VRIJE
UNIVERSITEIT
BRUSSEL

DISTRIBUTED COMPUTING & STORAGE ARCHITECTURES

MapReduce Algorithm: Practical Implementations in MrJob

Vitalijus Cernej
Seppe Lampe

January 10, 2021

Prof. Dr. Nikolaos Deligiannis
Xiangyu Yang
Yiming Chenn

engineering

Contents

1 Introduction

'MapReduce' is a programming technique which enables the parallel processing of large data sets in a distributed file system (DFS). The algorithm contains three types of building blocks, which can be chained together i.e., mappers, combiners and reducers. A mapper takes one line of input and generates zero to many (key, value) pairs for this input. Combiners and reducers perform operations on groups of (key, value) pairs with the same key and then yield again a (key, value) pair. A combiner directly takes input from one specific mapper, while a reducer can process the partial output from several combiners/mappers (Fig. 1). One of the most popular frameworks for MapReduce are Hadoop Distributed File Systems (HDFS). In this project, written for the course 'Distributed Computing & Storage Architectures' at Vrije Universiteit Brussel (VUB), we evaluate the use of MapReduce via the python library 'mrjob' for six different tasks.

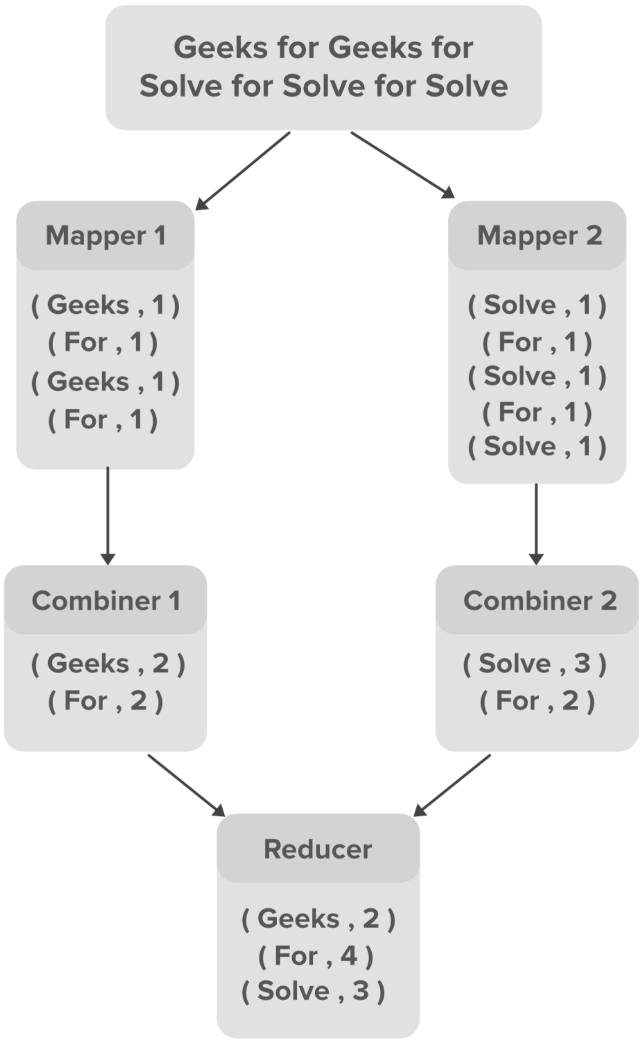

Figure 1: Overview of a MapReduce structure (adapted from https://www.geeksforgeeks.org/mapreduce-combiners/).

2 Objectives

2.1 Task 1 & 2

The first two tasks query data from the popular movie rating website 'IMDB'. The data set is contained in a tab-separated value (tsv) file, 'title.basics.tsv' and has the following structure:

[tconst titleType primaryTitle originalTitle isAdult startYear endYear runtimeMinutes genres]

Each row represents one entry, and the data set contains over 6 million entries. The first objective was to find the 50 most common keywords for all entries which have a 'titleType' of either 'movie' or 'short'. The second goal was to get, for each movie genre, the 15 keywords which are most common in the 'primaryTitle' of their respective entries. It was requested that the results of these tasks do not contain words with no meaning such as auxiliary verbs, prepositions and articles.

2.2 Task 3 & 4

The third and the fourth assignments operate on two comma-separated value (csv) files containing the retail sales of an individual company for the years 2009-2011. In total, there are 525461 and 541910 entries for the 2009-2010 and 2010-2011 data sets respectively.The files have the following structure:

[Invoice, StockCode, Description ,Quantity, InvoiceDate, Price, Customer, ID, Country]

The aim of the third job was to find the 10 customers which spent the most money in each of the retail years. While the purpose of the fourth exercise was to report the best selling product, once in terms of total quantity, and once in terms of total revenue for all retail years combined.

2.3 Task 5

The fifth task is performed on a 'json' file, 'arxivData.json', which contains meta-data on 20000 papers from the arXiv archive. Each entry contains info on the following fields:

[author, day, id, link, month, summary, tag, title, year]

Here, the goal is to find the most similar paper summary with respect to a given paper summary. This was achieved by applying cosine similarity, cosine similarity compares two vectors with each other and returns their normalized inner product. The vectors will be composed of the alphabetically ordered counts of the words present in the summaries.

2.4 Task 6

The final objective implements matrix multiplication via MapReduce. Two text files, which contain the matrices to multiply, are to be supplied as input.

3 Implementation

3.1 Task 1

The implementation of the first task contains two steps. In the first, the mapper 'mapper_get_words' receives entry by entry and only selects those where the 'titleType' matches 'movie' or 'short'. From the selected entries, the 'primaryTitle' is word tokenized via the natural languages toolkit (nltk) library and the words are subsequently tagged with their grammatical type. If the grammatical type does not correspond with one of the non-useful types such as articles and prepositions, and the word is not a stop word or 'untitled', then it will be passed on to the combiner as the key, value pair: *word, 1.* When a mapper is initialised, it is preceded by a small function which creates two variables containing the non-useful types and stop words, this ensures that the mappers have access to these variables, while not having to generate them for every entry.

The subsequent combiners and reducers 'sum_values' sum the counts for each respective word. The second step starts with a mapper 'mapper_None_count_word', which maps the *word, count* pairs to *None, (count, word).* As all values now have the same key, the subsequent combiners and reducers 'fifty_max_values' sort all value pairs based on the counts of the words and yield the 50 highest pairs. The job can be executed inline from the project directory with the following command:

python "1 IMDB/Task_1.py" "1 IMDB/title.basics.tsv" > "1 IMDB/Task_1.txt"

The results of the MapReduce algorithm will be written to 'Task_1.txt', alternatively this can also be omitted to have the program write the results to the terminal. The job can also be executed in a local cluster, to simulate a DFS via:

python "1 IMDB/Task_1.py" -r local --no-bootstrap-mrjob "1 IMDB/title.basics.tsv" > "1 IMDB/Task_1.txt"

Note that copy-pasting commands from this report into a terminal might lead to some formatting changes e.g., disappearing whitespaces, wrong type of quotation mark, etc. It is therefore advised for each task to copy-paste the commands inserted at the top of each .py file, as these are formatted correctly.

3.2 Task 2

The second program contains, just as the first, two steps. In the first step, the data is passed to a similar mapper as the one used in the first task. This mapper 'mapper_get_genre_words' firstly selects only those entries where the 'titleType' equals 'movie', it then again word tokenizes each word in the 'primaryTitle' and adds the corresponding grammatical tags. Subsequently, it iterates through the 'genres' and for each genre it will loop through the aforementioned words. Here the common stop words and words with an undesired type will be filtered out, those that go through this filter will be passed to the combiners as *(genre, word), 1.* The following combiners and reducers 'sum_values' sum up all the counts for every *genre, word* combination. Then, in the second step, the *(genre, word), count* combinations will be mapped to *genre, (count, word)* by the 'mapper_to_genres' mapper. Lastly, the 'fifteen_per_genre' combiner and reducers sort, for each key (genre) the values based on the count and select up to fifteen tuples with the highest count. The job can be executed inline from the project directory via the following command:

python "1 IMDB/Task_2.py" "1 IMDB/title.basics.tsv" > "1 IMDB/Task_2.txt"

The results of the MapReduce algorithm will be written to 'Task_2.txt', alternatively this can also be omitted, which will cause the results to be written to the terminal. The job can also be executed locally, to simulate a DFS via:

python "1 IMDB/Task_2.py" -r local --no-bootstrap-mrjob "1 IMDB/title.basics.tsv" > "1 IMDB/Task_2.txt"

3.3 Task 3

Task three is composed out of two steps. The first step, starts with a mapper, which obtains the customer, year, quantity and price for each transaction, 'mapper_year_customer_with_revenue". If the customer field is specified, it will pass *(year, customer), revenue* to the subsequent combiners and reducers, with revenue equal to quantity times price. The 'sum_value' combiners and reducers sum the values for each customer, year combination and yield *(year, customer), revenue* to the mapper of the next step. This mapper 'mapper_to_years', transforms the input to *year, (revenue, customer)* pairs. Lastly, the final combiner and reducer 'max_10_per_year' sort, for each year, the revenue, customer pairs based on revenue and yield the ten highest. The code can be executed in command line from the project directory via:

python "2 Online Retail/Task_3.py" "2 Online Retail/retail0910.csv" "2 Online Retail/retail1011.csv" > "2 Online Retail/Task_3.txt"

'Task_3.txt' will contain the results of the MapReduce algorithm, alternatively if this part is not included, the results to be passed to the terminal. The job can also be executed locally, to simulate a DFS via:

python "2 Online Retail/Task_3.py" -r local --no-bootstrap-mrjob "2 Online Retail/retail0910.csv" "2 Online Retail/retail1011.csv" > "2 Online Retail/Task_3.txt"

3.4 Task 4

The fourth goal, is implemented by yet another two-step job. The first step starts with a mapper 'mapper_code_with_quantity_revenue', which retrieves the stock code, quantity and price from each entry. If the stock code is non-empty, then *stock code, (quantity, revenue)* will be yielded, revenue equals price multiplied with quantity. The 'sum_quantity_sum_revenue' combiners and reducers will sum the quantities and revenues for each stock code and yield *stock code, (quantity, revenue)*. The consecutive mapper 'mapper_to_None' will transform these pairs into *None, (quantity, revenue, stock code)*. These values will be sorted, once based on quantity, and once based on revenue by the 'max_per_quantity_and_revenue' combiners and reducers. Which will yield only the highest pairs for each sorting. The outcome can be generated by passing in the following command into command line when located in the project directory:

python "2 Online Retail/Task_4.py" "2 Online Retail/retail0910.csv" "2 Online Retail/retail1011.csv" > "2 Online Retail/Task_4.txt"

The output will be contained in the 'Task_4.txt' file and, if desired, the program can also be executed in local mode via:

python "2 Online Retail/Task_4.py" -r local --no-bootstrap-mrjob "2 Online Retail/retail0910.csv" "2 Online Retail/retail1011.csv" > "2 Online Retail/Task_4.txt"

3.5 Task 5

The fifth task takes an extra argument in the command line, '--source_file'. This argument takes the location of a file, containing the text to find the most relevant summary to. The program consists again of two steps and starts with a variant of the normal mapper i.e., mapper_raw. This mapper does not go through the input line by line but rather opens a whole file at once. As the input data is in 'json' format, it is possible to iterate through this. Each iteration will contain all the meta-data of one specific paper. The mapper_raw, 'get_id_summary', yields *id, summary* for each paper in the database, afterwards the program proceeds to the next step. This step starts with an initialisation function for each normal mapper, here the 'source_file' is opened and its content is stored in a class variable. The 'get_cosine_similarity' mapper starts by passing the summary, along with the summary in the 'source_file, to the 'CountVectorizer' function of the 'sklearn' library. Before the summaries are vectorized they will be converted to lowercase, stop words and non-words will be removed and the remaining text will be tokenized and lemmatized. The resulting vectors will be passed to the self-made cosine similarity function, to eventually yield *None, (id, cosine similarity)*.

Note that the two mappers can actually be combined into one mapper. On an actual DFS, this would cause a bottleneck as for each input file all the mapping would be done by solely one mapper. By splitting this up into two steps, the count vectorizing and calculation of the cosine similarities can be distributed among any number of mappers and executed in parallel. Now, the mapper_raw does no longer form a bottleneck as reading in the data and generating the *id, summary* pairs is quite a small and lightweight task. The 'max_value' combiner and reducer will yield only the pair with the highest cosine similarity. To execute the program from the project directory, the following code can be entered:

```
python "3 Similar Paper Recommendations/Task_5.py" --source_file="3 Similar Paper
Recommendations/summary.txt" "3 Similar Paper Recommendations/arxivData.json" > "3
Similar Paper Recommendations/Task_5.txt"
```

Note that the usage of 'mapper_raw' along with '-r local' causes an error, more specifically *[WinError 2] The system cannot find the file specified*, will be raised, we have already spent considerable time trying to solve this issue, but no solution has been found yet.

3.6 Task 6

The final objective contains two steps and starts with a mapper_raw. Similar to the previous exercise, the first step only contains a mapper, the matrices are read in by the 'read_matrix' mapper_raw. Two environment variables 'matrix1' and 'matrix2' have been set on beforehand and initially contain empty tuples. When the two files are read in, these values will be filled in by the tuples *(name, (row size, column size))*. Where the column size of 'matrix1' will always be equal to the row size of 'matrix2', otherwise an assertion error will be raised. For each of the values in each of the rows in each of the matrices, the tuple *(matrix name, row index, column index, element)* will be yielded. Note that this implementation has the reasonable assumption that the two matrices have different names. In the second step, a mapper 'generate_tuples' will yield, for each entry coming from 'matrix1' *(row index, column), (name, column index, value)* for each 'column' between 0 and the column size of 'matrix2'. Similarly, for each input coming from 'matrix2', *(row, column index), (name, row index, value)* will be yielded for each 'row' between 0 and the row size of 'matrix1'. These mappers will take up the largest part of the execution time. The combiner 'combine_tuples' will simply combine the values for the same key in a list. This combiner has a negative impact on time performance as it also takes considerable time to execute. However, it does decrease the amount of data which is to be transferred between the preceding mapper and subsequent reducers by 1/3, which would be significant in an actual DFS. Before initialising the reducers, all data will be sorted in memory, and without this combiner, the data is too large to fit in the memories (RAM) of our system, which will provoke a 'MemoryError'. The reducer 'calculate_dot' will yield the dot product for each matrix element (row, column) separately, and makes clever use of the default sorting of values in MrJob. The task can be executed by entering the following code in command line from the project directory:

python "4 Matrix Multiplication/Task_6.py" "4 Matrix Multiplication/A.txt" "4 Matrix Multiplication/B.txt" > "4 Matrix Multiplication/C.txt"

4 Results

4.1 Task 1

Below, the five most mentioned words are list, along with their count rate. the full list of fifty words is documented in the 'Task_1.txt' file.

- love: 4351 times counted
- one: 3527 times counted
- black: 2221 times counted
- life: 1934 times counted
- two: 1885 times counted

4.2 Task 2

The complete results of the second assignment are stored in 'Task_2.txt', below the most popular word for each genre is listed, along with its count:

- Action: black, 213 times counted
- Adult: sex, 97 times counted
- Adventure: king, 64 times counted
- Animation: doraemon, 39 times counted
- Biography: life, 57 times counted
- Comedy: love, 590 times counted
- Crime: black, 123 times counted
- Documentary: one, 341 times counted
- Drama: love, 857 times counted
- Family: little, 93 times counted
- Fantasy: love, 36 times counted
- Film-Noir: johnny, 9 times counted

- Game-Show: champion, 1 times counted
- History: one, 35 times counted
- Horror: blood, 247 times counted
- Music: rock, 56 times counted
- Musical: love, 50 times counted
- Mystery: murder, 82 times counted
- News: echo, 8 times counted
- Reality-TV: british, 3 times counted
- Romance: love, 665 times counted
- Sci-Fi: alien, 93 times counted
- Short: blott, 1 times counted
- Sport: liverpool, 126 times counted
- Talk-Show: interview, 2 times counted
- Thriller: black, 152 times counted
- War: battle, 44 times counted
- Western: wild, 48 times counted

4.3 Task 3

'Task_3.txt' contains the full list of results, as demonstration the best customer of each year is listed below:

- Customer 18102 spent €41005.74 in 2009
- Customer 18102 spent €328605.6 in 2010
- Customer 14646 spent €270897.14 in 2011

4.4 Task 4

The outcome of the fourth objective is stored in 'Task_4.txt' and is listed below:

- Item '84077' was sold the most: 108545 times.
- Item '22423' had the highest revenue: €327813.65.

4.5 Task 5

The outcome of the fifth task is dependent on what is written in the 'source_file'. In our example, we altered some of the words from arXiv id '1705.07962v2'. Logically, our result is that arXiv id '1705.07962v2' is the most similar paper to our version, they have a cosine similarity of 0.854.

4.6 Task 6

The last task objective generated a matrix of size 1000 by 2000, in order to verify whether the generated matrix is correct it was compared against the outcome of the 'numpy.matmul' function, this is done via the adjoined 'Result Verifier.py' file. One matrix is subtracted from the other and the L2 norm is calculated from this outcome. This was equal to $\approx 2.3e^{-7}$, this small value is the accumulated result of small rounding errors caused by the limited precision on floating point numbers in python, both matrices can be considered equal.

5 Contributions

During this project the work was divided as the project moved along. Vitalijus started working on task 1, while Seppe started on task 2. However, there was a lot of back and forth discussion about the tasks as we tried figuring out functioning solutions together. Once task 1 and 2 were working correctly, the third exercise was started by Vitalijus and the fourth by Seppe, there was again some conversation about the two tasks. However, these two exercises took considerable less time than the first two, not because they are easier but because we understood the library a bit better by now. The fifth exercise was first implemented by Vitalijus and then reworked by Seppe. However, a number of things were re-used such as the function for cosine similarity and the CountVectorizer. The last exercise was written together as we went through the course lectures again.

6 Conclusions

This project has shown in practice how the processing of large amounts of data can be handled via a MapReduce framework in a DFS. During this project, we got valuable hands-on experience with the MRJob library, which allows to write MapReduce jobs in Python and run them on different platforms. Even though our code was never executed on an actual DFS, this project has given us many insights on what the best-practices are to create jobs for real distributed file systems.