Implementation of an Optimal Structure for Inverted Index

Jose Juan Hernández Gálvez [1], David Cruz Sánchez [2], Jorge Hernández Hernández [3], Juan Carlos Santana Santana [4]

[1]jose.hernandez219@alu.ulpgc.es

[2]david.cruz107@alu.ulpgc.es

[3]jorge.hernandez12@alu.ulpgc.es

[4]juan.santana176@alu.ulpgc.es

October 2022

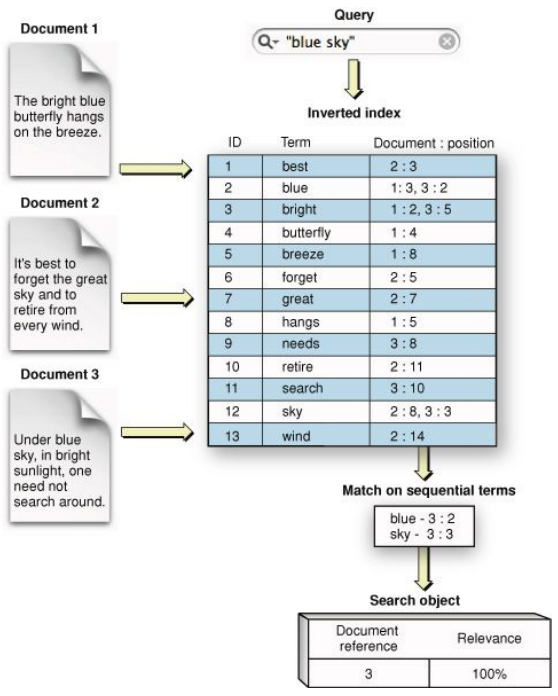
Abstract

The volume of documents uploaded to the internet is ever increasing, therefore, it appears the necessity of fast full text search while minimising the increment of processing when a new document is added. The data structure at the core of large-scale web search engines is the inverted index, which is essentially a collection of sorted integer sequences called inverted lists. The challenge we proposed ourselves was to try and implement our own structure of inverted index with the purpose of reaching a solution as optimal as possible. However, there is not a right design when it comes to optimization since we are selecting the best solution from all the feasible ones we are capable of coming up with. Due to that reason, we have developed several implementations using python and tested their respective build and consult times along with their memory usage and performed some bechmarking with the data. In the end, we came to an implementation which outperforms the others in both memory usage and execution time.

Keywords— Benchmark, python, inverted index, query, document

1 Background

There exists two type of indices based on the directionality. One takes you forward through the index (the one we are most used to), and the other takes you backwards (the inverse) through the index. In terms of information retrieval, the two types are identical, it's just a question of what information you have, and what information you're trying to find that differentiate them. Then, the forward index stores the document name as index and words as mapped references while inverted index stores the words as index and document names as mapped references. When it comes to building the index, the inverted one is slow as each word has to be checked before preparing the index while the forward is fast, as keywords are appended when found. In terms of querying, this is where inverted index has a huge lead against the other one since it is much faster than the other letting the user find all occurrences of a word within millions of documents stored in a database.


Document 1
The bright blue butterfly hangs on the breeze.
Document 2
It's best to forget the great sky and to retire from every wind.
Document 3
Under blue sky, in bright sunlight, one need not search around.
Query
"blue sky"
Inverted index
Match on sequential terms
Search object

ID	Term	Document : position
1	best	2 : 3
2	blue	1: 3, 3 : 2
3	bright	1 : 2, 3 : 5
4	butterfly	1 : 4
5	breeze	1 : 8
6	forget	2 : 5
7	great	2 : 7
8	hangs	1 : 5
9	needs	3 : 8
10	retire	2 : 11
11	search	3 : 10
12	sky	2 : 8, 3 : 3
13	wind	2 : 14

blue - 3 : 2
sky - 3 : 3

Document reference	Relevance
3	100%

Figure 1: Inverted Index Example

The inverted index is a common technique used for enhancing query performance. For text-based search, an inverted index consists of a list of all the unique words appearing in a document collection, and for each word, a list of identifiers for those documents that contain the word as we can see in figure 1. An inverted index is usually implemented as a hash map: the key is the word and the value is an array of document identifiers. If the array only contains the document identifiers, it is called a "record-level inverted index." If the array also contains the location of each word, then it is called a "word-level inverted index"[1], being this the one that we want to implement. Because of the massive number of documents indexed by such engines and stringent performance requirements imposed by the heavy load of queries, the inverted index stores billions of integers that must be searched efficiently[2]. For the implementation, we chose Python as the programming language since it is one of the most important programming languages according to the **TIOBE index**.

2 Methodology

Since our purpose was to optimize the inverted index, we had to consider different implementations so we were able to compare their respective KPI's to reach a conclusion. The implementations we did were based on python dictionaries which had words as its keys. As their values, we contrived different structures to save the document ids and the word's positions within document.

The first one uses a list of tuples to store the occurrences of a word in a document, each tuple has the book id and the position in which it appears in the document.

- Structure: {word: [(bookID, position), (...)], (. . .)}

The next implementation uses dictionaries to store the occurrences, each one has a bookID as key and a list of positions in which the word appears in the document as value.

- Structure: {word: {bookID: [position, (. . .)], (...)}, (. . .)}

The last implementation uses a list of objects, each object has a bookId and a list of the positions in which the word appears in the document.

- Structure: {word:[Object(bookID, [position, (. . .)]), (. . .)], (. . .)}

Memory	512 gigabytes
RAM Memory	8 gigabytes
CPU	Intel core i5 2.9 GHz
Graphics	Integrated Intel Iris Graphics 550
Operating System	macOS Catalina

Table 1: Hardware Specifications.

Depending on the hardware the results may vary, but we are going to do the benchmarking with the hardware of the table 1.

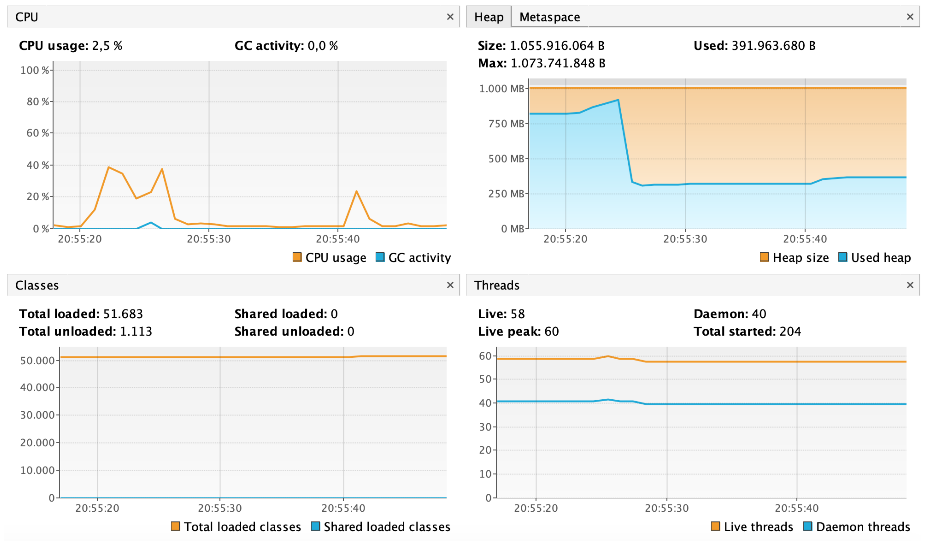
3 Experiment

Since our main objective was to find the optimal solution we had to run several tests in order to check the performances of the different structures we had. The experiments were done using unit tests in python. We tested the two main functionalities, building the index and querying some words with nine different books, hundreds of times. After those tests, we were able to tell their respective performances and execution times.

Execution Times (seconds)	List of Tuples	Dictionaries	Objects
Build	0.1475	0.1282	0.1995
Consult	0.0047	0.0051	0.0050

Table 2: Table of Execution Times in Seconds.

As we can see in the table 2, the fastest building time is the dictionary one even though it is not by far. On the other hand, the implementation with the fastest consult time is the list of tuples, being the difference between them minimal. If we take into consideration the possible dimensions of the problem, we could say that the build time is something to take more into consideration than the consult time since having to index millions of documents would take quite a long time. However, building the index is an action that is going to be performed not that often compared to consulting since, in a minute, the number of consults could be enormous while there could be no necessity of building the index again since no new documents will be added. Once taken all this into account, we could say that we should give more importance to the structure that allows us to minimise the consult times rather than the build times, which is then the list of tuples structure. However, the final conclusions will be decided along with the results of computer performance in terms of memory usage and CPU load.

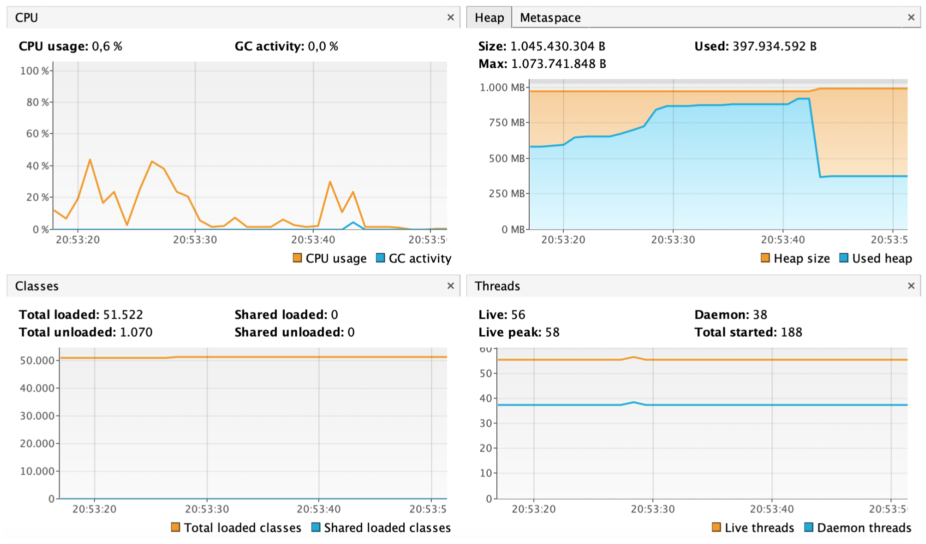

CPU usage: 2,5 % **GC activity:** 0,0 %

Size: 1.055.916.064 B **Used:** 391.963.680 B
Max: 1.073.741.848 B

Total loaded: 51.683 **Shared loaded:** 0
Total unloaded: 1.113 **Shared unloaded:** 0

Live: 58 **Daemon:** 40
Live peak: 60 **Total started:** 204

Figure 2: VisualVM Monitor of List of Tuples Implementation



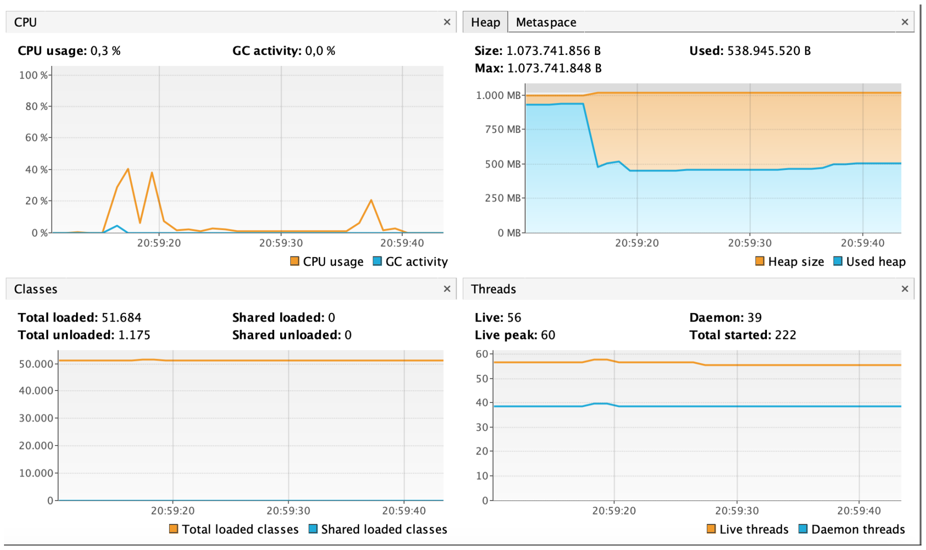
CPU usage: 0,6 % **GC activity:** 0,0 %

Size: 1.045.430.304 B **Used:** 397.934.592 B
Max: 1.073.741.848 B

Total loaded: 51.522 **Shared loaded:** 0
Total unloaded: 1.070 **Shared unloaded:** 0

Live: 56 **Daemon:** 38
Live peak: 58 **Total started:** 188

Figure 3: VisualVM Monitor of Dictionaries Implementation



CPU usage: 0,3 % **GC activity:** 0,0 %

Size: 1.073.741.856 B **Used:** 538.945.520 B
Max: 1.073.741.848 B

Total loaded: 51.684 **Shared loaded:** 0
Total unloaded: 1.175 **Shared unloaded:** 0

Live: 56 **Daemon:** 39
Live peak: 60 **Total started:** 222

Figure 4: VisualVM Monitor of List of Objects Implementation

If we start comparing the different implementations, if we focus on CPU usage, we appreciate that there are no major differences between them, which is expected since we are not using the CPU to its fullest by doing parallel programming for instance. From what we could tell, the dictionaries implementations put a higher load in the CPU compared to the others if we see the figure 3. In terms of memory usage, we notice in the figure 2 that the list of tuples starts by using almost all available memory (around 850 MB) but, as the execution continued it dropped to almost 250MB, keeping at that rate throughout all run. A worse but similar behaviour is shown by the object implementation as we can appreciate in figure 4 but with higher values since used memory at a point was around 900MB and then when it dropped, it kept its value at around 500MB. If we take a look at the dictionary performance shown in figure 3 we can clearly state that this is the worst implementation memory wise since its available memory kept decreasing throughout the execution and it never really stayed at low levels, indicating that it requires a voluminous amount of memory.

4 Conclusions

As we have seen, optimizing the inverted index is a huge challenge which could have many possible solutions. In this paper, we proposed multiple structures using python dictionaries and, to choose the optimal structure, we needed to define which were the indicators. In this case, we took into consideration the CPU usage, memory usage and the respective times in seconds each implementation needed to build the index and do a consult.

From all the possibilities we came up with, we decided that the tuple list inverted index is the best out of the three, as it is the least computer demanding and performs well in the build execution and in querying time. Also, it uses natural python types in contrast to the object implementation and it does not overuse the hash function, oppositely to the dictionary approach.

5 Future work

Many different adaptations, tests, and experiments have been left for the future due to lack of time (i.e. update the inverted index with new documents, test different types of inverted index persistance). Future work concerns deeper analysis of particular mechanisms, new proposals to try different methods, or simply improve our code.

For further development, we have decided to change the programming language from python to java, therefore migrating the whole project from one programming language to the other. The main reason for this language switch is due to the fact that dynamic (Python) versus static (Java) bindings. Every time Python wants to call a function, it has to take the function name, as a string, and lookup that string in a dictionary to find the actual function to call. It has to do this every time, because every object might be patched with different functions at runtime. Java, on the other hand, can very quickly find the address of the function to call by simply looking in the Nth slot in it's virtual method table. Both Python and Java execute on top of simulations of real computers, but the static typing and sophisticated tools for Java make it much much faster at run time than Python.

This project is highly scalable, for that reason, there are a lot of modules left that we can still implement. In the near future, we are planning on implementing a crawler, to automatically download documents from different websites and create the datalake we are going to use to build the inverted index with. Also, we want to develop a query engine, to retrieve information from the datamarts we have previously built.

References

[1] Yan Huang, Xiaojin Li, Guo-Qiang Zhang (2021). ELII: A novel inverted index for fast temporal query, with application to a large Covid-19 EHR dataset. Journal of Biomedical Informatics.

[2] Pibiri, G. E., & Venturini, R. (2020). Techniques for inverted index compression. ACM Computing Surveys (CSUR), 53(6), 1-36.