Big Data and Cloud Computing Mini-Report

Mafalda Matos (up201605629)

25 of March of 2023

1 Goal of Assignment

The goal of this assignment is to compare the time efficiency VS. the size of data, when doing the same task using different code implementations:

- Using a sequential Python implementation;
- Using Pyspark;
- Using Apache Beam.

The task given is to import text data from a CSV file, tokenize the words and count them.

We should also run the comparison in a local computer as well as in Google Cloud to see how the Hardware can impact the result.

2 Pipeline Code

Figure 1: Pipeline Code

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
import re
from apache_beam.io import WriteToText
from apache_beam.dataframe import convert
import pandas as pd

# create pipeline

input_file = 'test.csv'
output_path = 'counts.txt'

with beam.Pipeline() as p:
        # import texts from csv
        df = pd.read_csv(input_file, names=["type", "title", "text"])
        df = df.text

        # tokenize words
        # count word frequency
        counts = (
                convert.to_pcollection(df, pipeline=p) # converts dataframe into pcollection
                | 'ExtractWords' >> (
                        beam.FlatMap(
                                lambda x: re.findall(r'[A-Za-z\']+', x)).with_output_types(str))
                | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
                | 'GroupAndSum' >> beam.CombinePerKey(sum))

        def format_result(word_count):
                # Format the counts into a PCollection of strings.
                (word, count) = word_count
                return '%s: %s' % (word, count)

        output = counts | 'Format' >> beam.Map(format_result)

        # output | WriteToText(output_path)
```

The pipeline code starts by opening an Apache Beam pipeline and importing the CSV file using the Pandas function. Here we define a header so that we can easily get the row with the texts.

Then we tokenize and count the word frequency. We start by converting the dataframe into a PCollection so that we may continue to use the data in the pipeline. The next step is to tokenize words by using a FlatMap that finds all of what can be considered part of a word. Afterwards, we initiate the count at one by creating a Map that adds a one to the words, and we end this part by summing all of the repeating words, so that we get something like (word, count).

The final step in the pipeline is to use a Map to format the result into something readable, in this case a PCollection of "word: count". We have also

added in a comment of what the command to write the output to a text file would look like, even though we don't use it. We chose only to calculate the counts as the task (not to print or save them) and we want to keep things more or less equal for every code.

3 Code Design and Implementation Choices

Figure 2: Sequential Python Code

```
import csv

# import texts from csv
texts = []
with open('test.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        line_count = 0
        for row in csv_reader:
                texts.append(row[2])
                line_count += 1

# tokenize words
words = []
for t in texts:
        word = t.split()
        for w in word:
                words.append(w)

# count word frequency
freq = {}
for w in words:
        if w in list(freq.keys()):
                freq[w] += 1
        else:
                freq[w] = 1
#print(freq)
```

The sequential Python code is as simple as it could be. First, we import the texts from the CSV file by saving the third row only - the one that has the full texts. Afterwards we will tokenize the words by using the split function, and finally, we will count the word frequency using a dictionary. If the word is not in the dictionary then we will start the count at 1, and if it then we will add 1 to the value it already has.

There are a lot of optimizations that could be made - namely using other data structures to store the data, for example, using dictionaries instead of lists, as well as doing everything in two loops instead of three - but this serves as a good "control" code to see how much more efficient the other ways of coding are.

Figure 3: Pyspark Code

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import Tokenizer
import pyspark.sql.functions as f

# import texts from csv
spark = SparkSession\
  .builder \
  .appName("PythonWordCount") \
  .getOrCreate()

df = spark.read.csv("test.csv")

# tokenize words
tokenizer = Tokenizer(inputCol='_c2', outputCol='words_token')
df_words_token = tokenizer.transform(df)

# count word frequency
result = df_words_token.withColumn('word', f.explode(f.col('words_token'))) \
  .groupBy('word') \
  .count().sort('count', ascending=False) \

spark.stop()
```

This code should be the most simple yet fast implementation in Pyspark. It begins by starting a Pyspark session and reading the CSV file. Then we tokenize the words of the texts using Tokenizer, which will transform the dataframe. To finish, we use the SQL functions of Pyspark to explode the column with the tokenized words, so we can group and count them, then we end by simply closing the session.

4 Hardware

4.1 Locally

- Processor: 1 CPU - Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz
- Number of cores: 4 cores
- Memory: 8 GB

4.2 Google Cloud

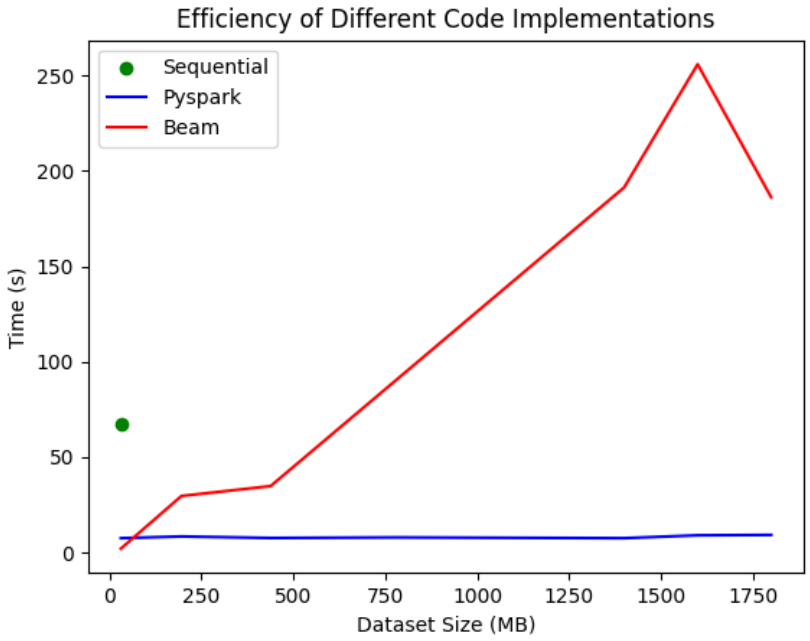
- Processor: 2 vCPU - Intel(R) Xeon(R) CPU @ 2.20GHz
- Number of cores: 4 cores
- Memory: 16 GB

5 Plots

To get the run times for different files, we just ran the scripts after changing the "test.csv" file to the desired file, as well as adding a timeit command to count the time passed. We then compiled the results in the following plot.

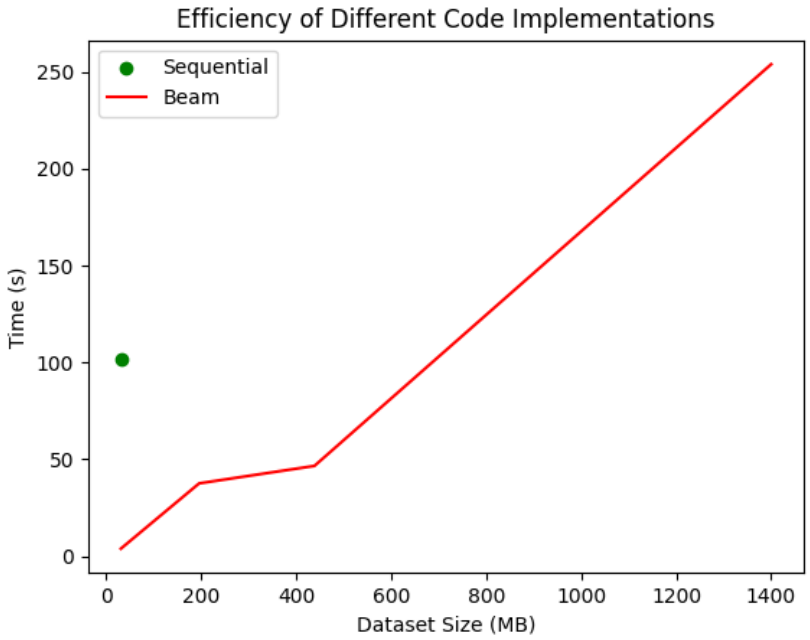
5.1 Locally

Figure 4: Plot of Efficiency for Local Computer



5.2 Google Cloud

Figure 5: Plot of Efficiency for Google Cloud



For the sequential Python implementation, we noticed that as soon as the dataset hit around 100 MB in size, our implementation took way too long to actually do the task (more than 20 minutes), which is why we haven't included the values in Figure 4 or 5.

As we can see, the computers perform similarly, with the Google Cloud computer being slightly worse, which is to be expected given the Hardware limitations.

Not only that but we can also see that the sequential Python code runs with O(n!) complexity, while the Beam code runs with O(n) (in general) and Pyspark runs with O(1). There are potential optimizations to be made however, so this result may not be fully conclusive.

6 Difficulties Found

Importing the CSV in Apache Beam was challenging as there is no direct way to do it onto the pipeline. Hence why in Beam code (Figure 1), we imported the CSV as a dataframe and then converted that into a PCollections to be able to use in the pipeline. There are other methods of doing this, but this seemed

to be the most simple. Of course, that doesn't mean that it's necessarily the most efficient, so there might be some optimizations to be made here.

6.1 Locally

Pyspark was somewhat difficult to install because it has a lot of requirements such as Python, Java and Spark, that you have to add onto your .bashrc file and your /etc/environment file (in Linux). This was tricky in our case because the path_files in the computer we were using were incorrect, but after some trouble shooting we were able to get everything to work.

6.2 Google Cloud

Due to difficulties in installing Pyspark on Google Cloud, we were not able to run the Pyspark implementation there.

It would also be interesting to try using a virtual machine from Google Cloud which has GPUs, but due to a lack of time we were unable to do so.

7 References

1. Apache Beam Wordcount Example - `https://beam.apache.org/get-started/wordcount-example/`
2. Apache Beam Dataframes Notebook - `https://colab.research.google.com/github/apache/beam/blob/master/examples/notebooks/tour-of-beam/dataframes.ipynb#scrollTo=t6xNIO0iPwtn`
3. Pyspark Wordcount Example - `https://docs.ovh.com/gb/en/data-processing/wordcount-spark/`