Final Report for Tiny Google

By:

Sam Yu and Mufaro Nyereyemhuka

***Design***

The design of project done in two parts. Create the inverted index, then use the inverted index to find the desired words in the provided texts. Since we were allowed to use whatever implementation and language, we wanted I used python to write this project.

My first order of business was to create a function to strip all unnecessary fluff from each word. The function made every word passed in lowercase, removed any form of punctuation and then stemmed it. Once I created an easy way to have every word uniform. I began writing a simple spark program to count each word.

The way I created the inverted index in spark was first loading all the text files into a rdd object. I wrote the word count program we have went over dozens of times in class. I started just did the simply mapping each word with the number one and then reducing it by summing each mapped word. Once I had all the words counted and stored in a csv the rest was easy. I set the word as the key and stored the files that were counted as the list for the value. That is how my build index works with spark.

Once I had the index built all I had to do was search the index for any word the user requested. All I had to do was sort the list of files that were returned for the word by highest count. Then once I had the list of files I searched through each sentence in the list of files for the first instance of that keyword and printed the first 10 results.

The setup for writing Hadoop was almost identical as spark because in this particular case I used the map and reduce functions in my spark implementation. The process was a bit more involved syntactically though for the Hadoop implementation but, it was still fairly simple since we were just counting a word by one and then reducing to count all of the mapped words. My implementation for map and reduce were outputted to the terminal and I had to catch the output going into the buffer to create the dict. Once all the words were counted and captured the rest of the procedure was the same. Use the cleaned words to create a dictionary that had the word as the key and the file and its word count as the value

***Implementation***

This is the code taken straight from the project where map and reduce are implemented.

***Spark:***

def word\_count():

conf = SparkConf().setAppName("Pyspark Pgm")

sc = SparkContext(conf = conf)

rdd = sc.textFile('input\\*')

counts = rdd.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

counts.coalesce(1).saveAsTextFile("res.csv")

***Hadoop: Special thanks to*** [***https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/***](https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/) ***where I learned most of this Hadoop.***

def read\_input(file):

for line in file:

# split the line into words

yield line.split()

def read\_mapper\_output(file, separator=" "):

for line in file:

yield line.rstrip().split(separator, 1)

def mapped(separator="" ""):

# input comes from STDIN (standard input)

data = read\_input(sys.stdin)

for words in data:

# write the results to STDOUT (standard output);

# what we output here will be the input for the

# Reduce step, i.e. the input for reducer.py

#

# tab-delimited; the trivial word count is 1

for word in words:

print(f"{word} {separator} 1")

def reduced(separator=" "):

# input comes from STDIN (standard input)

data = read\_mapper\_output(sys.stdin, separator=separator)

# groupby groups multiple word-count pairs by word,

# and creates an iterator that returns consecutive keys and their group:

# current\_word - string containing a word (the key)

# group - iterator yielding all ["&lt;current\_word&gt;", "&lt;count&gt;"] items

for current\_word, group in groupby(data, itemgetter(0)):

try:

total\_count = sum(int(count) for current\_word, count in group)

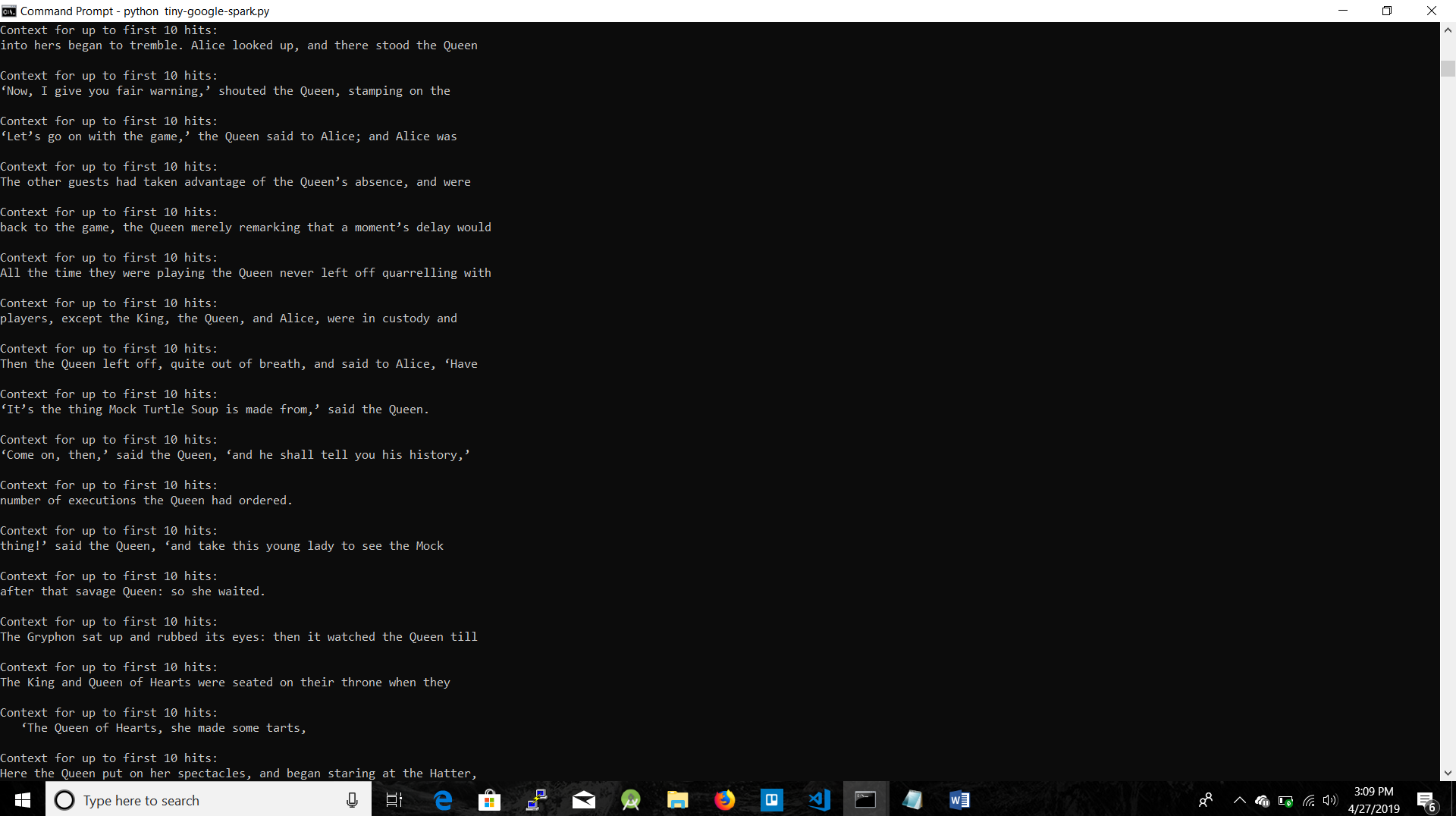
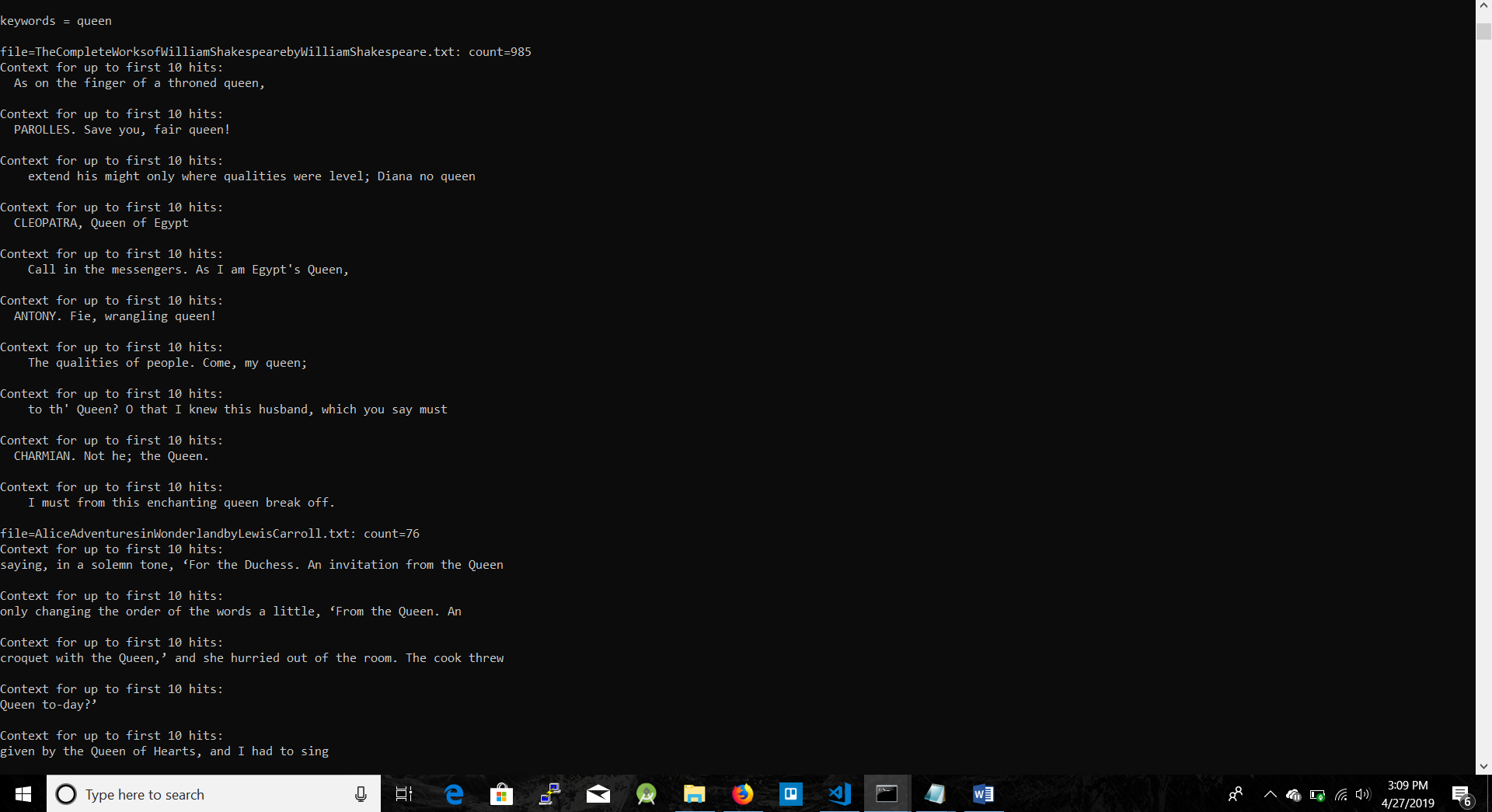
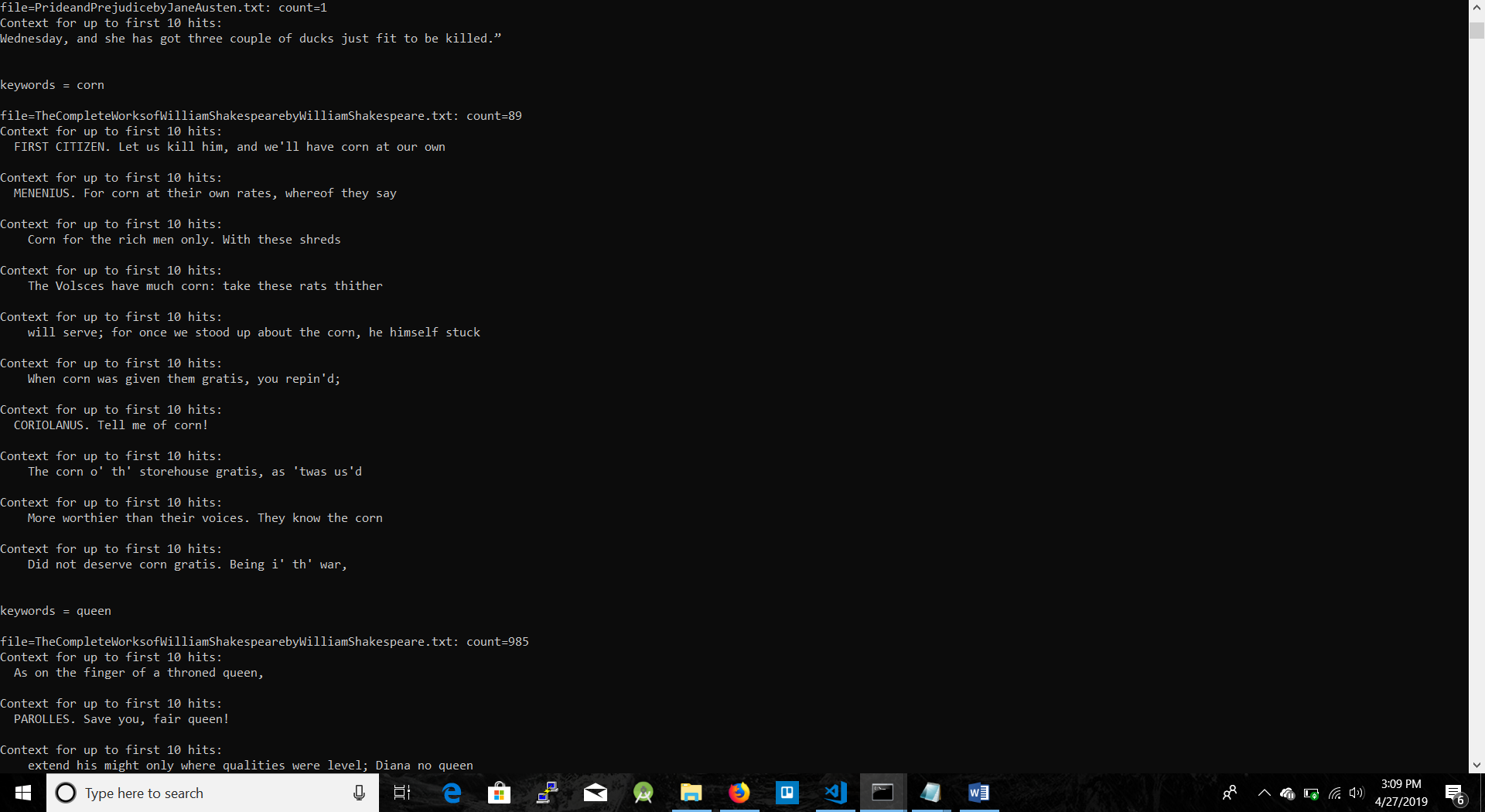
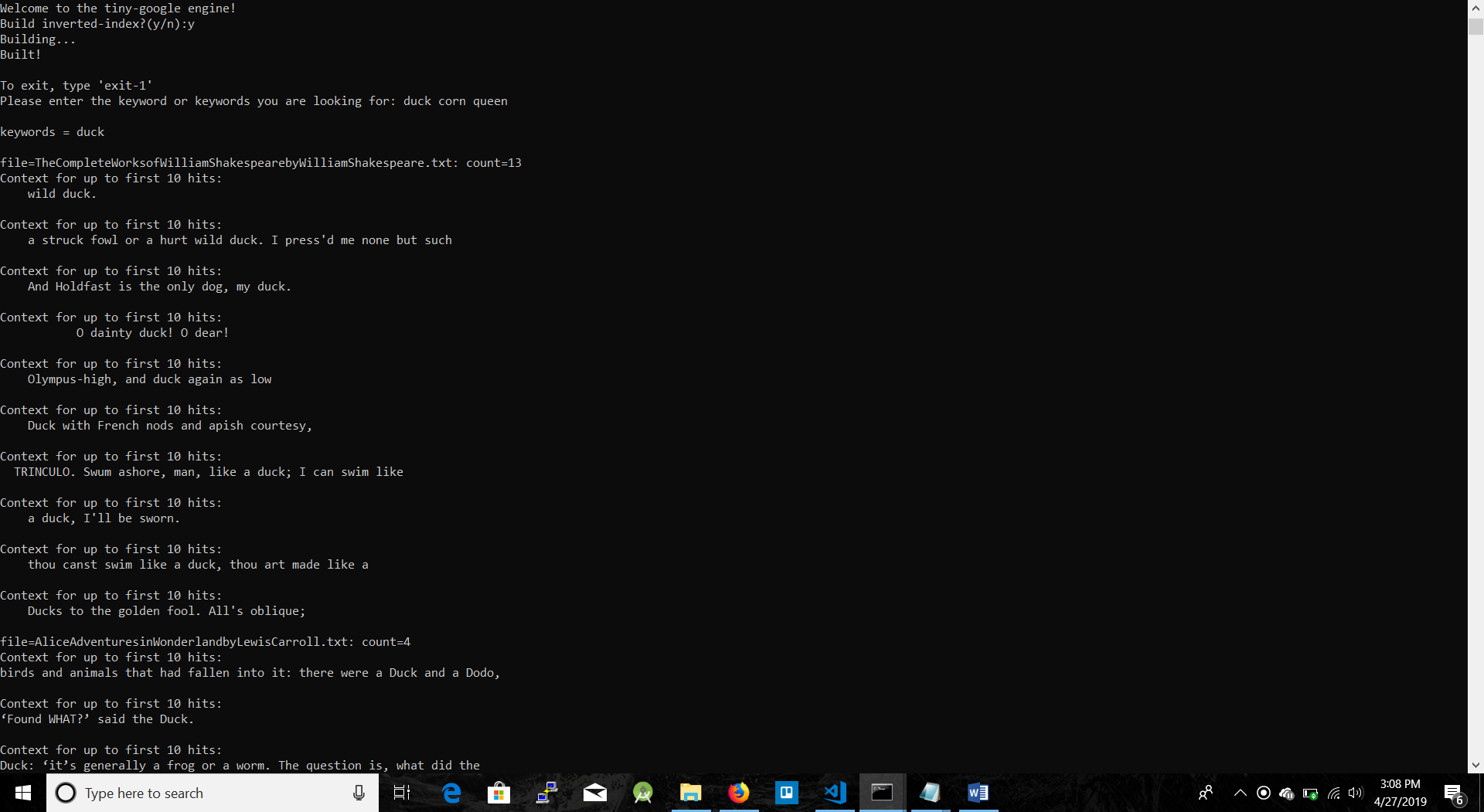
print(f"{current\_word} {separator} {total\_count}")

except ValueError:

# count was not a number, so silently discard this item

pass

***Snapshots***

Theses are the provided snapshots of the programing running on input duck corn queen. Provided in the output.txt file is the complete output of running tiny-google-spark and tiny-google-hadoop if you are curious how all of it looks. In addition I will provide my inverted-index as inverted-index.json.

***Comparison***

Overall, I much prefer working in Spark over Hadoop. Spark has much more python support and is much easier to install and configure over Hadoop. Spark has the ability to compile and write things in Scala which makes debugging and testing much easier. For Hadoop everything is in java and dependencies and file paths are much more difficult to get correct. The support and documentation is atrocious for Hadoop. Pip is far superior to Maven for handling dependencies. In the coding scheme I did, you can see how much better spark is when you simply just look how much easier it was to quickly write map and reduce. It was just a few lines for the spark implementation to create an iterator to and map things in the rdd object. While in Hadoop you had to create different files for map and reduce and make sure that each function can run in parallel. Handling file systems in Hadoop is a nightmare. Making everything in HDFS and using cat to just to see what you actually created is extremely unintuitive. Whilst if you look at my spark code you can see I could just load all my files into rdd with a simple regex expression and popped it out in a csv.