Group Assignment 4
Page Rank

Team 2: Thanh Le, Antoine Si

PAGE RANK

BUILD ADJACENCY MATRIX

```python
# function to build the adjacency matrix
def build_adjacency_matrix(self, input_file):
    with open(input_file, 'r') as file:
        lines = file.readlines()

        # record the number of pages
        self.page_count = (int)(lines[0].strip())

        # record the number of links
        self.link_count = (int)(lines[1].strip())

        # initialize a matrix of 0s with size of page_count x page_count
        adjacency_matrix = np.zeros((self.page_count, self.page_count))

        # iterate over each line in the input file to record the link to the matrix
        for line in lines[2:]:
            link = line.split()
            src = int(link[0])
            dst = int(link[1])
            adjacency_matrix[src][dst] = 1

    return adjacency_matrix
```

BUILD TRANSITION PROBABILITY MATRIX

```
# function to build the transition probability matrix
def build_transition_probability_matrix(self, adjacency_matrix):
    transition_probability_matrix = np.nan_to_num(np.divide(adjacency_matrix, adjacency_matrix.sum(axis=1)[:, None]), nan=0)
    return transition_probability_matrix
```

- Given an adjacency matrix A
- If a row of A has no 1's, then replace each element by 1/N.
- Otherwise, divide each 1 in A by the number of 1's in its row
    - Ex: if there is a row with three 1's, then each of them is replaced by 1/3.

BUILD TRANSITION PROBABILITY MATRIX

```
# function to build the transition probability matrix with teleporting
def build_transition_probability_matrix_with_teleporting(self, transition_probability_matrix, teleportation_rate):
    transition_probability_matrix_with_teleporting = transition_probability_matrix * (1 - teleportation_rate)
    transition_probability_matrix_with_teleporting += (teleportation_rate / self.page_count)
    return transition_probability_matrix_with_teleporting
```

- Given a transition probability matrix B and teleportation rate α
- Multiply B by 1 − α
- Add α/N to every entry of the resulting matrix.

PAGE RANK

```python
# function to implement pagerank algorithm
# input_file - input file that follows the format provided in the assignment description
def pagerank(self, input_file):
    # record the start time
    start_time = time.time()

    # build the adjacency matrix
    adjacency_matrix = self.build_adjacency_matrix(input_file)

    # build the transition probability matrix
    transition_probability_matrix = self.build_transition_probability_matrix(adjacency_matrix)

    # build the transition probability matrix with teleporting
    teleportation_rate = 0.15
    transition_probability_matrix_with_teleporting = self.build_transition_probability_matrix_with_teleporting(transition_probability_matrix, teleportation_rate)
```

PAGE RANK

```python
# initialize initial probability distribution vector
probability_vector = np.ones(self.page_count)/self.page_count

# initialize epsilon value as threshold for indicating steady state
eps = 1e-5

# initilize count of iterations
iterations = 0

# calculate the page rank
while True:
    last_probability_vector = probability_vector
    probability_vector = np.dot(last_probability_vector, transition_probability_matrix_with_teleporting)
    iterations += 1
    if np.sum(np.abs(probability_vector - last_probability_vector))/self.page_count < eps:
        break

# initialize a dictionary to store the result where key is the page_id and value is the corresponding pagerank value
result = {k: v for k, v in enumerate(probability_vector)}

# sort the page rank
result = dict(sorted(result.items(), key=lambda item: item[1], reverse=True))

# record the end time
end_time = time.time()

# print the output
print("Number of pages: ", self.page_count)
print("Number of links: ", self.link_count)
print("\nPageRank calculated in {} iterations ({} seconds)".format(iterations, end_time - start_time))
print("Page ID\t\tPageRank Value")
for page_id in result:
    print("{}\t\t{}".format(page_id, result[page_id]))
```

The initial vector in power iteration loop is initialized with 1/num_of_pages in all the elements

The vector is updated over each iteration and is calculated by the dot product of the last vector and the transition probability matrix with teleporting

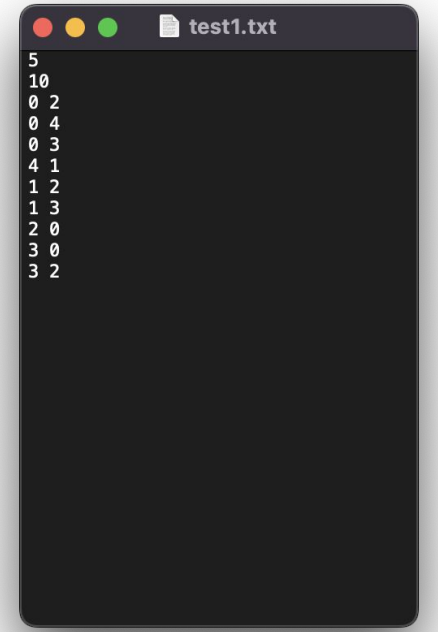
PAGE RANK

```python
# initialize initial probability distribution vector
probability_vector = np.ones(self.page_count)/self.page_count

# initialize epsilon value as threshold for indicating steady state
eps = 1e-5

# initilize count of iterations
iterations = 0

# calculate the page rank
while True:
    last_probability_vector = probability_vector
    probability_vector = np.dot(last_probability_vector, transition_probability_matrix_with_teleporting)
    iterations += 1
    if np.sum(np.abs(probability_vector - last_probability_vector))/self.page_count < eps:
        break

# initialize a dictionary to store the result where key is the page_id and value is the corresponding page rank value
result = {k: v for k, v in enumerate(probability_vector)}

# sort the page rank
result = dict(sorted(result.items(), key=lambda item: item[1], reverse=True))

# record the end time
end_time = time.time()

# print the output
print("Number of pages: ", self.page_count)
print("Number of links: ", self.link_count)
print("\nPageRank calculated in {} iterations ({} seconds)".format(iterations, end_time - start_time))
print("Page ID\t\tPageRank Value")
for page_id in result:
    print("{}\t\t{}".format(page_id, result[page_id]))
```

Epsilon value is set to be 1e-5 in this assignment. Smaller value yields more accurate result but takes more time (more iteration).

Determine the steady state by checking if the difference between the current iteration and the last one falls below the specified epsilon.

EXPERIMENTAL RESULT (TEST1.TXT)




```
Number of pages:  5
Number of links:  10

PageRank calculated in 19 iterations (0.00019788742065429688 seconds)
Page ID         PageRank Value
0               0.31899680476330367
2               0.25168131582784947
3               0.17661591062273685
1               0.13232630994072855
4               0.12037965884538143
```

EXPERIMENTAL RESULT (TEST2.TXT)


```
Number of pages:  10
Number of links:  15

PageRank calculated in 78 iterations (0.0006330013275146484 seconds)
Page ID         PageRank Value
8               0.000146371827523615
9               0.00013953238693068264
5               0.0001321575836596206
3               0.00012420551176896475
1               0.00011417120956767715
0               0.00011092571318972676
2               9.487597832823341e-05
7               8.484167612694584e-05
4               7.276889727873644e-05
6               1.696858297238537e-05
```

EXPERIMENTAL RESULT

```
Number of pages:  5
Number of links:  10

PageRank calculated in 5 iterations (0.00014090538024902344 seconds)
Page ID         PageRank Value
0               0.31854055729166664
2               0.2509200321180555
3               0.17461386458333333
1               0.13776766319444445
4               0.11815788281250002


Number of pages:  10
Number of links:  15

PageRank calculated in 2 iterations (9.918212890625e-05 seconds)
Page ID         PageRank Value
3               0.10453333333333337
5               0.10347083333333336
8               0.09745000000000001
9               0.09745000000000001
0               0.08558541666666668
2               0.08222083333333335
1               0.07726250000000001
7               0.054950000000000006
4               0.05300208333333335
6               0.01245
```

Epsilon = 1e-2

```
Number of pages:  5
Number of links:  10

PageRank calculated in 38 iterations (0.0002722740173339844 seconds)
Page ID         PageRank Value
0               0.31899306233476216
2               0.25168227014986366
3               0.17661913720967856
1               0.1323241623655669
4               0.12038136794012859


Number of pages:  10
Number of links:  15

PageRank calculated in 184 iterations (0.0006999969482421875 seconds)
Page ID         PageRank Value
8               1.4230775553975328e-08
9               1.3565821473402226e-08
5               1.2848817580776256e-08
3               1.2075689635463531e-08
1               1.1100120054327817e-08
0               1.0784581666256505e-08
2               9.224170907035266e-09
7               8.248601325899551e-09
4               7.0748439915250015e-09
6               1.649744352000145e-09
```

Epsilon = 1e-10

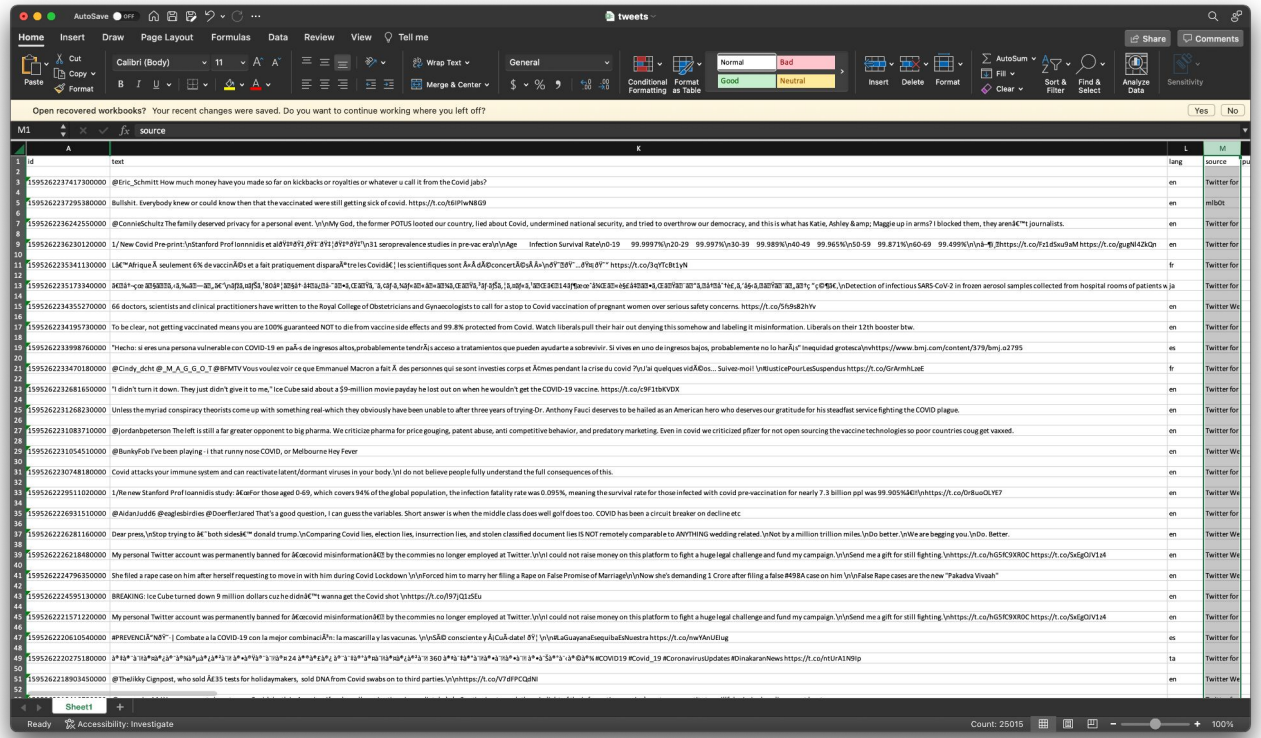
CONCLUSION

- Being able to detect the steady state is the metric for determine the PageRank’s accuracy.

- The value of epsilon should be picked wisely to be able to detect the steady state while not keeping the program running in much time → PageRank’s efficiency.

EXTRA CREDIT

TWITTER DATASET



Using:

- id (column A = index 1)
- text (column K = index 11)

for building the index

INDEX FORMAT

Each term entry in index will be of the form:

term: [(doc_id1, term_frequency1), (doc_id2, term_frequency2), …]

The positional data and weights are discarded in this assignment since the goal is focus on performance comparison.

TRADITIONAL INDEX

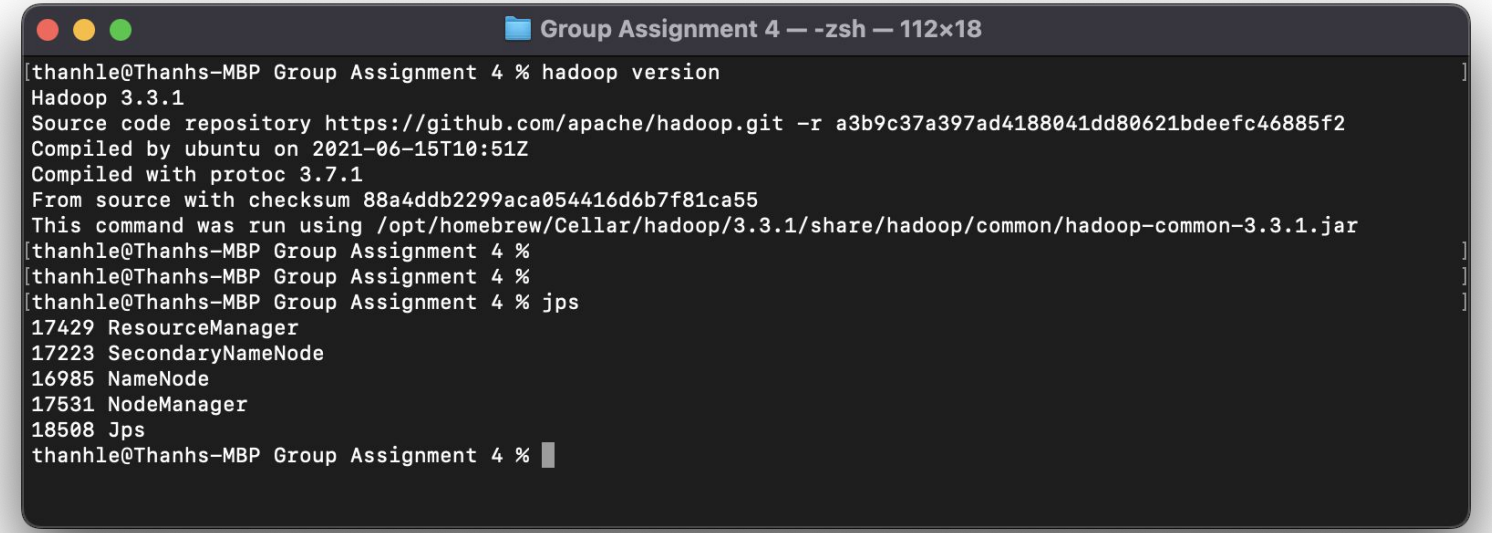
Using the one from Group Assignment 3

```python
# function to build an index using PyLucene
def build_lucene_index(self):
    directory_path = os.path.dirname(self.path) + '/index'

    # construct the directory to store the index on local file system
    self.directory = FSDirectory.open(File(directory_path).toPath())

    # configure an index write
    config = IndexWriterConfig()

    # always overite existing index to avoid duplicate files
    config.setOpenMode(IndexWriterConfig.OpenMode.CREATE)

    # construct writer, reader, and searcher
    self.writer = IndexWriter(self.directory, config)
    self.reader = DirectoryReader.open(self.directory)
    self.searcher = IndexSearcher(self.reader)

    # iterate through each line in the dataset file
    workbook = openpyxl.load_workbook(self.path).active
    for i in range(3, workbook.max_row + 1, 2):

        # get the tweet id in column 1
        id = workbook.cell(row=i, column=1).value

        # get the tweet text in column 11
        text = workbook.cell(row=i, column=11).value

        # construct doc and add to the IndexWriter
        self.add_doc(id, text)

    # close the writer
    self.writer.close()
```

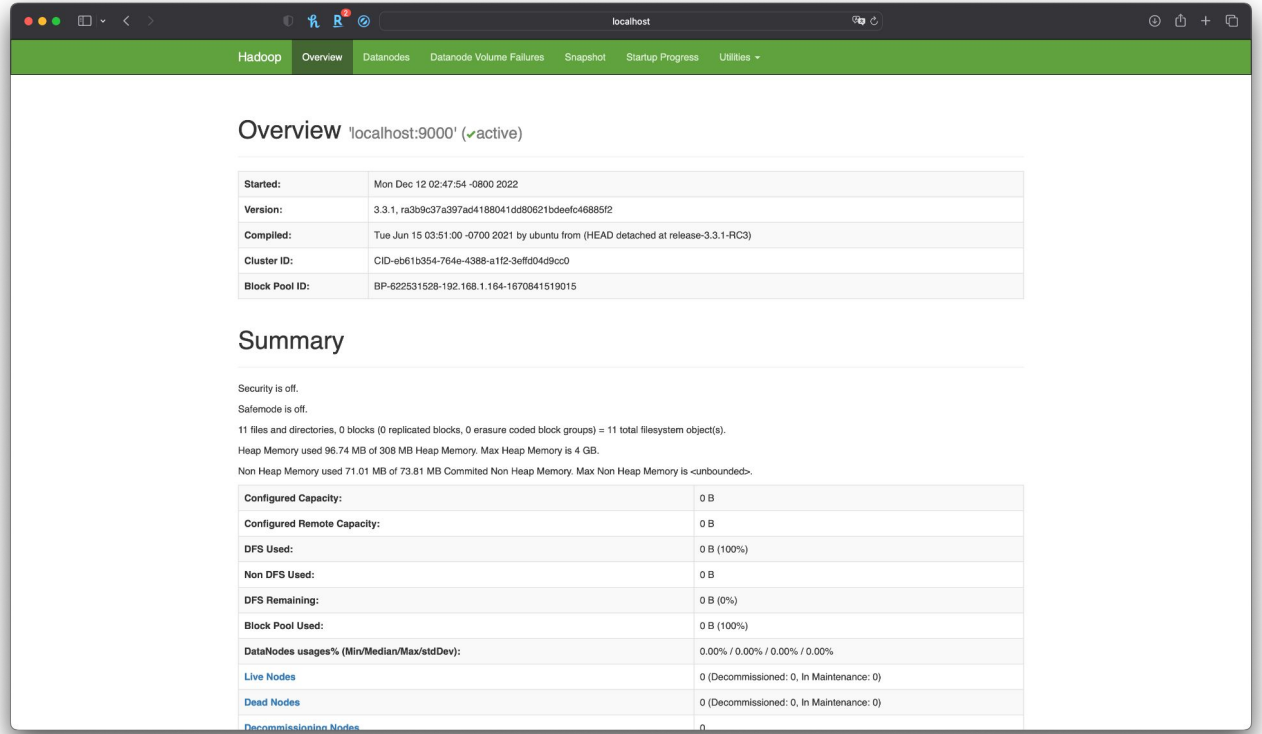
```python
# function to add a new document with specific id and text to the IndexWriter
def add_doc(self, id, text):
    # create a new document
    doc = Document()

    # configure how metadata is stored in the index
    metaType = FieldType()
    metaType.setStored(True)
    metaType.setTokenized(True)

    # configure how content data is stored in the index
    # store the doc id, term frequency, and postitions
    contentType = FieldType()
    contentType.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS)
    contentType.setStoreTermVectors(True)
    contentType.setStoreTermVectorPositions(True)
    contentType.setStored(True)
    contentType.setTokenized(True)

    # add the title and content field to the document
    doc.add(Field("id", id, metaType))
    doc.add(Field("text", text, contentType))

    # add the document to the IndexWriter
    self.writer.addDocument(doc)
```

```python
def build_traditional_index(self):
    print("-------------- Traditional Indexing --------------")
    self.index = collections.defaultdict(lambda: collections.defaultdict(list))

    # record the start time
    start_time = time.time()

    # build index using PyLucene
    self.build_lucene_index()

    # iterate through each document in the lucene index
    self.doc_list = self.get_doc_list()
    for doc in self.doc_list:

        # get the doc_id of the current document
        doc_id = doc.doc

        # get the term vector of the current document
        term_vector = self.reader.getTermVector(doc_id, "text")

        # check if there is term in the document
        if term_vector is not None:
            term_iter = BytesRefIterator.cast_(term_vector.iterator())

            # iterate through each term in the document
            while term_iter.next():
                terms_enum = TermsEnum.cast_(term_iter)

                # convert the term from UTF-8 byte format to string
                term = terms_enum.term().utf8ToString()

                # get the postings list of the term
                postings = terms_enum.postings(None, PostingsEnum.ALL)

                # iterate through each posting
                if postings.nextDoc() is not DocIdSetIterator.NO_MORE_DOCS:

                    # get the term frequency
                    freq = postings.freq()

                    # record the term frequency in the index
                    self.index[term][doc_id] = freq

    # record the end time
    end_time = time.time()
```

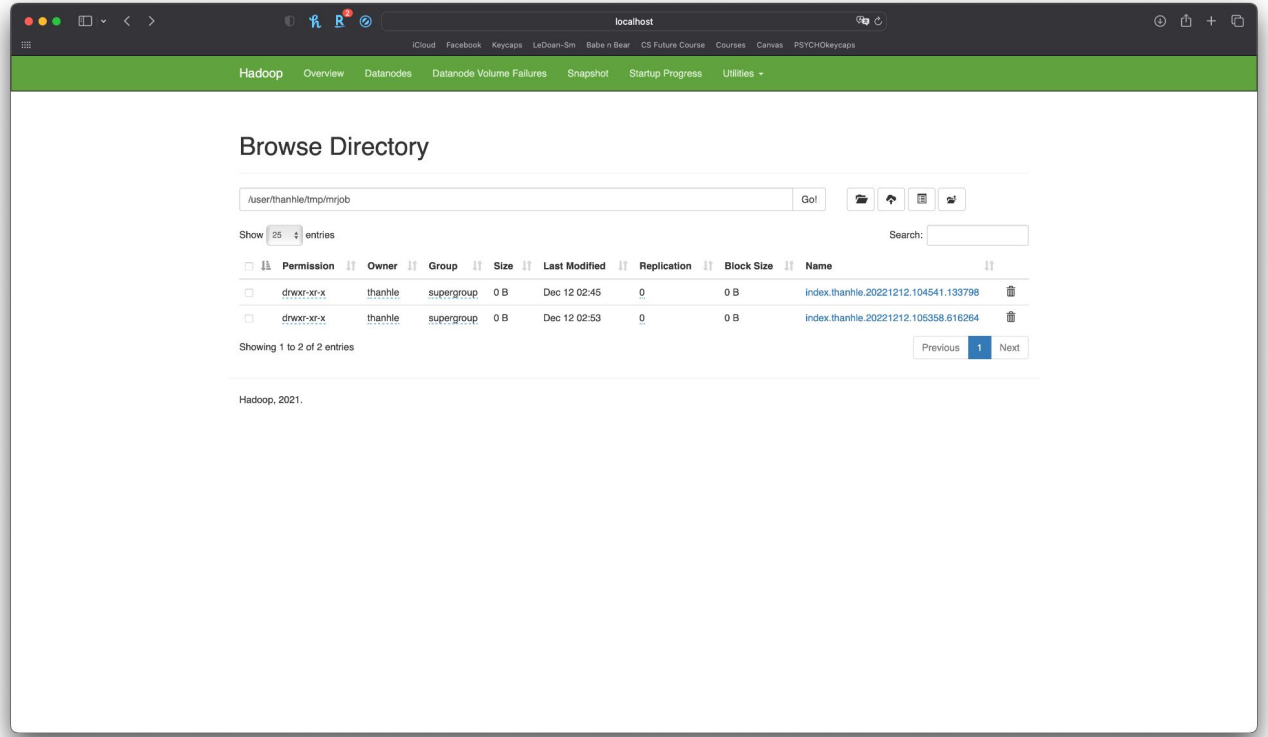
INSTALLING HADOOP



SETTING HADOOP CLUSTER LOCALLY



UPLOADING FILE TO HADOOP



LIBRARIES

```
# Python 3.0
import re
import sys
import os
import collections
import time

import lucene
from java.io import File
from org.apache.lucene.document import Document, Field, FieldType
from org.apache.lucene.util import BytesRefIterator

from org.apache.lucene.index import IndexWriter, IndexWriterConfig, PostingsEnum, IndexOptions, TermsEnum
from org.apache.lucene.store import FSDirectory

from org.apache.lucene.search import IndexSearcher, MatchAllDocsQuery, DocIdSetIterator
from org.apache.lucene.index import DirectoryReader

import openpyxl
from mrjob.job import MRJob
from mrjob.step import MRStep
```

- Openpyxl: python library to open and read excel files for tweets dataset
- MRjob: python library for MapReduce developed by YELP

MAP REDUCE

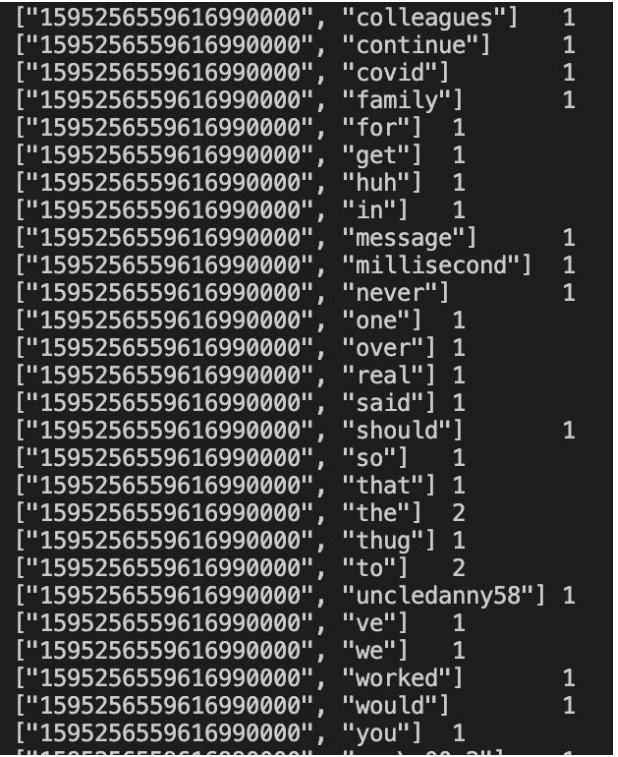
Using MapReduce operation to count the word frequency of each tweets

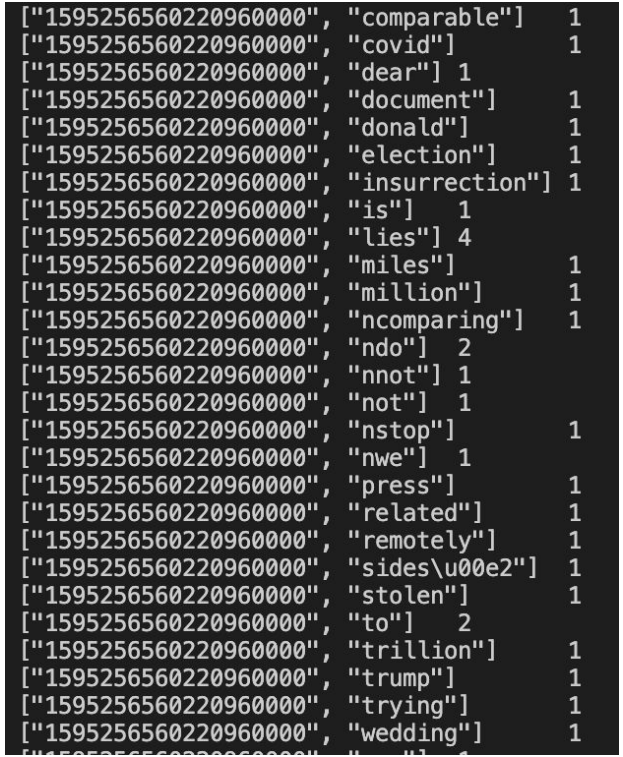
```
# regex to find non-alphabetical words
WORD_RE = re.compile(r"[\w']+")

# map-reduce job to count word frequency
class word_count(MRJob):

    def mapper_raw(self, input_path, input_uri):
        # iterate through each line in the dataset file
        workbook = openpyxl.load_workbook(input_path).active
        for i in range(3, workbook.max_row + 1, 2):

            # get the tweet id in column 1
            id = workbook.cell(row=i, column=1).value

            # get the tweet text in column 11
            text = workbook.cell(row=i, column=11).value

            # iterate through each term in the text and remove non-alphabetically words
            for term in WORD_RE.findall(text):

                # yield the ((id, term), term_frequency) pair
                yield ((id, term.lower()), 1)


    def reducer(self, key, value):
        # sum all term frequency with the same key (id, term)
        yield (key, sum(value))


    def steps(self):
        return [
            MRStep(
                mapper_raw=self.mapper_raw,
                reducer=self.reducer)
        ]
```

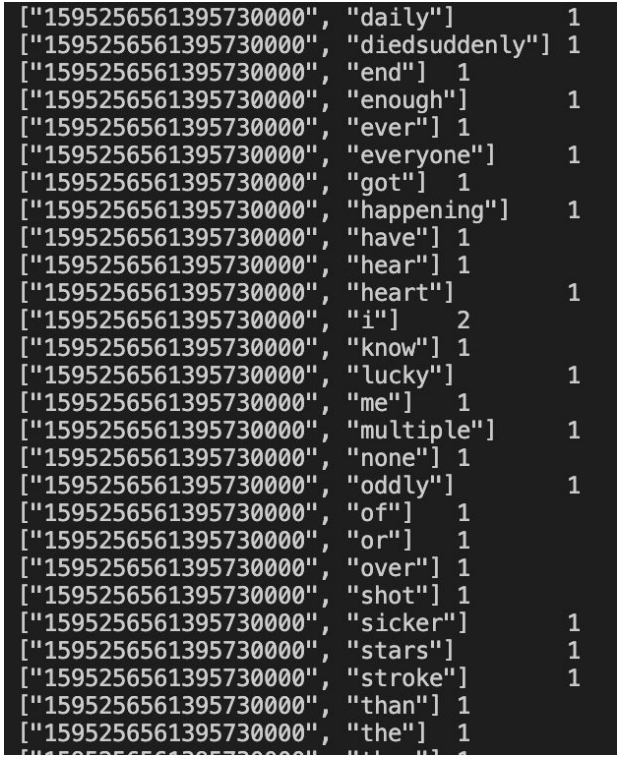
BUILDING DISTRIBUTED INDEX

```python
# function to build the distributed index using map reduce
def build_map_reduce_index(self):
    print("-------------- Distributed Indexing using Map Reduce --------------")
    self.index = collections.defaultdict(lambda: collections.defaultdict(list))

    # record the start time
    start_time = time.time()

    # instantiate the map-reduce job to count word frequency running on local hadoop cluster
    job = word_count(args=['-r', 'hadoop', self.path])

    # run the map-reduce job
    with job.make_runner() as runner:
        runner.run()

        # parse the output of the job and record the data to the index
        for (doc_id, term), freq in job.parse_output(runner.cat_output()):
            self.index[term][doc_id] = freq

    # record the end time
    end_time = time.time()

    # print the indexing time
    print("TF-IDF Index built in", end_time - start_time, "seconds.\n")
```

EXPERIMENTAL RESULT (WORD_COUNT OUTPUT)







EXPERIMENTAL RESULT (RUNTIME)

```
-------------- Traditional Indexing --------------
TF-IDF Index built in 16.255134105682373 seconds.


-------------- Distributed Indexing using Map Reduce --------------
No configs specified for inline runner
TF-IDF Index built in 12.771775245666504 seconds.
```

```
-------------- Traditional Indexing --------------
TF-IDF Index built in 16.14408802986145 seconds.


-------------- Distributed Indexing using Map Reduce --------------
No configs specified for inline runner
TF-IDF Index built in 12.734126091003418 seconds.
```

The distributed index results in a faster runtime on multiple tries compared to the traditional index

TRADITIONAL VS. DISTRIBUTED INDEXING

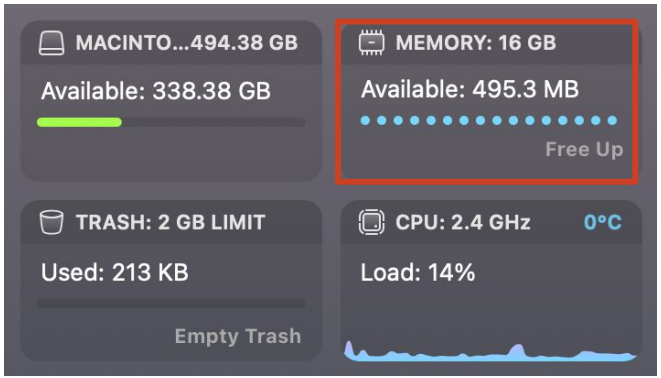
The data set given was around 15.3 MB or about 25,013 tweets.

The data we have shown with both traditional and distributed processing showed runtimes of about 16 seconds and 13 seconds respectively. This makes the speedup approximately 1.23 times faster for distributed compared to traditional.

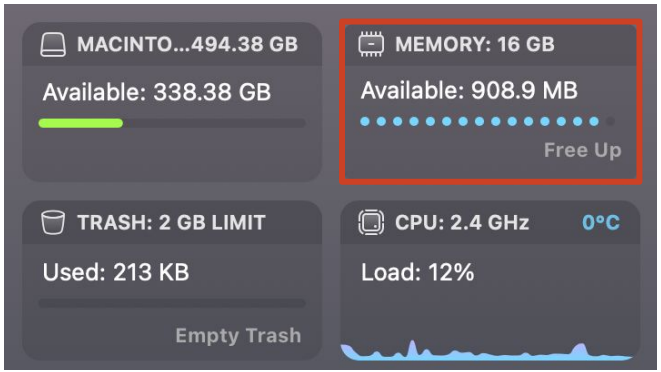
If the dataset were multitudes larger, distributed processing would be needed as distributed processing was designed for scalability, since it runs the data in parallel on different nodes/computers in a system rather than on a single computer in traditional processing.

EXPERIMENTAL RESULT (MEMORY USAGE)

- Free up the RAM memory before each run.
- Keep track of RAM memory usage during indexing with Map Reduce and Hadoop.



During indexing



After indexing

Much memory is used during indexing. The memory is released after usage.

CONCLUSION

- The distributed index built using Map Reducing running on Hadoop cluster shows better performance in run time.
- Analysis of speed up:
  - Hadoop optimizes memory and disk spill use. It aims to use as much memory as possible without triggering swapping → memory tuning to maximize the performance
  - Each MapReduce job in Hadoop collects the information about the various input records read, number of reducer records, number of records pipelined for further execution, swap memory, heap size set, etc.
  - Hadoop controls the amount of mapper and the size of each job. When dealing with large files, Hadoop splits the file into smaller chunks so that mapper can run it in parallel → speeding up the process.

TASK DIVISION

- Both of us first started working on implementing the index.py file by ourselves and discussed with each other when any problem happened
- We compared our query results with each other to make sure we got the same outputs
- We combined our works into a final version with detailed comments in each step which helps explaining the codes
- Antoine created the output.txt file
- Thanh commented the codes and submitted the zip file
- We worked on the presentation slides together