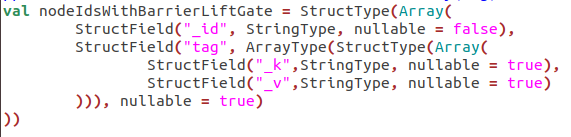
In the cleanup() method it outputs < "Total Nr of Highways that refer to Node with barrier lift\_gate", totalOccurencesOfHighwaysThatReferToNode>

Result:

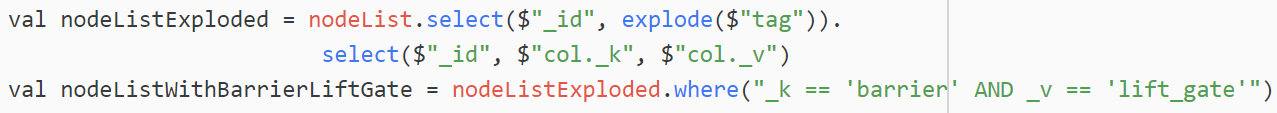


#### Spark

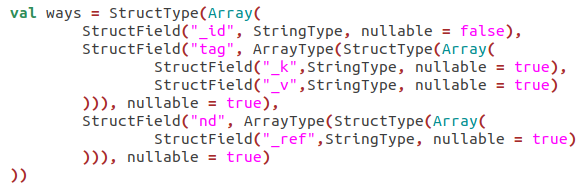
X2 dataframes are created, 1 for keeping track of node id’s and their child tags.



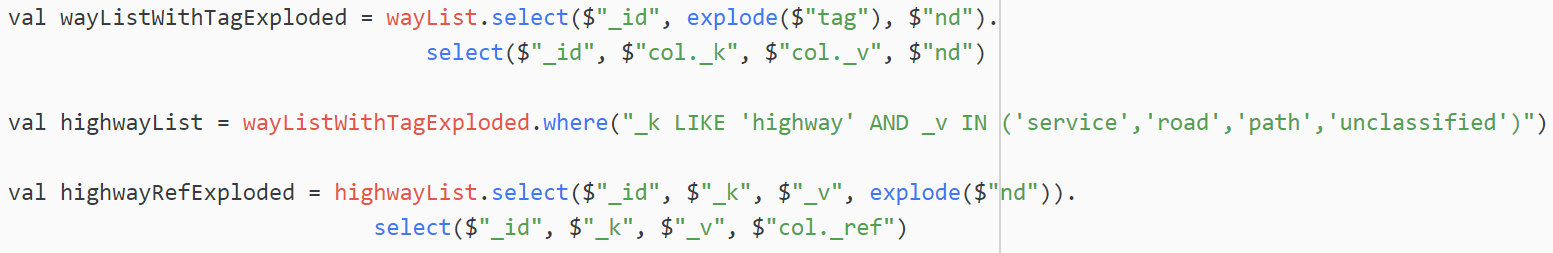
The array(tag) is exploded to filter on (key=barrier, value=lift\_gate).



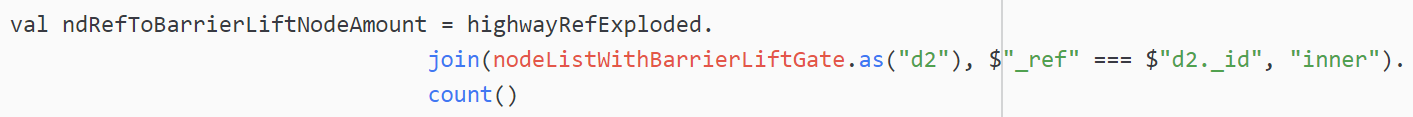
The the other dataframe keeps track of way id’s, their child tags and child nds.



The array(tag) is exploded to filter on (key=highway, value in (path,road,unclassified,service). After filtering out the desired ways the array(nd) is exploded.



An inner join is done on the two dataframes. The join is connected on way.nd\_ref == node.id). Finally, a count() function on the result of the inner join.



Result:



### Task 7 - Which 15 highways contains the most number of trafficcalming=hump?

#### MapReduce

TODO:: code

#### Spark

TODO:: Code explanation

Result: **Note\*\*** my osm file had no occurences of traffic calming hump, so I used bump instead just to get some actual matches. The output is also supposed to show 15 highways but there were less than 15 matches in the datafile which is why only 11 is showing here.

Machine generated alternative text:
_tdl countl 
| 293237541 | 
1 48934531 
11149252761 
1 48934511 
1 201998981 
11149253041 
14842645991 
1 2063917961 
1 294015851 
1 2932375381 
1 294015861 
41 
31 
31 
31 
21 
21 
21 
21 
11 
11 
11 

### Task 8 - Which building has the largest latitudinal extent? ( biggest difference between the northernmost and southernmost node)

#### MapReduce

TODO:: Code

#### Spark

TODO:: Code explanation

Result:

Machine generated alternative text:
_tdl 
diffl 
14911979251 e. 00294113161 

\_id is the way’s id. Diff is the difference between its largest node latitude and smallest node latitude.

On OpenStreetMap we can look at which building the way is using it's id. <https://www.openstreetmap.org/way/491197925> The building is:

Machine generated alternative text:
1064 
Melløs 
Kallum 
Kallum 
118,119 
Melløs 
stadion 
118 
119 
Rygge Storsenter P 
• Høydem 

Which does look like it is the longest latitudinal building in the area.

# 3.3.2 – Creative Part

TODO

# 3.3.3 – Comparing MapReduce & Spark

TODO