REPORT- Documents All Pairs Similarity

In this article, we will discuss how to solve the " *Documents All Pairs Similarity*" problem, the purpose of which is to find all documents whose similarity exceeds a certain threshold. Documents are represented as vectors whose cells represent the TF-IDF of that particular document. Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus.

Note: Due to the performance of the machine, the number of documents selected will be equal to 750 and 1000, from the dataset 'BeIR/nfcorpus'. The threshold is equal to 90%. The number of logical cores of the machine is *8*.

The sequential problem uses the efficiency of the numpy library to compute the cosine similarity and then select the documents that exceed the threshold. The running time is equal to *[464 milliseconds]* for 750 documents and *[744 milliseconds]* for 1000 documents.

To solve the problem in a parallel way the *PySpark* library was used to create the *MapReduce* model and distribute the data. *MapReduce* facilitates concurrent processing by breaking the data into smaller chunks and processing them in parallel. Finally, it aggregates all the data from multiple workers to return a consolidated output to the application. The number of used workers is equal to $[1, 2, 4, 6, 8, 10, 12, 16]$. Here the code:

```
def pairs_similarity_spark(matrix, bd_array, T, sc, slice_):

    documents_rdd = sc.parallelize(matrix, slice_)
    # Map the RDD to term-document pairs
    pairs = documents_rdd.flatMap(lambda x: map_function(x,
                                  bd_array))\
                .groupByKey()\
                .flatMap(lambda x: reduce_function(x, T))\
                .collect()

    return pairs
```

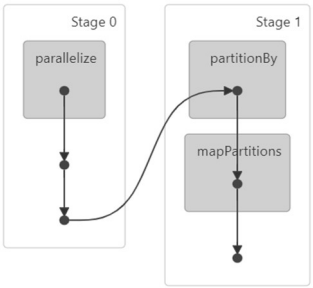
```
def map_function(id_document, boundary_arr):
    result = []

    id_ = int(id_document[0])
    document = id_document[1:]
    boundary = boundary_arr[id_]
    for term in range(boundary+1,len(document)+1):
        if id_document[term] != 0 and boundary!=len(document)
                                          +1:
            result += (term, (id_, document))
    return result
```

First we create the **RDD**(Resilient Distributed Datasets): — The dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. What data parallelism does is, creates parallelism from the beginning by partitioning the dataset into smaller partitions.

Our data will be divided into '*workers*2* 'partitions.

The first map transform each line of the RDD into the triple: *(Term, IdDoc, DocumentText).* Thanks to the fact that the matrix is sparse we can skip many terms, and reduce our data structure. The result of the first map will be passed to the spark function *GroupByKey().*



Then *groupByKey* operation takes the result of the first map and groups it using the key (term) and fed to another map. However map is a transformation and transformations are lazy, means that when we call some operation in RDD, it does not execute immediately. Spark maintains the record of which operation is being called (Through DAG). From the following image we can see the DAG VISUALIZATION. In Spark, a job is associated with a chain of RDD dependencies organized in a direct acyclic graph (DAG) that looks like the following:

```
def reduce_function(x, T):
    key = x[0]
    list_of_documents = list(x[1])
    result = []

    for i in range(len(list_of_documents)):
        for j in range(i+1,len(list_of_documents)): #to not
                                          have pairs
            d1 = list_of_documents[i]
            d2 = list_of_documents[j]
            union_terms = np.union1d(np.nonzero(d1[1]), np.
                                          nonzero(d2[1]))
            if key == np.max(union_terms):
                similarity = np.dot(d1[1], d2[1])
                if similarity >= T:
                    result += [(d1[0],d2[0],similarity)]

    return result
```

The second map is used to simulate the reduce. This is because the reduce must be associative and distributive. The second map takes as input the output of the *groupByKey* (*term – list of documents*) and calculates the similarity between the documents. However, to decide which reducer will have to execute it in order not to have multiple operations, it is checked that:

$$term = max(doc1 \cap doc2).$$

The result is a list of triples *(doc1, doc2, similarity)* and it is passed to *collect().*

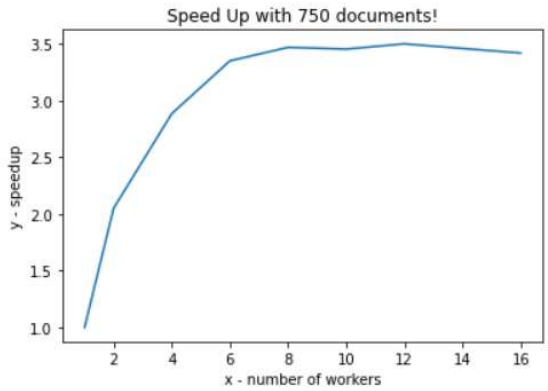
At the end of everything, the *collect* takes place. All the transformations are done in a DAG basis and the actions (here it's the collect()) is done at last using the original data, so that's why it might take time. Following the results of the computational time for each number of workers:

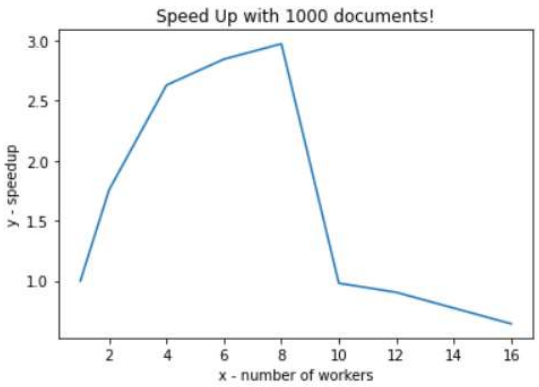
n.docs/n.workers	1	2	4	6	8	10	12	16
750 docs	36,76 min	17,92 min	12,73 min	10,96 min	10,59 min	10,63 min	10,49 min	10,74 min
1000 docs	58,26 min	33,17 min	22,18 min	20,50 min	19,61 min	59,43 min	64,41 min	90,59 min

The sequential problem far exceeds the execution time of the problem performed by pyspark, this for two reasons:

1) The data limit, PySpark would surely perform better with a greater number of documents and with several cluster-nodes.
2) LazyEvaluation by PySpark. It's not the collect() that is slow. Actually, Spark works on the principle of Lazy evaluations.

So to calculate the speedup more fairly we use the time on only *one worker*. As we can see we reach the highest speed up with the number of workers equal to the number of logical processors of the device.





Why it works better with 750 documents instead of 1000 documents?
Memory Limit. The computer can slow down massively if some program is using too much memory. Processes need random-access memory (RAM) to run fast. When you start a process (program), the operating system will start assigning it memory. But what if there's not enough memory to fit all of the data in it? If the program is trying to process more data than it has got memory, it will start to spend a lot of time saving and reading data from the disk instead. This is pretty slow. To make matters worse, your program isn't talking to the processor a lot anymore (it's busy scratching on the clay tablet, aka your hard disk). That makes the processor think you don't need it as much and it will start accepting other jobs, which puts even more strain on the available resources. The computation will probably finish at some point, but it will take a LOT of time.

Conclusion: In conclusion, pyspark is an excellent solution for processing lots of data, but you need to have the right machines (something I unfortunately don't have). As seen with fewer documents and more memory space, Spark is able to divide the work between the various workers and the partitioned dataset is able to be kept in memory. The problem lies in the final collect, but the code should be optimized more to reduce as much as possible the data to be sent to be processed by the collect. To reduce the elements sent to the collect we used the algorithm explained in class and the fact that the representation of the documents is sparse, many terms are set to 0 so the result of the map is reduced.

Most Frequent Implementation Problem: OutOfMemoryException (which unfortunately I could not solve). Tried to connect to GPU but with little attempt. To use the university cluster unfortunately I didn't have time between studying and other things to configure everything and find a hole to process the task. One reason for this error was because I was overloading the driver by adding too many workers for testing. Collect() operation will collect results from all the Executors and send it to your Driver. The Driver will try to merge it into a single object but there is a possibility that the result becomes too big to fit into the driver's memory. Using too many unnecessary workers will therefore only cause overhead and slow down the process.