

7CCSMBDT Big Data Technologies

Coursework 1

Due: Monday 1 March 2021 (23:59 UK time)

TASK 1: BIG DATA COLLECTION USING APACHE SQOOP:

Describe three features of Apache Sqoop that help import data into a distributed file system efficiently. Your description should state the feature and a brief justification of what the feature does and how it can help efficiency.

I would like to start talking about the different types of tools and frameworks we have at hand for data collection, where the data is collected and ingested into a big data stack. There are 5 main data access connectors types. Which one to use depends primarily on the type of data you are willing to ingest, in other words, its architecture. These types are the following:

1. Publish-subscribe messaging: the **publishers send** messages (that contain data and information) to a topic managed by a broker (intermediary). The **subscribers subscribe to topics** they want to receive information about. Lastly, the **brokers simply route those messages** from one side to the other. Example of tools that work like this: Amazon Kinesis or Apache Kafka.
2. Source-sink connectors: the **source connectors import data** from another system, like a RDBMS, into a centralised data store, like HDFS. On the other hand, the **sink connectors export data** to another system. A good tool to manage this type is Apache Flume.
3. Database Connectors: they **import data from a RDBMS**, like MySQL, **into a big storage** and analytics **frameworks**, like HDFS. The main tool that works with this type is **Apache Sqoop**, which we will cover later.
4. Messaging Queues: the **producers push data to the queues** (the channel that sends the data, a kind of storage space for the data) and the **consumers pull that data from the queues**. Similar to the publish-subscribe messaging, but without the idea of a broker or a topic, here the producers and the consumers do not need to be aware of each other and the messages are not organised by topics.
5. Custom connectors: used to **collect data from specific sources**, like social networks (Twitter), NoSQL databases or Internet-of-Things. These are kind of built ad-hoc on the data sources and data collection requirements.

More to the point, we have to describe three features of Apache Sqoop, a Database Connector for collecting data. This framework can import data from a RDBMS to HDFS (Hadoop Distributed File System), or export data from Hadoop to a RDBMS. It can also interact with other Hadoop related systems like HBase or Hive. So, in short, it is a command-line interface application for transferring information. The three main features that efficiently help import data are the following:

- a. **Parallelism**: key benefit of Apache Sqoop. This is done by MapReduce jobs, which allow the data to transfer much faster (in parallel). The more mappers you specify when calling the import command, the faster the data will be transferred to HDFS.

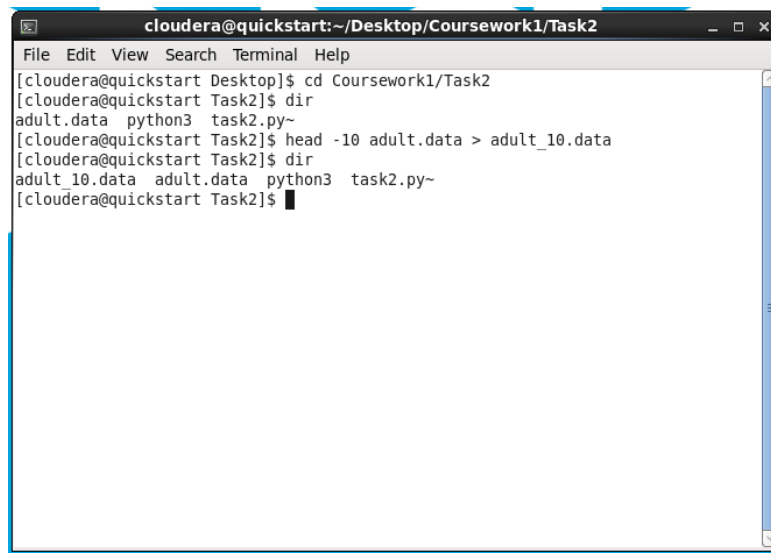
Anyways, there is a trade-off between number of mappers and the number of concurrent queries to the relational database. MapReduce is a programming model that performs large-scale, distributed computations efficiently, and as a benefit, it also tolerates hardware failures.

- b. **--incremental append command:** this is one of the parameters you can specify in the import command. Its benefit is efficiency, since Apache Sqoop does not have to upload all the data again, it simply updates since a specific parameter (*check-column* and *last-value*). In case you want to avoid updating existing rows unintentionally, you can use another parameter called *incremental lastmodified* that identifies the records that have changed based on the time (here *las-value* is a date).
- c. **--direct command:** this command tells Apache Sqoop to use a specific connector for the database at use. In the case of MySQL, mysqldump will be used for retrieving data from the database server, which is sometimes much more efficient and faster.

TASK 2: MapReduce FREQUENCY-BASED COMPUTATIONS:

Write a program `task2.py` using `mrjob`, which outputs the 10 most frequent values in the Age attribute of the Adult dataset and their frequency. That is, the ten values in Age that appear the largest number of times in the dataset and the number of times each of these values appears.

For this task of the coursework, I have first downloaded the *adult.data* document directly from the Machine Learning Repository. After storing it in a new folder called Task2, which is stored in Coursework1, I have also created another file with the 10 first entries of this data set for testing purposes. This is done with the command `head -n file>new_file`:



```
cloudera@quickstart:~/Desktop/Coursework1/Task2
File Edit View Search Terminal Help
[cloudera@quickstart Desktop]$ cd Coursework1/Task2
[cloudera@quickstart Task2]$ dir
adult.data  python3  task2.py~
[cloudera@quickstart Task2]$ head -10 adult.data > adult_10.data
[cloudera@quickstart Task2]$ dir
adult_10.data  adult.data  python3  task2.py~
[cloudera@quickstart Task2]$
```

As we can see, first there was just a document called *adult.data*, and after running the command specified before, we have another new file called *adult_10.data* that contains the first 10 (head) entries of the original data set.

Now, we have to create a python file that does all the job we are required to do. For that, we first need to import the modules we need. These modules are the following:

- *MRJob*: which allows us to perform MapReduce tasks in Python2.6 or more, and Python3.3 or more. As we are performing the code in Python3.6, it will serve us.
- *MrStep*: which allows us to set the steps we want our program to follow to perform the MapReduce tasks. This is necessary because, by default, *mrjob* uses a single MapReduce task to execute the program. As we will be defining to MapReduce jobs, we will need this module.
- *re*: to read the data in the dataset we will use. RE stands for Regular Expressions or RegEx in Python.

```
# We will start by importing the libraries we will need.
from mrjob.job import MRJob # this module let us write MapReduce jobs in Python 2.6+/3.3+ and run them on several platforms.
from mrjob.step import MRStep # this module allows us to set the steps our program is going to make. Which mapper/combiner/reducer goes before or after
import re # this one is for Regular expressions (RegEx) in Python.
```

The next part of code is to read the information in *adult.data*:

```
WORD_RE = re.compile(r"[\w']+") # this command is to read the document/file we are wishing to apply the MapReduce task on.
```

The remaining part of the code is where the 'magic' happens. We will define a class, which we will call *agecount*. In this class we will also define the requested mappers, combiners and reducers we will use to perform the MapReduce tasks.

```
# Now we create the class that will do all the job we want.  
class agecount(MRJob):
```

The first function we will define will be called *mapper*. This class will receive as inputs the lines of the file we are applying this class to (*adult.data*) and will output the age value and a 1, representing it appears 1 time. This is the code we applied to define this function:

```
# We will define first the mapper. This mapper will accept as parameters an empty key ( ) and a line as values. It will  
# output a number of key-value pairs (age, 1)  
def mapper(self, _, line):  
    # This command will split the lines of adult.data and store them in a list called 'data'.  
    data = line.split(", ") # Now, the age will be stored in data[0], as it was in the first column  
  
    age = data[0].strip() # We store the age value of data in the variable 'age'  
    # .split() just removes spaces at the beginning and at the end of the string  
  
    # 'yield' is similar to 'return', but it does not terminate the function. It produces key-value pairs.  
    # In this case, we will yield each age in each line  
    yield age, 1
```

We pass the lines of *adult.data* to the mapper. After that, as the mapper is only interested in the age value, he stays with that value with the two first commands of this function. In data we have a list of values with the different information in the file. As *adult.data* is organised in a way that the attribute age is in the first position of each line, we will stay with that value and store it in age by simply indexing and stripping data.

The last part of the function is the one that return the ages and a 1, as it only maps the information. *yield* is similar to *return*, but it does not terminate the function; instead, it produces a key-value pair. So, if we apply this function to *adult_10.data*, the input and the output are the following:

MAPPER INPUT:

```
39, State-gov, 77516, Bachelors, 13, Never-married, Adm-clerical, Not-in-family, White, Male, 2174, 0, 40, United-States, <=50K  
50, Self-emp-not-inc, 83311, Bachelors, 13, Married-civ-spouse, Exec-managerial, Husband, White, Male, 0, 0, 13, United-States, <=50K  
38, Private, 215646, HS-grad, 9, Divorced, Handlers-cleaners, Not-in-family, White, Male, 0, 0, 40, United-States, <=50K  
53, Private, 234721, 11th, 7, Married-civ-spouse, Handlers-cleaners, Husband, Black, Male, 0, 0, 40, United-States, <=50K  
28, Private, 338409, Bachelors, 13, Married-civ-spouse, Prof-specialty, Wife, Black, Female, 0, 0, 40, Cuba, <=50K  
37, Private, 284582, Masters, 14, Married-civ-spouse, Exec-managerial, Wife, White, Female, 0, 0, 40, United-States, <=50K  
49, Private, 160187, 9th, 5, Married-spouse-absent, Other-service, Not-in-family, Black, Female, 0, 0, 16, Jamaica, <=50K  
52, Self-emp-not-inc, 209642, HS-grad, 9, Married-civ-spouse, Exec-managerial, Husband, White, Male, 0, 0, 45, United-States, >50K  
31, Private, 45781, Masters, 14, Never-married, Prof-specialty, Not-in-family, White, Female, 14084, 0, 50, United-States, >50K  
42, Private, 159449, Bachelors, 13, Married-civ-spouse, Exec-managerial, Husband, White, Male, 5178, 0, 40, United-States, >50K
```

MAPPER OUTPUT:

```
"49"    1  
"52"    1  
"31"    1  
"53"    1  
"28"    1  
"37"    1  
"42"    1  
"39"    1  
"50"    1  
"38"    1
```

Now we will define a combiner by creating another function which we will call *combiner*. A combiner takes a key and a subset of the values for that key as inputs and returns zero or more (key, value) pairs. Combiners are optimisations that run immediately after each mapper and can be used to decrease total data transfer. This combiner is set in the coursework task mainly for optimisations issues. It is more efficient to use on that to not use it.

I found about this in a repository in GitHub which is recommended in the week 4 module slides: <https://github.com/mattwg/mrjob-examples>; and understood the concept of combiners in a StackOverflow entry: <https://mrjob.readthedocs.io/en/latest/guides/concepts.html>.

After all, the combiner does a similar job as the reducer, but it increases efficiency. So, I will show the code of both the combiner and the reducer as the input and output of these two functions is the same. First, I will show the code of both of them:

```
# Here we will define a combiner. A combiner takes a key and a subset of the values for that key as inputs and
# returns zero or more (key, value) pairs. Combiners are optimisations that run immediately after each mapper and
# can be used to decrease total data transfer. This is set mainly for optimization, it makes the MapReduce job
# much faster.
# https://mrjob.readthedocs.io/en/latest/guides/concepts.html#:-:text=A%20combiner%20takes%20a%20key,to%20decrease%20total%20data%20transfer.
def combiner(self, age, count_of_ages):

    # We yield the age as a key and the sum of the values of each age as the value
    yield age, sum(count_of_ages)

# Now we define the first reducer. This reducer accepts as parameters the key (age) and value (list of values) from the
# mapper and combiner defined above, and yields (returns) the sum of the keys we have passed. In other words, it yields
# the number of times each age appears.
def reducer1(self, age, count_of_ages):

    # We yield the sum of the values of each age. This time we will not return any key. Now we will return a value that
    # is actually a pair of values, the number of appearances of each age and the actual age value.
    yield None, (sum(count_of_ages), age)
```

As we can see both of them do similar things. However, they yield different outputs. The first one, the combiner, yields the age value and the total number of times that age value appears in the *adult.data* file. On the other hand, the reducer (*reducer1*) does not yield a key; instead, it yield only a value that is actually a pair of things: the total number of times each age appears (*sum(count_of_ages)*) and the actual age.

So, the input the input the combiner receives is the output of the mapper. Now, the output of the combiner for the 10 first lines of *adult.data* is the following:

COMBINER INPUT: MAPPER OUTPUT

COMBINER OUTPUT:

```
"31" 1
"49" 1
"52" 1
"28" 1
"37" 1
"53" 1
"42" 1
"38" 1
"39" 1
"50" 1
```

This output will be the one received by the first reducer (*reducer1*), the output of this reducer will be looked just like this, but it will not be the same. While the combiner outputs a key-value ([key, value]) pair which looks like this: [age, total times each age appears]. The reducer will output an empty key and a list of pairs of values, which actually is a tuple. So, it will output something that looks like this: [, (total times each age appears, age)].

So, technically:

REDUCER1 INPUT: COMBINER OUTPUT

REDUCER1 OUTPUT: REDUCER INPUT. It looks just like that in the command, but as we explained in the last paragraph, it is actually different.

This brings us to the last reducer of this second task. We now have a list of tuples that contains the total times of appearances of each age, and the age value. We have passed this as a value because we want to output both of them. That explains why our second reducer (*reducer2*) receives an empty key and a list of tuples just like the one outputted by the first reducer.

I will now leave the code below this paragraph to explain it just after:

```
# Now we define the second reducer, the one that will yield/output the 10 most frequent values of 'age'. This reducer
# will accept as parameters an empty key (.) and the value from the reducer1, a tuple (sum_of_ages, age_value), and it will output
# the 10 most frequent age values alongside with the actual age value.
def reducer2(self, _, sum_of_ages_pairs):

    # One of the characteristics of a generator is that once exhausted, you cannot reuse it
    # https://stackoverflow.com/questions/46991616/yield-both-max-and-min-in-a-single-mapreduce

    # So we will create a list instead of a generator, but this list will have the same content as the generator we
    # have passed as a parameter. As a tuple cannot be sorted, we are basically converting it into a list so it can
    # be sorted.
    sum_of_ages_pairs = list(sum_of_ages_pairs) # we create the list of the generator
    sum_of_ages_pairs.sort(reverse=True) # we sort the list of values from the most frequent to the least frequent this is done
    # with the parameter reverse=True

    count = 0 # we initialise this variable to 0 so it serves us as a counter, so we can later exit the for loop when
    # we have already outputted the 10 most frequent values

    # We create a for loop so it iterates through the 10 first values of the sorted list we created above
    for age in sum_of_ages_pairs: # iterate through all values in our list
        if count == 10:
            break # whenever our counter reaches 10, we exit the loop as we already outputted what we wanted
        yield age # here we yield/return/output one value per iteration
        count += 1 # increase the counter after outputting the top-10 values
```

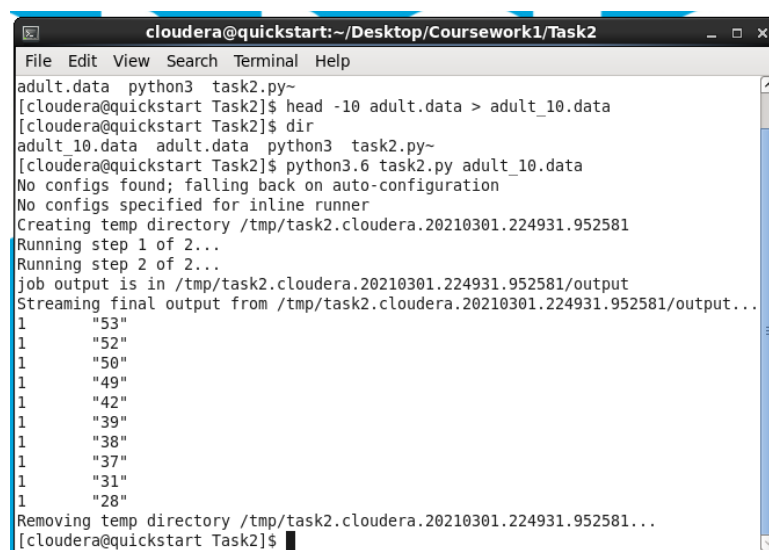
The first thing we do, is convert the tuple of values, that was the input of this reducer, into a list. This allows us to sort the list and have at the beginning the most frequent values. As we apply the *sort* method to a list of tuples, it sorts the first element of the list. This explains why we passed before the total times each age appears instead of the actual age.

After sorting our list, we set a counter to 0 so we can exit our loop with a *break* statement whenever we consider our job finished: when we already printed the 10 most common values. For this to work, we cannot forget to increment the value of our counter each time the loop is run. If not, we will enter an infinite loop that will crash our program.

Each time the loop runs, we yield the value we want to print through the command. This is the tuple we outputted as a value in *reducer1*. So, the input and the output of this second reducer for the first 10 rows of *adult.data* are the following:

REDUCER2 INPUT: REDUCER1 OUTPUT

REDUCER2 OUTPUT: it is the same as the whole program output:



```
cloudera@quickstart:~/Desktop/Coursework1/Task2
File Edit View Search Terminal Help
adult.data python3 task2.py~
[cloudera@quickstart Task2]$ head -10 adult.data > adult_10.data
[cloudera@quickstart Task2]$ dir
adult_10.data adult.data python3 task2.py~
[cloudera@quickstart Task2]$ python3.6 task2.py adult_10.data
No configs found; falling back on auto-configuration
No configs specified for inline runner
Creating temp directory /tmp/task2.cloudera.20210301.224931.952581
Running step 1 of 2...
Running step 2 of 2...
job output is in /tmp/task2.cloudera.20210301.224931.952581/output
Streaming final output from /tmp/task2.cloudera.20210301.224931.952581/output...
1      "53"
1      "52"
1      "50"
1      "49"
1      "42"
1      "39"
1      "38"
1      "37"
1      "31"
1      "28"
Removing temp directory /tmp/task2.cloudera.20210301.224931.952581...
[cloudera@quickstart Task2]$
```

As all the age values have the same number of occurrences, the `.sort()` method sorts the second value of the tuple, that explains why the age values are sorted from the highest to the lowest.

The last part of our code is the one that defines the steps our program has to execute the mappers, combiners or reducers. As I said before, this must be set because by default, *mrjob* uses a single MapReduce task to execute the program. So, if you wish to use more, you must specify it in *MRStep*.

We will apply *mapper-combiner-reducer1-reducer2*, in the same order we explained it in the report:

```
# Lastly, we set the steps we want our program to follow. mrjob by default uses a single MapReduce task to execute the program.
# We redefine steps() to create two jobs, each defined with MRStep(). This is mainly the order in which we will execute the
# map/reduce jobs.
def steps(self):
    return [
        MRStep(mapper = self.mapper, # first, we map the different age values in the specified document (adult.data)
              combiner = self.combiner, # second, we call our combiner to optimise the time of execution
              reducer = self.reducer1), # third, we reduce the different ages and return the count of each one
        MRStep(reducer = self.reducer2) # fourth and last, we return the 10 most frequent age values
    ]
```

I will leave in the report the result of applying this program (*task2.py*) to the whole dataset of *adult.data* too:

```
cloudera@quickstart:~/Desktop/Coursework1/Task2
File Edit View Search Terminal Help
1      "37"
1      "31"
1      "28"
Removing temp directory /tmp/task2.cloudera.20210301.224931.952581...
[cloudera@quickstart Task2]$ python3.6 task2.py adult.data
No configs found; falling back on auto-configuration
No configs specified for inline runner
Creating temp directory /tmp/task2.cloudera.20210301.225111.095471
Running step 1 of 2...
Running step 2 of 2...
job output is in /tmp/task2.cloudera.20210301.225111.095471/output
Streaming final output from /tmp/task2.cloudera.20210301.225111.095471/output...
898    "36"
888    "31"
886    "34"
877    "23"
876    "35"
875    "33"
867    "28"
861    "30"
858    "37"
841    "25"
Removing temp directory /tmp/task2.cloudera.20210301.225111.095471...
[cloudera@quickstart Task2]$
```

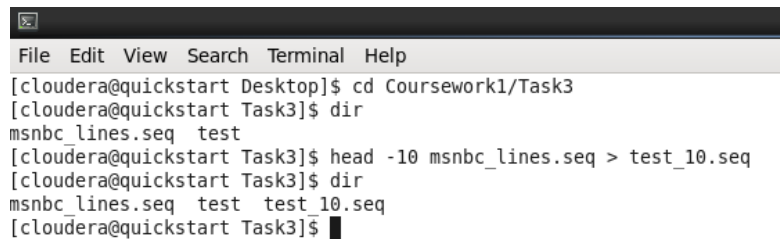
As we can see, the most frequent age value in the whole *adult.data* data set is the age of **36**, which is repeated almost 900 times. We can also observe that the range of values are between the 23 and the 38 years.

These values outputted by the command line are also stored in *task2.out*, which will be submitted to KEATS.

TASK 3: MapReduce INVERTED INDEX:

In this task, you will implement a simple inverted index and apply it to a web dataset coming from msn.com. Write a program `task3.py` using `mrjob`. Given the dataset, your program must output each symbol of the dataset along with a set of line-ids indicating the lines in which the symbol appears.

For this task of the coursework, I have first downloaded the *msnbc_seq* document from KEATS. I will now proceed just like before: after storing it in a new folder called Task3 (in Coursework1), I have also created another file with the 10 first entries of this data set for testing purposes. This is done with the command `head -n file>new_file`:



```
[cloudera@quickstart Desktop]$ cd Coursework1/Task3
[cloudera@quickstart Task3]$ dir
msnbc_lines.seq  test
[cloudera@quickstart Task3]$ head -10 msnbc_lines.seq > test_10.seq
[cloudera@quickstart Task3]$ dir
msnbc_lines.seq  test  test_10.seq
[cloudera@quickstart Task3]$
```

Just like before, we can observe that before we had only two documents (*msnbc_lines.seq* and *test*), and now we have a third file, which we have called *test_10.seq*. On this file we will perform tests to our code and check its efficiency. We will later export the final results in another file called *task3.out*. This file will perform the MapReduce task on the whole data set.

This task of the coursework asks us to create an inverted index, which serves mainly to faster locate specific values in a document. It is used for example to find words in a text, or, in our case, to find symbols in a certain line of the file. We will have to output the symbol and the lines where this symbol can be found.

Now, we have to create a python file that does all the job we are required to do. For that, we first need to import the modules we need. These modules are the following:

- *MRJob*: which allows us to perform MapReduce tasks in Python2.6 or more, and Python3.3 or more. As we are performing the code in Python3.6, it will serve us.
- *re*: to read the data in the dataset we will use. RE stands for Regular Expressions or RegEx in Python.

In this Python file we will write the code necessary for task 3 of the Coursework 1.

We have to create something called 'Inverted Index' that serves mainly to find specific things faster. In our case we will find symbols in a specific line row. Each row has a number of symbols, and we will have to output the symbols and the lines where these symbols are located.

We will start by importing the libraries we will need.

from mrjob.job import MRJob # this module let us write MapReduce jobs in Python 2.6+/3.3+ and run them on several platforms.
import re # this one is for Regular expressions (RegEx) in Python.

Now we will define the class that will perform the tasks we are going to define to the mapper and reducer. Note that in this task we will not use neither *MRStep*, nor any type of *combiner*. *MRStep* will not be used because we will only have one mapper and one reducer, so no kind of sequential order has to be specified.

Just like before, the first command of our code is to compile the file on which we will perform the MapReduce task on. This is done with the *re* module we imported at the beginning.

WORD_RE = re.compile(r"[\w]+") # this command is to read the document/file we are wishing to apply the MapReduce task on.

We will call the class *inverseIndex* and, as we said, it will only have a mapper and a reducer. As the whole code is visible in one single screenshot, I will put it all together in the next image, to later explain it better with my words:

```
# We will define first the mapper. This mapper will accept as parameters an empty key (__) and a line as values. It will
# output a number of key-value pairs (symbol, line_number)
def mapper(self, __, line):

    # This command will split the lines of msnbc_lines and store them in a list called 'data'.
    data = re.split(' +', line) # Now, each line will be stored in data. Depending on the line number, this data list
    # will have one format or another. If the dataline has 6 integers, in other words, if
    # the dataline is between 100000 and 989818 (last entry of the datafile), the first
    # value of the list will be the line number. In every other case, the first value of
    # data will be an empty string (''), so we would have to select the next value.

    # Here we store in line_number the value of the line where the information is.
    if data[0] == '': # If the line number is below 100000.
        del data[0] # delete the first element of the list.
    line_number = int(data.pop(0).strip()) # We store the line number (first value of data) in the variable 'line_number'.
    # .strip() just removes spaces at the beginning and at the end of the string.

    # 'yield' is similar to 'return', but it does not terminate the function. It produces key-value pairs.
    # In this case, we will yield a symbol and the line where it is located.
    for symbol in data: # For every symbol stored in the specific line number:
        yield symbol, line_number # yield/return/output the symbol and the line number of that specific symbol.

# Next, we will define the reducer. This will collect all the symbols with the line number it belongs to and output a
# sorted list of the line numbers where it appears. So, it will accept two different parameters: a key (symbol) and a
# value (the line number where it is located)
def reducer(self, symbol, line_number):

    # We create a set for one main reason: its characteristic related to duplicated values. A set cannot have two equal
    # values, if you try to add a value that is already stored there, it will omit it. This makes sets much more efficient.
    total_lines = set() # we initialize the set.

    # We traverse the list of values passed by the mapper and add the line numbers to the set created above.
    for line in line_number:
        total_lines.add(line) # For sets you use .add() instead of append().

    # Now we convert the set into a list and sort it storing the first rows at the beginning, just like the output shown in
    # the coursework description
    total_lines = list(total_lines) # convert the set into a list.
    total_lines.sort() # sort the values of the list converted in the last code line.

    # Now we are in position to yield/return the symbol and all the lines where that symbol appears. As we created a set, the
    # values of total_lines will not include any duplicated values
    yield symbol, total_lines
```

As we can see in the image, we first define the mapper. This mapper will basically accept as input the lines of the file you are passing in the command line. It will keep two values, the first one, *line_number* will contain the number of the line, the first value of the document. The other value will be stored in *data* and it will contain all the symbols that line has.

We can observe that if the line number is lower than 100000 the first value of the list is deleted because it is an empty string ("). After that, we *.pop()* the first value, the line number and the rest of the information stays in *data* (all the symbols).

Now, to avoid using more than one MapReduce task, we yield the symbol and the line number that contains that symbol. This will output something like the following image. I have applied the mapper only to the 10 first lines of *msnbc_lines.seq*.

As we can see from the following image, the output is composed of a key that is a string (the symbol) and a value that is an integer (the line number)

MAPPER INPUT: the file specified in the command line, in my case it was the 10 first lines of the *msnbc_lines.seq* file:

```
task3.py task2.py test_10.seq
1 1 1
2 2
3 3 2 2 4 2 2 3 3
4 5
5 1
6 6
7 1 1
8 6
9 6 7 7 7 6 6 8 8 8 8
10 6 9 4 4 4 10 3 10 5 10 4 4 4
```

MAPPER OUTPUT:

```
Streaming final output from /tmp/task3.cloudera.20210301.232949.934648/output...
"6"      8
"6"      9
"7"      9
"7"      9
"7"      9
"6"      9
"6"      9
"8"      9
"8"      9
"8"      9
"8"      9
"6"     10
"9"     10
"4"     10
"4"     10
"4"     10
"10"    10
"3"     10
"10"    10
"5"     10
"10"    10
"4"     10
"4"     10
"4"     10
"5"      4
"1"      5
"6"      6
"1"      7
"1"      7
"1"      1
"1"      1
"2"      2
"3"      3
"2"      3
"2"      3
"4"      3
"2"      3
"2"      3
"2"      3
"3"      3
"3"      3
```

The reducer will accept as a parameter the symbol and the line number where it belongs to. What it will do will be create a set (which does not accept duplicates) and put in that set all the lines where that symbol appears.

After creating the set it will convert it to a list so it can be sorted and the output is the same as the one shown in the coursework description.

REDUCER INPUT: this will be a symbol and a list containing all the line numbers.

REDUCER OUTPUT:

```
[cloudera@quickstart Task3]$ python3.6 task3.py test
No configs found; falling back on auto-configuration
No configs specified for inline runner
Creating temp directory /tmp/task3.cloudera.20210301.234803.283955
Running step 1 of 1...
job output is in /tmp/task3.cloudera.20210301.234803.283955/output
Streaming final output from /tmp/task3.cloudera.20210301.234803.283955/output...
"5"      [4, 10]
"6"      [6, 8, 9, 10]
"7"      [9]
"3"      [3, 10]
"4"      [3, 10]
"8"      [9]
"9"      [10]
"1"      [1, 5, 7]
"10"     [10]
"2"      [2, 3]
Removing temp directory /tmp/task3.cloudera.20210301.234803.283955...
[cloudera@quickstart Task3]$ python3.6 task3.py test_10.seq
No configs found; falling back on auto-configuration
No configs specified for inline runner
Creating temp directory /tmp/task3.cloudera.20210301.235201.479622
Running step 1 of 1...
job output is in /tmp/task3.cloudera.20210301.235201.479622/output
Streaming final output from /tmp/task3.cloudera.20210301.235201.479622/output...
"5"      [4, 10]
"6"      [6, 8, 9, 10]
"7"      [9]
"3"      [3, 10]
"4"      [3, 10]
"8"      [9]
"9"      [10]
"1"      [1, 5, 7]
"10"     [10]
"2"      [2, 3]
Removing temp directory /tmp/task3.cloudera.20210301.235201.479622...
```

As we can see, the output from the *test* file from KEATS is the same as the coursework. The second part of the command line is the output when applied to the first 10 rows of the whole data set.