Inverted index, the first step in the query engine

Victoria Torres Rodríguez[1]
[1] *victoria.torres101@alu.ulpgc.es*

Joel del Rosario Pérez[2]
[2] *joel.del101@alu.ulpgc.es*

Alba Martín Lorenzo[3]
[3] *alba.martin112@alu.ulpgc.es*

Álvaro Travieso García[4]
[4] *alvaro.travieso101@alu.ulpgc.es*

Jorge Langlenton Ferreiro[5]
[5] *jorge.lang101@alu.ulpgc.es*

Abstract

Living in the 21st century, where technology stands as one of our main companions for the rest of our lives, has led to tasks as simple as managing a library in applications that exponentially reduce waiting time. Search engines have revolutionized information retrieval to the extent that users expect near-instantaneous responses to their search queries. One of the main ways to organize and retrieve words is inverted indices. Therefore, an implementation based on this method was created, with the aim of minimizing the workload that someone might have to do filtering a large number of books. Dividing ourselves into work subgroups, once 3 implementations of this method were created in Python, benchmaring was used to take a final decision. The basic structure of these indexes were nested dictionaries, a list of references and a list of tuples. The load in memory once the indexes (KB) are stored and the execution time in seconds were the metrics that were taken into account, reaching the conclusion that indexes based on nested dictionaries are the ones that execute the least execution time and have the least memory consumption, since their serialization is almost direct.

Key words: *Execution time, Benchmarking, Inverted Index, Serialization, Python, Dictionaries, Json, XML, Search Engines.*

1 First contact in search engines

Search engineering, as it is known known, is based on a main component called indexing. Indexing is the process by which search engines organize information before a search to enable super-fast responses to queries. Searching through individual pages for keywords and topics would be a very slow process for search engines to identify relevant information. Instead, search engines (including Google) use an inverted index, also known as a reverse index.

As the main objective of the work, an inverted index has been created based on the use of python dictionaries and lists with different structures, in which we have verified that among the three implementations, the index based on dictionaries of dictionaries manages to obtain the best results. In this way, we have shown that even within these forms of search, not only the reading algorithms condition the effectiveness, but also the storage of the data and its structure is one of the main components that will determine said effectiveness. Thus concluding that a good use of the SOLID principles and polymorphism, the ability to make use of polymorphism and the good structuring of the inverted index, lead to better and more efficient results, thus reducing execution time and memory space.

2 Problem statement

The problem raised in the study is the need to streamline searches in text files as an online library, allowing the generation of data structures that store the metadata of multiple books, as well as data of the different words that appear in them efficiently. Thus obtaining in which books each of these words appear and what parts of each one are found. These data structures are the main object of study, since the main search engines are developed from them.

3 Method

3.1 Analysis and design

Prior to implementation, a class diagram has been created in StarUML to represent the fundamental objects of the system that will be part of the solution and the relationships between the elements of the system, in order to see what components the system needs and how they influence each other. For this purpose, the SOLID methodology and the polymorphism of the classes have been taken into account. The main class is a controller that will depend on the constructor of the inverted index (InvertedIndexBuilder) and the document reader (DocumentReader), which receives the path of a document in XML and deserializes it, as it can be seen in Figure 1. InvertedIndexBuilder needs a class that removes punctuation marks, capitalization and spaces, which is Tokenizer. It is also associated with the InvertedIndex class that creates a JSON file with the index and, in turn, the InvertedIndexStore class depends on InvertedIndex to write to the JSON file. DocumentReader needs a Document class and, in turn, this second one only exists with a Metadata class that initializes the document attributes. In addition, we have a DocumentBuilder class to build documents.

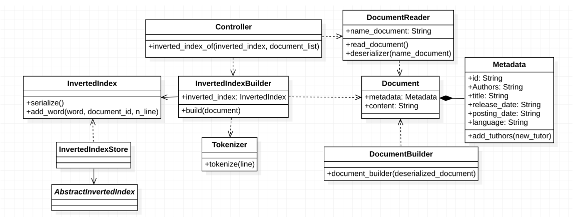

Figure 1: Class diagram for inverted index implementation

3.2 Test environment

Once the representation of the system in the UML class diagram is done, we start the implementation. The implementation is done in the Python programming language (version 3.7), an interpreted language widely used by companies around the world to build web applications, analyze data, automate operations and create reliable and scalable business applications. The tests will be run in the PyCharm integrated development environment (IDE), specifically for the Python programming language, developed by the company JetBrains. * Note that the tests will be run on an M1pro chip, with a 10-core CPU, a 16-core GPU and 16 GB of RAM, so the execution time of the same program under other conditions will vary.

3.3 Methodology

One of the kernels in the search types in this sector is the creation of the inverted index, the shorter its execution time, the more agile the searches to be performed. We have three different interpretations of how the inverted index should perform: the nested dictionaries, the reference list and the list of tuples. To evaluate each implementation we must measure different aspects of performance in order to establish which is best. A good way to evaluate is to measure this time on the three implementations created. Another interesting factor to check is the memory usage. Thus, the metrics we will use will be the execution time and the space occupied in memory by the use of dictionary, reference list and list of tuples.

4 Experiments

The tests that were executed and the results that were obtained in said analysis will be demonstrated. To evaluate an inverted index implementation, It has to be must measured different aspects of performance in order to establish which one is the best.

4.1 Experiment 1

Given a directory of between 12 and 15 books on average, we execute the main function, printing in seconds how long each one takes. Starting with the nested dictionaries, we will see that it takes a total of 2.0126 seconds, while the reference list takes 15.4927 seconds. This produces an acceleration of almost 8 times. Last implementation takes 2.2207 seconds, almost the same time that the first test takes, as we can see in Table I. Because of this difference, it will be taken into account only the two of all.

Type of index structure	Time (seconds)
Nested dictionaries	2.0126
List of references	15.4927
List of tuples	2.2207

Table 1: Benchmarking of the main function

4.2 Experiment 2

Progressively, different access to implementations and the time needed to make comparisons and select the best will be tested. In this case, it will access 100,000 random words and remove the books associated with that word in the inverted index to have a good sample. The nested dictionaries test needs 0.1080 seconds to access that number of words. The second test, the tuple list test, requires 0.3425 seconds. These results are organized in Table 2.

Type of index structure	Time (seconds)
Nested dictionaries	0.108
List of tuples	0.3425

Table 2: Benchmarking of the access of 100,000 random words

4.3 Experiment 3

The last experiment is the time it takes for the system to store the nested dictionaries and the list of tuple implementations and the space it occupies in memory will be measured. Running the nested dictionaries test, it took 2.5736 seconds to complete the process. Also, if the created file is checked, it occupies 14,559 KB of memory. Once it has finished, it checks to see that 3.8895 seconds was the time it took to store the implementation of the list of tuples. Also, the memory space occupied is 37,098 KB doubleling the other implementation as it can be seen in Table 3.

Structure Type	Time (s)	Memory Used (KB)
Nested dictionaries	2.5736	14,559
List of tuples	3.8895	37,098

Table 3: Benchmarking of storing the inverted index

5 Conclusion

In this study, the problem of speeding up searches in text files as an online library through data structures such as JSON is addressed. One of main contributions of our work is to express this task as a constrained optimization problem, and to propose methods to solve it based on the SOLID principles and polymorphism.

As mentioned throughout the study, three implementations of the inverted index creation have been developed: based on nested dictionaries, reference lists and tuple lists. Through experimentation, it was obtained that the structure that best adjusted to the development of the final product were nested dictionaries, reducing the execution time almost 3 times compared to the rest of the implementations and almost halving the occupation in memory once serialized. For serialization, JSON was taken as the main object, since its ease of conversion in Python was convenient. For all these reasons, the final product of this project relies on an inverted index structure based on nested dictionaries as the main search engine.

The main contribution of this study is based on the approach to search engines from a "junior" point of view, coming to understand first-hand how this type of engineering works and where its importance lies: in the index structures. As a first contact, it has been understood that indexing is one of the main and most common methods in this type of problem, allowing to appreciate and understand from an initial point of view, how to develop a search software product, starting with its core.

6 Future Work

Many different adaptations, tests and experiments have been left for the future due to the lackof time (i.e. experiments with real data are often time consuming, requiring evendays to finish a single race). For all these reasons, we would like to name certain aspects that would be taken into account for a better implementation of this study.

First of all, the lack of a persister is notable, as this program simply creates a folder where the inverted indexes of all the books parsed in one call will be saved. What's wrong with it? The lack of compatibility with future big data projects. The format of the index and its storage form are not fully compatible with it, this does not mean that they cannot be carried out, simply that it will be more expensive in the face of changes in the future. For this reason, the idea of creating a persister that saves the inverted indices in different folders stored in alphabetical order: a, b, c... is proposed. In addition, the save format would be more indicated csv, for this very reason. In the face of an exhaustive study of different books, the information is more "organizable" with csv type files and their variants.

Finally, it is worth mentioning that a small number of serialization structures have been tested due to lack of time, finally staying with JSON, although as mentioned above, there could be other more favorable formats.

References

[1] *Título de la referencia 1* Autor. Año y editor.

[2] *Título de la referencia 2* Autor. Año y editor.

[3] *Título de la referencia 3* Autor. Año y editor.

[4] *Título de la referencia 4* Autor. Año y editor.

[5] *Título de la referencia 5* Autor. Año y editor.

[6] *Título de la referencia 6* Autor. Año y editor.