Big Data Portfolio Assignment – Part 1

Contents

[Importing data into HDFS 2](#_Toc23710228)

[Updating data in HDFS 3](#_Toc23710229)

[Simple Metrics 4](#_Toc23710230)

[Running the MapReduce program 4](#_Toc23710231)

[Code Components and Concepts Explained 5](#_Toc23710232)

[MapReduce 5](#_Toc23710233)

[Spark 7](#_Toc23710234)

[Simple Metrics Pseudocode 8](#_Toc23710235)

[Task 1 8](#_Toc23710236)

[Task 2 9](#_Toc23710237)

[Task 3 11](#_Toc23710238)

[Task 4 12](#_Toc23710239)

[Task 5 14](#_Toc23710240)

[Task 6 15](#_Toc23710241)

[Task 7 17](#_Toc23710242)

[Task 8 19](#_Toc23710243)

[Creative Part 21](#_Toc23710244)

[Task 1 21](#_Toc23710245)

[Task Description 21](#_Toc23710246)

[MapReduce 21](#_Toc23710247)

[Spark 22](#_Toc23710248)

[Task 2 22](#_Toc23710249)

[Task Description 22](#_Toc23710250)

[MapReduce 22](#_Toc23710251)

[Spark 23](#_Toc23710252)

[Discussing the programs 23](#_Toc23710253)

[Comparing running on single node cluster vs multi node cluster 23](#_Toc23710254)

[What happens behind the scenes? 23](#_Toc23710255)

[Comparing MapReduce & Spark performance 23](#_Toc23710256)

[Sources 25](#_Toc23710257)

# 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).

# Updating data in HDFS

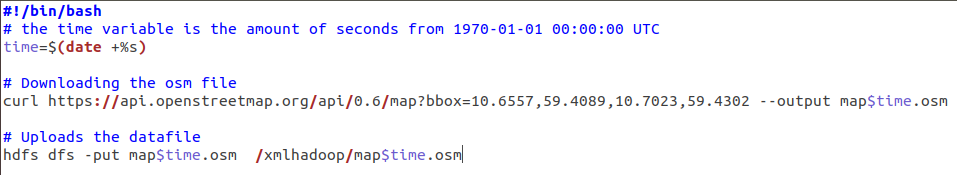
The HDFS does provide a command for overwriting existing files within the hdfs.

  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**

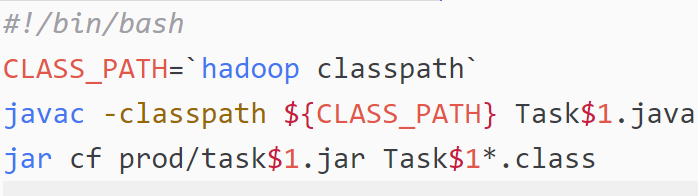
# Simple Metrics

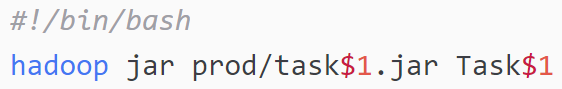
## Running the MapReduce program

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

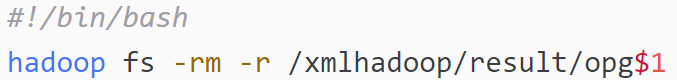
It is therefore a driving factor behind the structure and naming process for the contents of the hdfs.

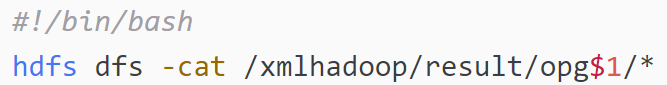
For example, the actual mapreduce programes for 3.3.1 are named as "Task {nr of task it solves}.java" which results in the following Bash scripts being able to compile and run tasks given the task nr as parameter:

**compile.sh**

**run.sh**

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**  


**printResult.sh**

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

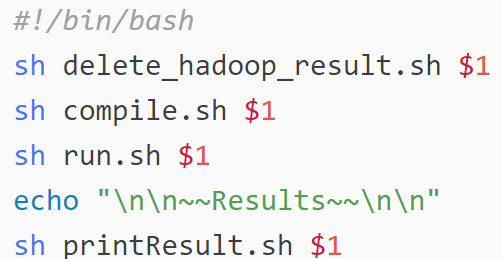


The MapReduce job will check that the specified output directory does not already exist. If it does exist an exception will be thrown, and the directory will not be overwritten.

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:

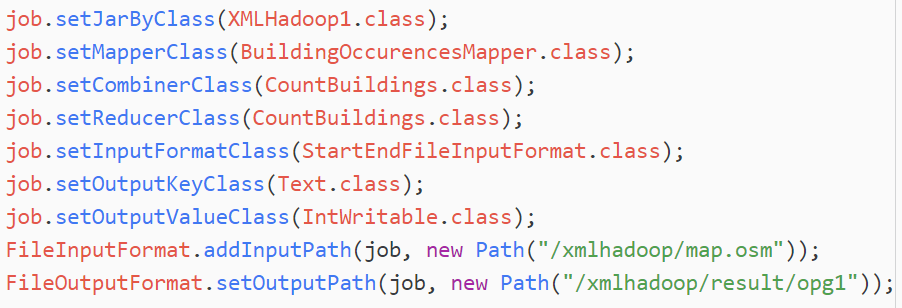
**full.sh**

## Code Components and Concepts Explained

### MapReduce

**Input Format** is an interface that is designed to define a logical split of data that can be distributed to different mappers. The data that gets sent to mappers are called **Input Splits**.   
The method createRecordReader() overrides the method in InputFormat. The method defines a custom RecordReader and calls the RecordReaders initialize() method.

The **RecordReader** is a class that reads the key value pairs from an InputSplit generated by the InputFormat. The task of this class is to convert the byte-output of the InputSplit and converts it into a key value pair more understandable for mappers to process.  
In all my MapReduce solutions I use either the TextInputFormat or a custom FileInputFormat. The TextInputFormat reads the file line-by-line and the InputSplits are <position in file, line content>.  
The Custom FileInputFormat is used when xml-elements are to be passed as InputSplit Values. This is done by defining string values for what to consider as ‘starttag’ and ‘endtag’ in the RecordReader. The RecordReader then reads the file and produces *‘starttag’ + content between + ‘endtag’* as Value for mapper.

The **Configuration** class in MapReduce is used to define some of the necessary settings needed to launch the program.  
The addResource() method is used to load a configuration resource, which in my case I use the config file for the HDFS daemons (hdfs-site.xml). This is so that the client node can know the location of the namenode.  
Other crucial information needed to launch the program is configured on a **Job** object. The Job object implements the interface JobContext, which gives access to all the methods that set configuration information. Here are some of the most common methods:  


The MapReduce framework spawns one map task for each InputSplit generated by the InputFormat. **Mappers** are individual tasks that transforms the InputSplit’s RecordReader output into a different set of output. Firstly, the setup() method is called. Followed by the map() method. The map() method is called for each key/value pair in the input value. That is where the transformation of the data happens. Lastly, the cleanup() method is called.  
Generating the output from the Mapper is done by calling the context’s write() method. If the Job does not have a any reducers, then the output shall be written directly to the OutputFormat. If the Job does have reducers, then the outputs are partitioned per reducer.

The **Partitioner** class is used to control the way the map output keys are distribured to the different reducers. This is done by a hash function. A partitioner is only created when multiple reducers are being used. My MapReduce code does not ever use more than 1 reducer for a job (being a single-node cluster), so a partitioner is never used.

The Reducer has 3 primary phases.   
1. The **Shuffle** stage transfers the map output to a Reducer.  
2. The **Sort** stage does the merging and sorting of map outputs. By default, the key value pairs are sorted on key.  
2.2 – **Secondary Sorting** allows for sorting the value of the key value pair.  
3. **Reduce** – Transforms the content by looping foreach key and all its values.

A **combiner** is essentially a mini reducer that is run on the mapper node after the mapper is finished. Combiners are used to lessen the workload on the reducer. Combiner classes are written the same as Reducer classes. I have implemented Combiners in some of my MapReduce solutions despite this not being necessary due to the fact I mostly only use 1 mapper and 1 reducer for each job.

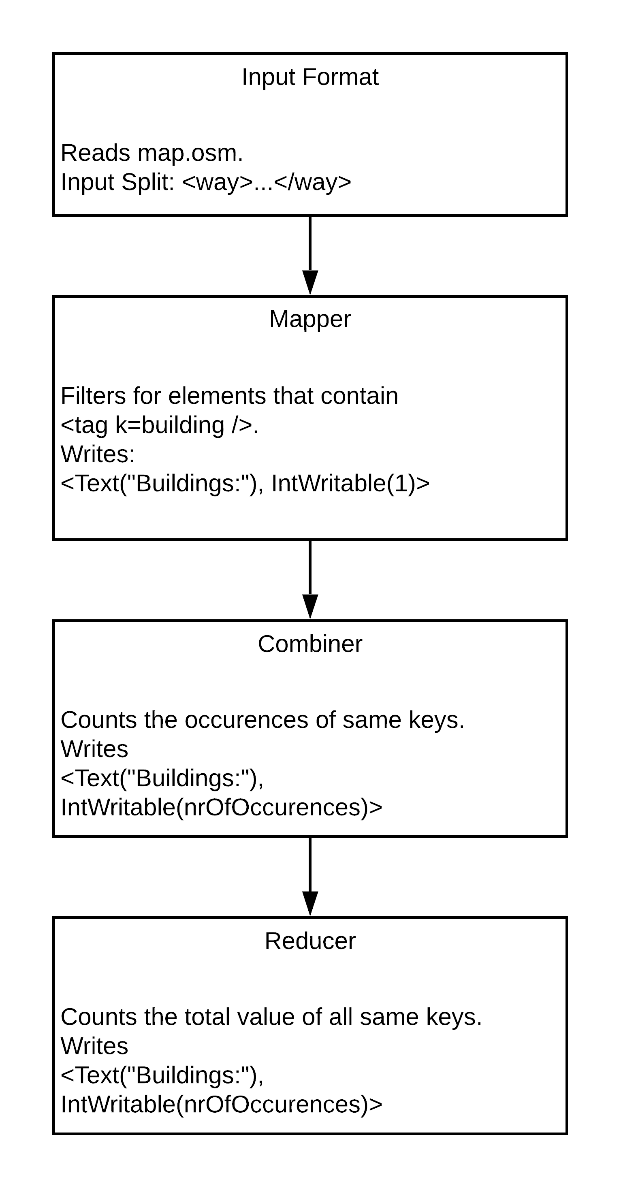
### Spark

TODO RDD vs DataFrame vs DataSet & more

## Simple Metrics Pseudocode

### Task 1

#### MapReduce



InputFormat: InputSplit is set to <way> elements because that is the element containing the relevant data.

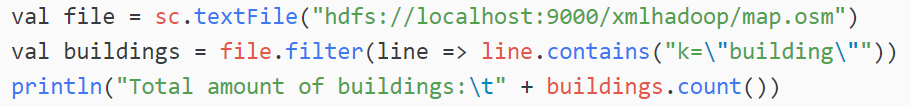
Mapper: Converts the Input Value to Element. Loops all the child <tag>s. If statement that checks if the tag has attribute building, wraps around the code that writes the output.

Combiner: Has a global int variable that initially is set to 0. In the reduce() method the variable adds the Int values. In the cleanup() it finally writes the output.

Reducer: Does the same as the Combiner (they are the same class)

#### 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.

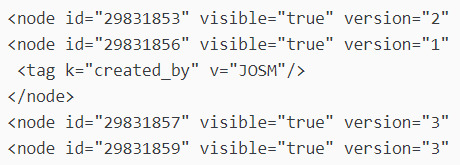


### Task 2

#### MapReduce

A screenshot of a cell phone

Description automatically generated

InputFormat: InputSplit logically would be set to <node></node> elements. However, the way that the OSM file is structured has occurences of this pattern:  
  
That means that the StartEndInputFileFormat example class provided by the course would in occasion produce Input Splits like: “<node .../><node …/><node…></node>” Because those tags don’t share a parent tag it is considered bad XML structure. That causes problems when trying to read and convert the string into Element. Therefore, in the RecordReader when writing the content, it wraps around a custom parent tag <nodes>*content*</nodes>.

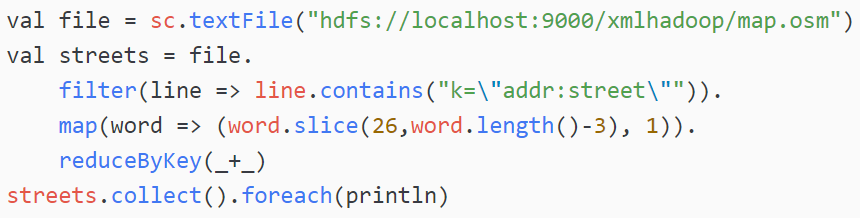
Mapper: Converts the Input Value to Element. Get the last child node (it is the only relevant one as it is the only element that will have any child elements) and loop through its child nodes to check if it has a <tag k=addr:street…/>. If so, it shall output <streetname, 1>

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).

#### 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.



### Task 3

#### MapReduce

A screenshot of a cell phone

Description automatically generated

Mapper: Uses substring() to extract element name, element id and version value. Passes those values into the constructor of custom Writable VersionNr.

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.

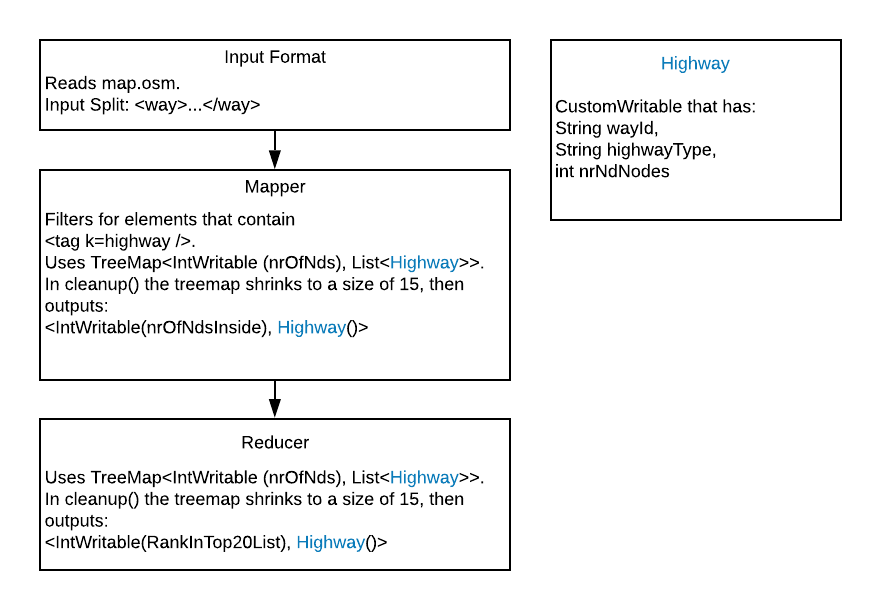
Reducer: Same as Combiner (is same class)

#### 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:



### Task 4



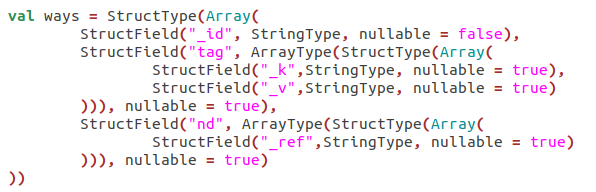
InputFormat: InputSplit is set to <way> elements because that is the element containing the relevant data.

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 the Input Value has a child <tag k=”highway”>  
In the cleanup() method the TreeMap is shrunken to a size of 20. Then proceeds to write those to the output.

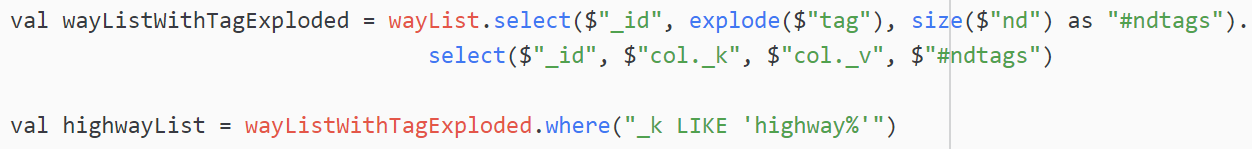
Reducer: The reducer also has a global TreeMap variable <IntWritable, List<Highway>> which is initialized in the setup() method. The reduce() method writes all the values to the treemap. In the cleanup() method the treemap is shrunken to a size of 20 and will be the output result.

#### Spark

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

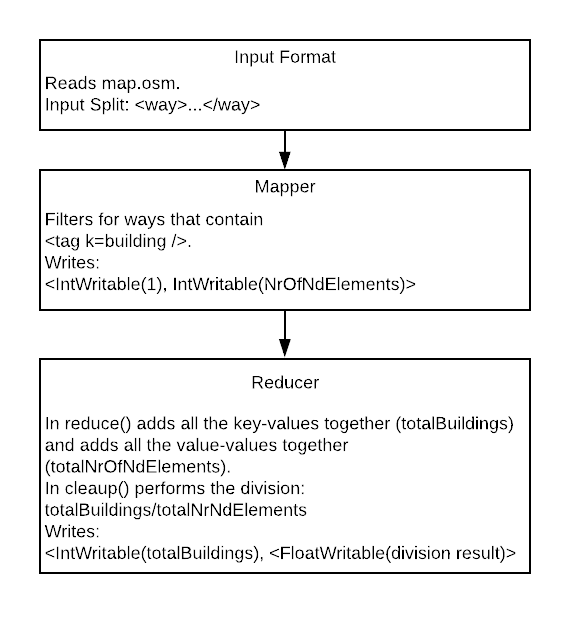


Upon loading the osm file it imports the data into the dataframe. Since there exists several <nd> and <tag> child elements they are created as arrays in the schema, which is why field “tag” and “nd” are of type ArayType. An aggregate function size() is done on the nd array to get the number of nds a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



### Task 5

#### MapReduce



InputFormat: InputSplit is set to <way> elements because that is the element containing the relevant data.

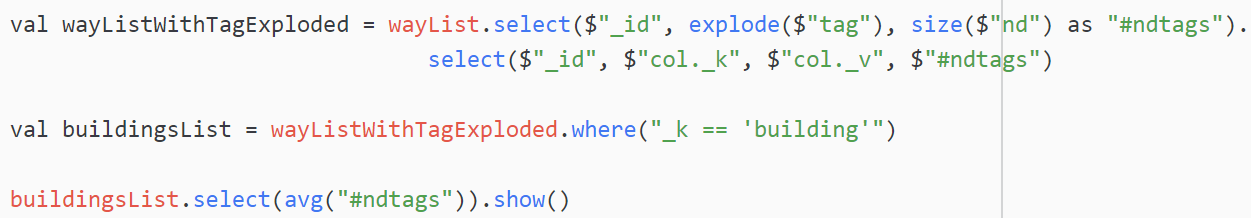
Mapper: In the map() method the Mapper outputs <1, nrOfNdNodes> foreach instance of <tag k=building> in the Input Value.

A conscious decision has been made to avoid using Combiners in this solution. The reason is that calculating averages based on other averages does not give an accurate total average.

Reducer: 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 outputs <totalBuildings, (totalNdsNodes/totalBuildings)>

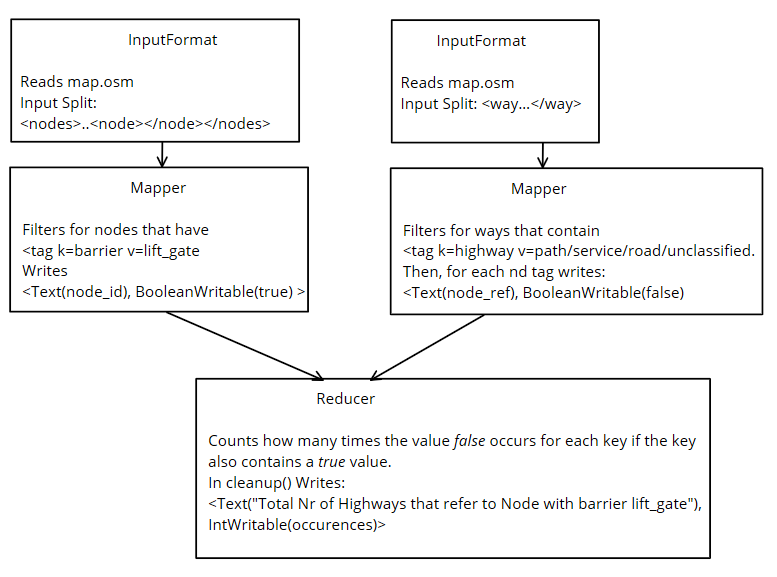
#### Spark

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.



### Task 6

#### MapReduce

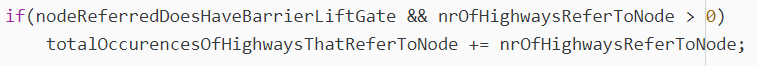


The solution uses a ReduceSideJoin method, which means x2 mappers and x1 reducer. Since two mappers are needed and will read the datafile differently, the MultipleInputs class must be used. A custom FileInputFormat creates Input Splits on <nodes>….<node></node></nodes> which is to be used by the NodeRefMapper. The other custom FileInputFormat creates Input Splits on <way…</way>, which is to be used by the HighwayWithSpecificValuesMapper.

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 that also contains <tag k=highway v=service/path/road/unclassified/>

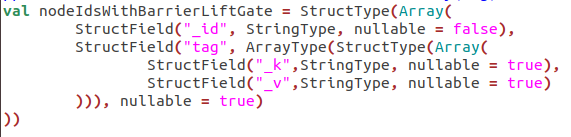
The OutputValue being a BooleanWritable is used to identify wether the map output was from the NodeRefMapper or the HighwayWithSpecificValuesMapper. (true means it was the output from NodeRefMapper, false means it was the output from HighwayWithSpecificValuesMapper).

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.   


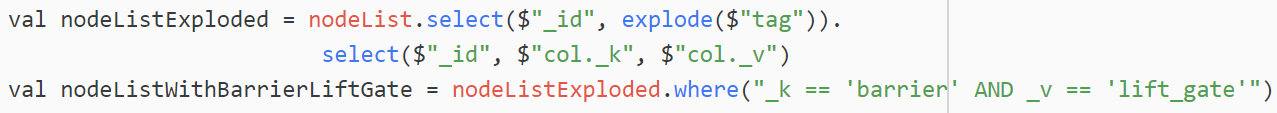
In the cleanup() method it shall output < "Total Nr of Highways that refer to Node with barrier lift\_gate", totalOccurencesOfHighwaysThatReferToNode>

#### Spark

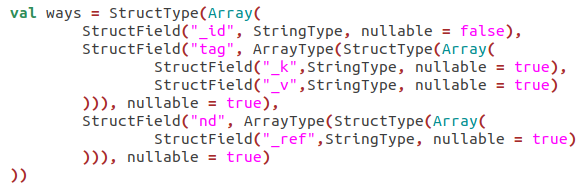
X2 dataframes are created, 1 for keeping track of node id’s and their child tags.



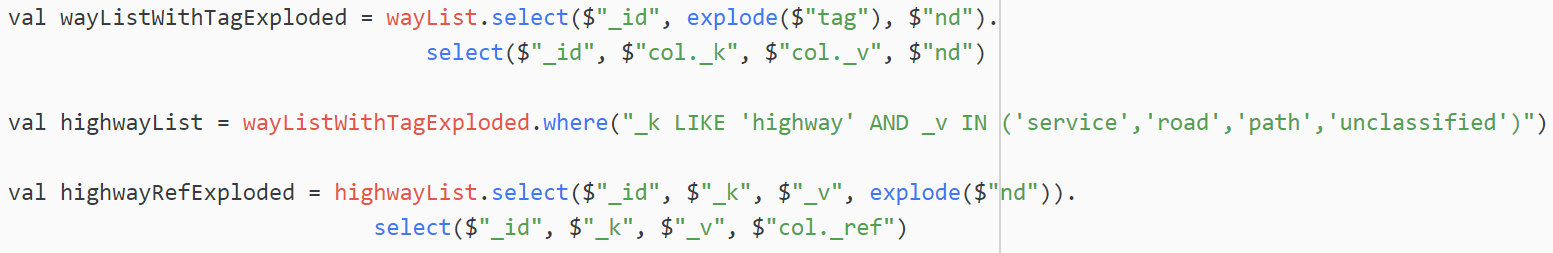
The array(tag) is exploded to filter on (key=barrier, value=lift\_gate).



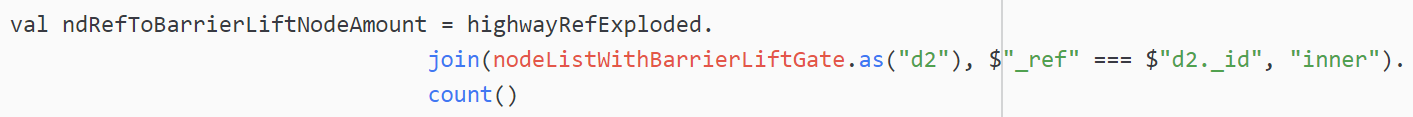
The the other dataframe keeps track of way id’s, their child tags and child nds.



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.

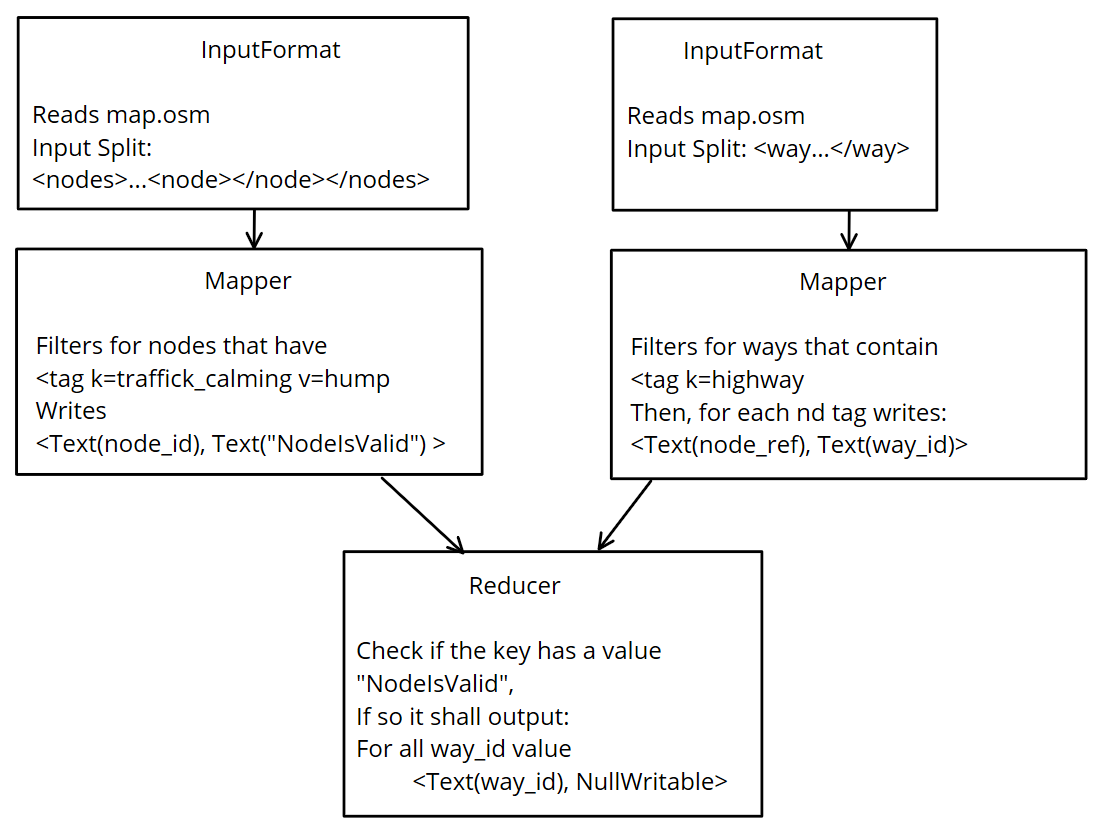


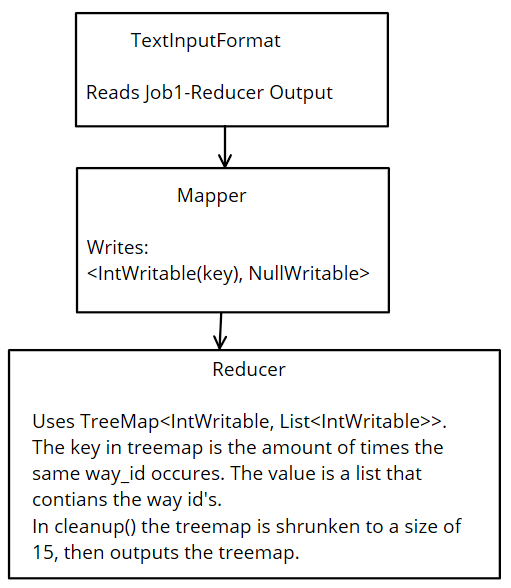
An inner join is done on the two dataframes. The join is connected on way.nd\_ref == node.id). Finally, a count() function on the result of the inner join.



### Task 7

#### MapReduce

Job 1:  


Job2:  


In Job1 the aim is to perform a reduce-side-join to combine the way nd’s to the corresponding nodes. Two Mappers are used, HighwaysMapper and NodeRefMapper.

HighwaysMapper: Recieves Input Splits values as <way> tags. If the <way> contains a <tag k=highway…/> then it shall output all the nd ref attributes as key and way id as value.

NodeRefMapper: Recieves Input Splits values as <nodes><node…></nodes>. Filters for nodes that have a child tag <tag k=traffic\_calming v=hump/>. Outputs node id as key and “NodeIsValid” as value.

Reducer: In the reduce() method has an arraylist that shall contain the way id’s that refer to the key (node id/ref). If the node being referred toexists then the arraylist shall be output as key. The value is NullWritable for efficiency.

The second jobs aim is to essentially be a top 15 word count on the result from job 1.

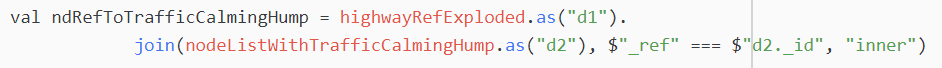
#### Spark

X2 Dataframes are loaded in. 1 is for keeping track of nodes that have trafficcalming hump. Filtering out the nodes with tag trafficcalming hump is done by exploding the tag array and using the where function to get k=trafficcalming & v=hump.

The other dataframe is for all the ways with tag highway.

Again, filtering out for tag k=highway is done by exploding the tag array and using where(). Next, the nd\_ref array is exploded, in preparation for the join operation.

An inner join is done on the two dataframes by linking the node\_id == node\_ref. An inner join means that the query will only get the records where the two dataframes have overlapping matches.

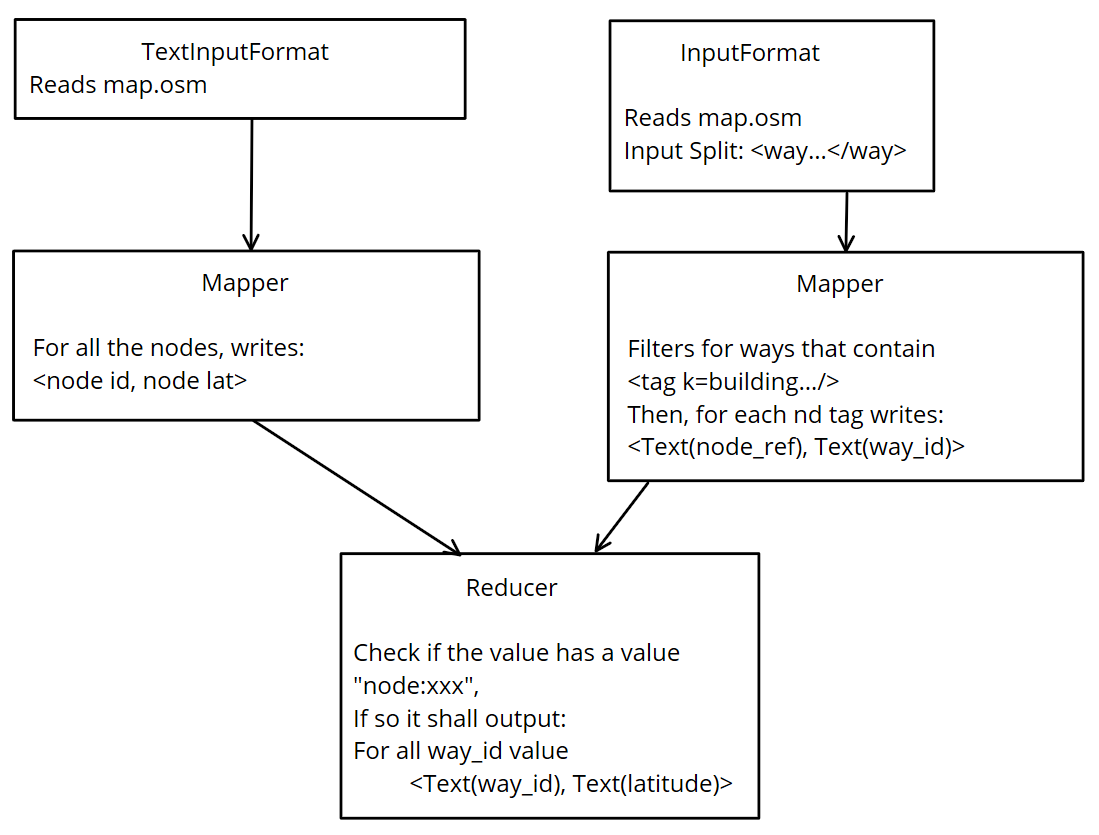


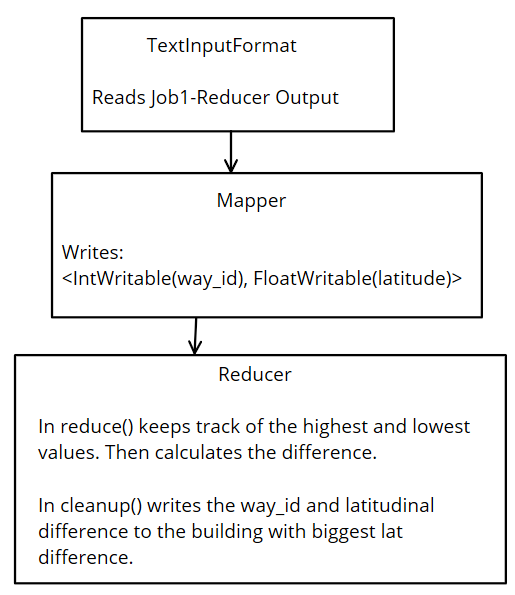
A count() function is done grouped on the way\_id to get the number of rows per way\_id. Finally, sorting on the ‘count’ from biggest to smallest and showing the top 15 is done by:



### Task 8

#### MapReduce

Job1:  


Job2:  


In Job 1 there are two mappers and one reducer, in order to achieve a reduce side join. The two mappers are NodeLatMapper and BuildingsMapper.

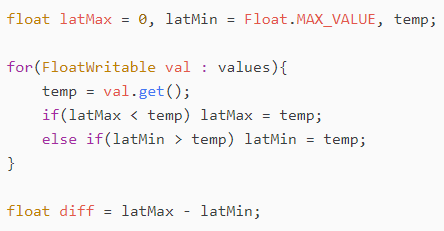
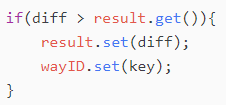
NoteLatMapper: The input values are lines from the datafile due to the input format being TextInputFormat. So, in the map() method, filters for lines that are <node …>.   
Proceeds to extract the node id and node lat attributes by using substring(), and outputs <node id, “node:node lat”>.

BuildingsMapper: The input values are <way> tags. In the map() method it filters for <way>s that contain <tag k=building…/>. If so, the it shall output for every nd child tag: <node ref, “way:wayid”>.

WayIdNodeRefReducer: In the reduce() method has an arraylist that shall contain the way id’s that refer to the key (node id/ref). If the node being referred to exists, then the arraylist shall be output as key. The value is the node’s latitudinal value.

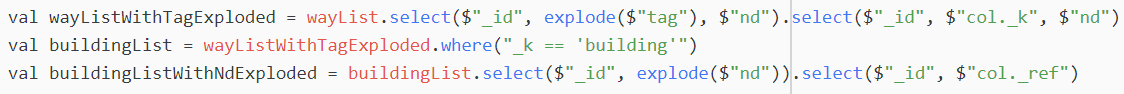
The aim of Job 2 is to get all the latitudinal values for each way, then get the biggest latitudinal difference per way, and output the way with largest latitudinal difference of all.

WayMapper: The input splits are the values from the reducer of the first job. They were formatted like this:  
wayId (\t) latitudinalValue  
The Mapper does no filtering, simply outputs <wayId, latValue>.

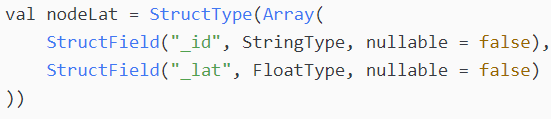
FinalReducer: This Reducer is used to calculate the biggest latitudinal difference for each way (key). The code that achieves this is in the reduce() method:  
  
A FloatWritable ‘result’ holds the biggest diff value. A Text variable ‘wayID’ is the wayID the result diff belongs to. After calculating the diff for the way in the reduce method(), it compares the diff value to the ‘result’ value. If the diff value is bigger, it shall update the wayID and result variables in order to keep track of the information that has the biggest lat diff.  
  
The output is written in the cleanup() so only 1 (the highest) value is outputted.

#### Spark

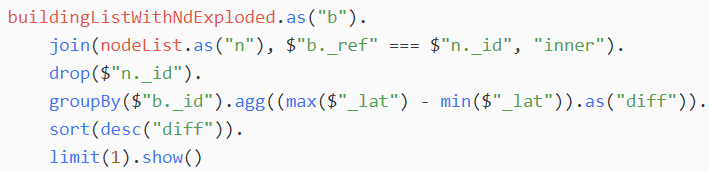
The solution requires x2 different dataframes. One of them is for keeping track of the buildings within <way> and has building\_id, array(tag), array(nd). The array(tag) is used to explode and filter on key=building. Once the filtering is done the array(nd) is exploded to prepare for the join with the other dataframe.



The other dataframe is for keeping track of the node id’s and their latitudinal value.



An inner join is done on the two dataframes connecting on the   
building.nd\_ref == node.id. From there, I want to get the difference between the maximum lat value and minimum lat value foreach building\_id.   
This done in code looks like:



# Creative Part

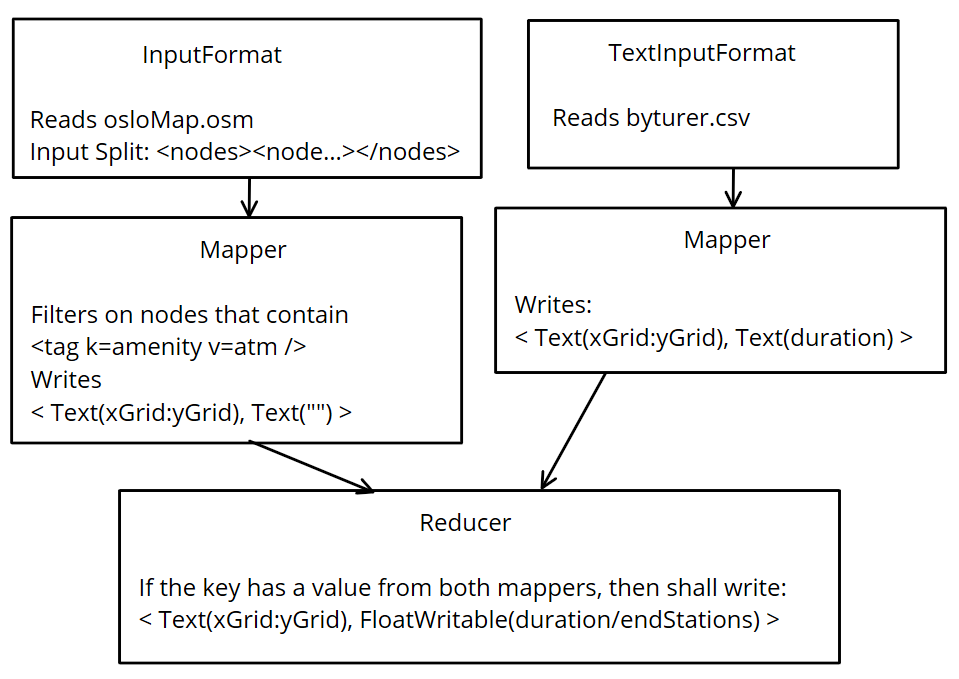
The data I am using for the creative part is open street map with bounding box: 10.662, 59.898, 10.850, 59.957.  
The other datafile is the data on oslo city bike trips <https://oslobysykkel.no/apne-data/historisk>.

## Task 1

### Task Description

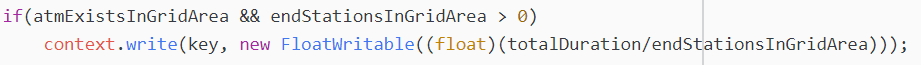
In a grid map of 1km what is the average duration of trips that ended in the grid area where the grid area also contains an atm machine?

### MapReduce



GetGridOfAtms: Filters for <node>s that contain child tag <tag k=amenity v=atm />. If so, then gets the <node> lat and lon attributes. Calculates the x and y distance from a specific point (10.662, 59.898) and divides by 1000 to get 1km grid area. Uses the result as key. The value is an empty Text.

GetGridOfEndedTrip: Gets the trips end station lat and lon values. Then calculates the x and y distance using the **same** specific point (10.662, 59.898) and again dividing by 1000 to get 1km grid area. The value is the trips duration.

GetAverageDurationReducer: Checks if the key has an empty value which would indicate that the grid area does have an atm machine. Also counts the number of trips that have ended in the grid area aswell as calculating the total duration of all trips ended there.  


### Spark

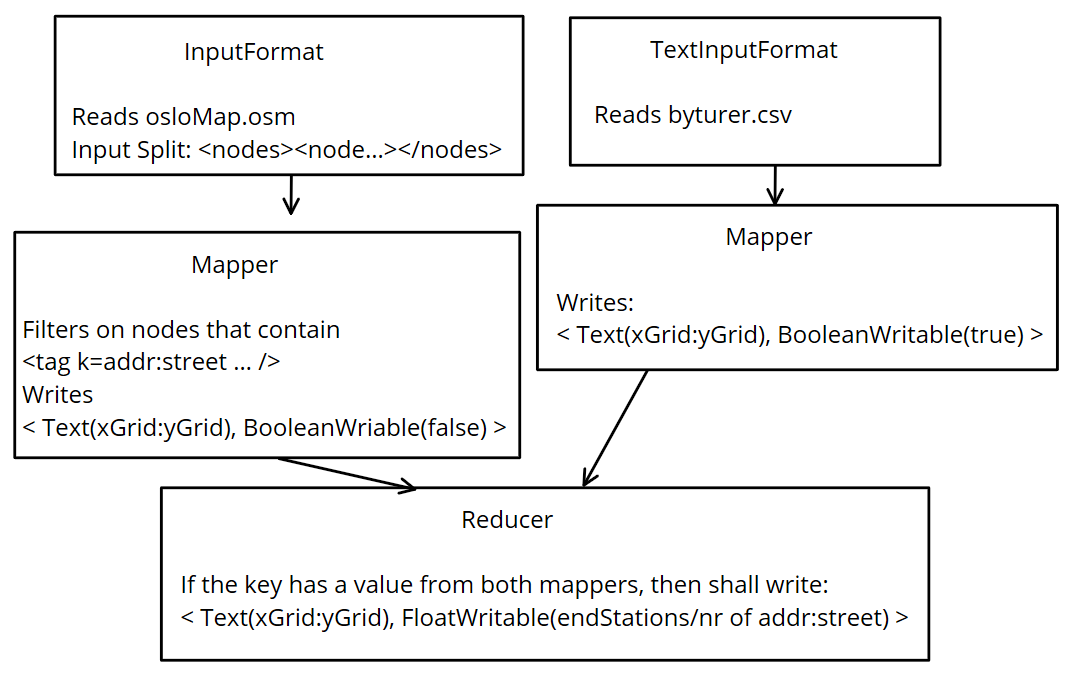
TODO

## Task 2

### Task Description

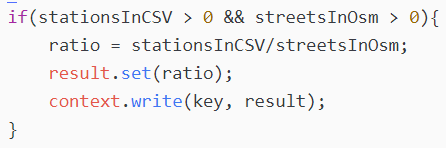
In a grid map of 1 km, what is the ratio of how many bike trips have started there divided by how many addr:street tags there are there?

### MapReduce



CSVReaderMapper: Calculates the x and y distance between specific point (10.662, 59.898) the same way as Creative Task 1, but with start station instead of end station. As value simply writes BooleanWritable(true).

OsloOsmMapper: Filters for nodes that contain <tag k=addr:street…/>. For those nodes, it calculates the x and y distance same as in Creative Task 1. As value writes BooleanWritable(false).

CountOccurences (Reducer): The reason the mappers had BooleanWritable as values is to help the reducer differentiate which mapper the value came from.   
True = CSVReader, false = OsloOsmMapper.  
Counts the number of start stations and streets in the grid area. After getting all the values for the key writes:  


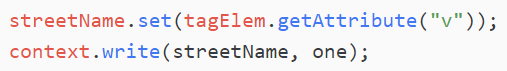
### Spark

TODO

# Discussing the programs

## Limitations and bugs

A bug in the mapreduce programs was that the MultipleInputs could not launch more than 1 map task if the datafiles were the same. The work around for this was to simply have duplicate datafiles in the HDFS.

In many of the mapreduce programs Writables are declared as member variables (inside the class outside of any fuction). If the value of the Writables must be changed when outputting results, .set() methods are used. The reason for utilizing the .set() method instead of initializing a new Writable object everytime is because it is faster. However, this can cause an inaccuracy problem:  
  
The map() and reduce() methods run simultaneously for each key/value pair. A problem that can arrise is that between the time it takes for a key/value pair in the map/reduce method to set the writable value and output the writable, the writable value could have been overridden again by another key/value pair. That would result in the wrong output for the kay/value pair.

## Comparing running on single node cluster vs multi node cluster

Despite the mapreduce programs performing as they should on the single node cluster, things would differ greatly when running the same programs on a multi-node cluster. This is because in the mapreduce programs, only 1 reducer is used per job and 1 mapper foreach type of mapper. For example, in simple metrics task 3 (*Which object in the extract has been updated the most times, and what object is that?)* where the result is supposed to be 1 value, using 1 reducer will give the exact answer the question is asking for. However, using x2 reducers (or more), will result in more than 1 answer – which is not accurately answering the question posed. This is because each reducer would produce an output. In order to get 1 reducer output/answer in a multi node cluster, a solution would be to run another job on the previous results and utilizing shuffle-sort to smartly distribute the key/values to the reducers. For example, using shuffle-sort to output the results from the mapper to the reducer by descending key-value will mean that reducer nr 1 would have the biggest key-values. Therefore, we would know that the task 3 answer would lie in the part-r-00000 file of the second job.

Choosing number of reducers.

## What happens behind the scenes?

TODO:: See lecture 15 notes – UDF & YARN

# Comparing MapReduce & Spark performance

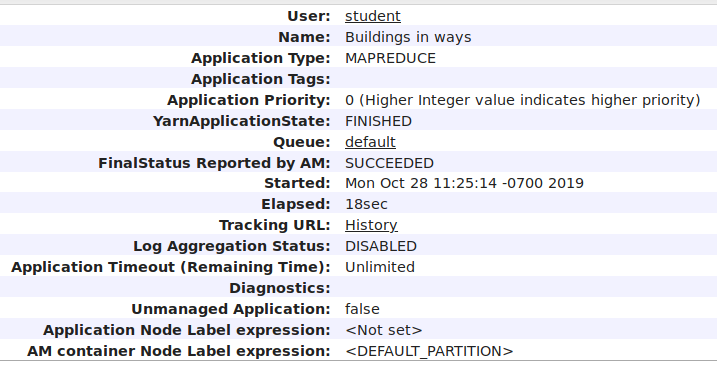
MapReduce is written in Java, and Spark is written in Scala. Both the languages are object-oriented, but Scala is also a functional programming language. Some differences between the two languages:

|  |  |
| --- | --- |
| Java | Scala |
| Huge chunks of code | More compact code |
| Variables are by default mutable | Variables are by default immutable |
| More readable code | Less readable code |
| A lot of documentation and community help | Does not have as big of a community so less help |
| Does not offer first-class functions | Offers first-class functions (meaning a function can be treated as a variable) |

Writing Java compared to Scala was a lot more time consuming as for example task 1 was solved in ~200 lines of Java code and 3 lines of Scala code.

A big difference is noticed when comparing the performances of the MapReduce implementations and the Spark implementations. MapReduce takes a lot longer because MapReduce is strictly disk-based. Spark uses memory and can use disk for processing. A claim from Spark is that Spark can run up to 100x faster than MapReduce in memory, and up to 10x faster on disk. The reasoning for this is that Spark does not do the Input/Output operations with the disks that MapReduce does. Instead it writes to memory. For that reason, Spark is the better option compared to MapReduce if data processing needs to be done in real-time.

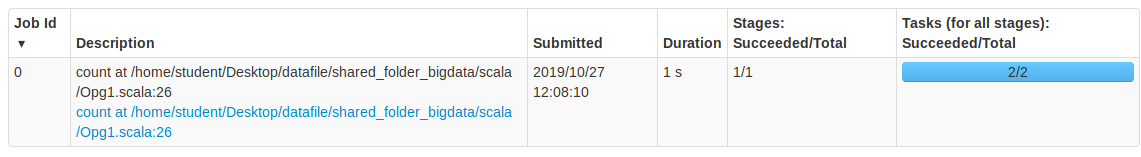
Hadoop has a Web User Interface for showing the status of cluster, job details and more (port 8088). I ran XMLHadoop1.java as a job example for analyzing through the web UI. Navigating to ‘All Applications’ => ‘application\_id’ gives detailed information for the application:

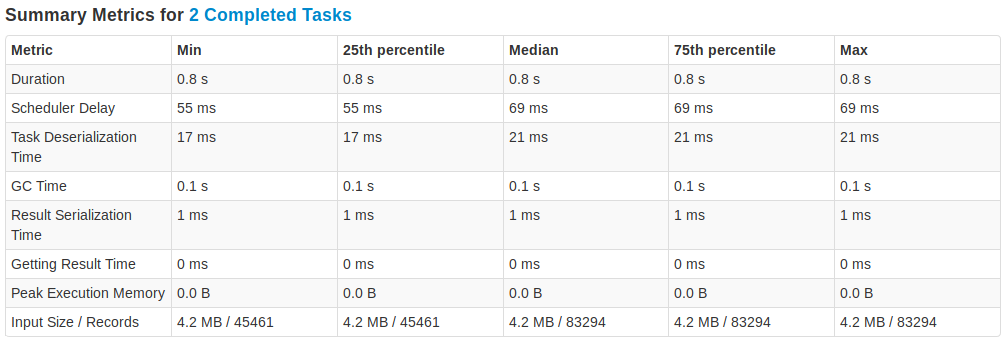


Here we can see that the elapsed time is 18 seconds.

The first time I ran and analyzed the XMLHadoop1.java job, it used 38 seconds to complete. The reason it was taking double the amount of time to complete was due to new objects being created upon writing to the map and reducer outputs. Reducing the number of objects created allows the code to run faster. Achieving this was done by having 1 Key Writable and 1 Value Writable declared as global varibles in the Map and Reduce classes. Calling .set() to alter the Writable value before writing to output instead of declaring new Writables each time had a significant effect on the time consumption. However, using .set() to change Writable values instead of creating new instances of type Writable can have a negative effect on the accuracy of the data processing (explained deeper at point XXXX).

Every Spark Contect launches a Web User Interface (by default on port 4040). It shows useful information such as stages and tasks, RDD memory usage and information on running executors.  
I run Opg1.scala as an example to analyze the information the Web UI can tell us. Navigating to “Jobs” gives an overview of all jobs in the Spark Context.

  
Here we can see that the job took only 1 second to complete. Following the ‘Job Description’ link gives us a more detailed look into the specific job. Here we can get a summary of metrics:



‘Summary of metrics’ is a useful tool for analyzing the specifics for what is slowing the tasks down.

# Sources

Importing Data into HDFS:  
<https://www.quora.com/What-is-the-difference-between-single-node-hadoop-and-multi-node-hadoop-cluster>   
<https://beyondcorner.com/learn-apache-hadoop/hadoop-single-node-multi-node-cluster/>  
<https://www.dezyre.com/article/hadoop-cluster-overview-what-it-is-and-how-to-setup-one/356>

Replacement Policy:

<https://knpcode.com/hadoop/hdfs/hdfs-replica-placement-policy/>

Explaining what happends in the background:  
Hadoop Documentation.

Comparing MapReduce and Spark:  
<https://www.guru99.com/scala-vs-java.html>  
<https://www.educba.com/mapreduce-vs-apache-spark/>