# Web Information Retrieval Project:
# Dynamic Indexing

**Written By: Elad Musba (cs username: eladmusba)**
**Track: Implementation**

## Abstract

One of the modern world most fundamental properties might be considered the ever-increasing ever-changing huge amounts of data. Big data is therefore a crucial aspect when one considers indexing it. While static indexing is the best way to start studying indexing large data, in reality most of the time we deal with constantly changing datasets, which has its own new demands: we need to make changes of data possible and yet keep the index structure, disk space, time to update and time to query minimal.  In this project I have implemented two methods to approach dynamic indexing: Simple-Merging and Log-Merging. I explore the trade-offs between the two methods mainly by comparing building/update time, query time and merging time in a presumed typical real-life scenario.

## Introduction

In this section I will describe the topics related to *dynamic indexing*. First, we will briefly go over the index building problem and solution in general lines. Then we discuss static indexing and why we should use dynamic indexing. Then I describe the Simple and Logarithmic approaches of building and merging indexes. In the end of this section there are simplifying assumptions and clarifications.

### *The Index Problem and Static Solution*

First let's lay out the basics of what data indexing means, and revisit some of the terms. An Index is a querying mechanism that answers the question of where a certain word, or even more generally a token was mentioned. The "where" part is best generally described as a document, and the structure that holds the documents that a word is in is a postings list. For example, if we want to look for information about a Binary Heap (query word) in an algorithms book (dataset), it would be simple and quicker to find information about it in the *index* of the book, pointing (postings list) to *pages* (documents) about it. A web search engine is a dynamic example of an index.

One might think about the following naïve solution for the indexing problem: Read the whole dataset and fill-in a table of rows of words and their increasing list of the documents that they were encountered in. Write the table to disk for later access. Like any other computing problem, the naïve solution works fine, until we push its space and time restrictions.

The time to look up a word has to be fast therefore a linear lookup is off the table. For example, a dataset of 10,000 amazon reviews of average 100 words per review holds ~40,000 different words (most are not grammatically correct words). To combat this, a good start would be to sort the words and implement a binary search method.

This hints at the next problem: we know that it is impractical to sort the words together with the lists they point to: the lists are sometimes huge and have a substantial variance in size. A solution to this is: don't sort the words with their lists, use pointers. This brings us to one of the fundamental conclusions in indexing: separate the dictionary – the words lookup mechanism - from the postings list. The structure in which one would typically store the latter is also called an Inverted Index.

Now is a good time to talk about computational space and memory. The inverted index in the naïve solution is directly proportional to the size of the whole dataset. We would like to minimize that. A common way to address this is as follows: Let us assume all the documents are numbered somehow with some document IDs (henceforth docID). We may notice that storing numbers as-is is wasteful. We can sort the docIDs and calculate the *gaps* between them for each postings list. Having to store gaps which are always numbers lower than the absolute docIDs might save us additional space, because we have to use more bytes for larger numbers. In order to exploit the low numbers of the gaps we need to use Variable-Length Integer encoding. The specific method I chose is detailed in **Methods**.

What we get so far is all these compressed numbers of postings lists of all the words sequentially in a file, and keep pointers and pointer lengths together with their words in the dictionary. This is enough for the *static* approach for the problem as it only deals with one input raw-data. There is no way to add documents to an existing index if our method entails going over all the data. Real-life scenarios might need an idea of dynamic indexing: being able to add and remove data (documents).

## *The Dynamic Solution*

The dynamic approach deals with mainly three fronts: data addition (or insertion), deletion and the occasional merging of indexes. For addition, the general approach is to build another index with the added data. As the number of those might increase infinitely, we will also need some method to merge all those added indexes, to save query time and disk space. Therefore, merging is also part of the solution, aimed to rebalance query time and disk space after some number of insertions. The way and timing in which we merge indexes is what this project is about.

For deletion the basic premise is: don't actually remove data. The reason for this is clear when one tries to think about deleting a docID from a postings list, in a tightly compressed inverted index file, and the need to also update the pointers in the dictionary file. If we are using a Front Code implementation for the dictionary part it gets even worse when deleting a document that also deletes a word: this word would have to be taken out of a block of words that should be of a constant size. The solution is therefore to keep somewhere all the deleted docIDs, and take this into account when answering queries.

## *Simple-Merging*

The simple merging approach may also be described as a manual approach. First, we build a main index from scratch (that should be of a considerable size). Then, for each addition of data, another sub-index is created in the same way the main index was created. Queries are done to all indexes and a union of the results is returned. When too many indexes are added – it could be that queries are too slow now or the sub-indexes data is larger than the main index - we will merge all the indexes (main and sub) to one big index.

For deletion – it is done here the same way as in the Logarithmic method next: by keeping a file containing all deleted reviews, adding to it when deleting and filtering by it when querying and when merging. Notice that merging is a great time to actually remove those docIDs, as all the indexes are being rewritten (and therefore reread) to one index anyway.

## *Logarithmic-Merging*

If the Simple method could be described as manual, the Logarithmic method (Log-merging) is more automatic. The basic notion of the method is as follows:

1. Let us define *n* as the maximal size of the temporary index (in-memory index).
2. While documents are fed into the index writer:
    a. If the temporary index size i.e. the summing of the lengths of all the postings lists currently in the index is smaller than *n*:
        i. Add the index data to the temporary index.
    b. Otherwise, write the temporary index to a directory named “TEMP”.
        i. If there is no index directory named “0”, TEMP is renamed to “0” representing an index write of size *n**2^0.
        ii. Otherwise, we merge TEMP and 0 to a directory named “1”, representing a write of size[1] *n**2^1. This merging goes on until we have an index directory that does not have another directory with the same name.

Theoretically, this algorithm resembles the Binomial Heap data structure. Practically though, merging was not done this way: notice that the amount of disk writes in this approach is unnecessarily higher than it needs to be. The way it was implemented in this project is similar to how ones count in binary: whenever we add a bit of 1 to a bit of 1, we write zero and carry the 1 to the next bit. We usually do this not for each bit like the algorithm above, but for all the first adjacent bits that are 1 at once. So, in our case merging would be done for all consecutive folder names: if there are directories with names TEMP, 0, 1, 2, 5 all TEMP, 0, 1, 2 directories would be read from and merged into an index with directory name “3”, resulting in two index directories 3 and 5.

This method has some substantial advantages:

1. As we merge bigger indexes, we do it less frequently which might help merging time. Also, the number of sub-indexes in this method is not linearly increasing as the simple-method: whenever we have indexes of total size roughly *n**2^*i*, *i* is a natural number, it results in one index directory again, rebalancing the query time.
2. Insertion or addition of data here is just running the same algorithm again. Continuing to add more data is almost the same as reading one big file of it all, as long as the program keeps running (for that in-memory index).
3. On the practical side, log-merging makes decisions automatically regarding merging with a fairly wise merging trigger. This is a big advantage for the implementor and the maintainer of the program.

In Conclusion, we can see that the Log merging method has convincing reasons to be used in dynamic indexing.

## *Assumptions and Clarifications*

The following are mainly related to changes from the exercises. Most points here are mildly affecting the project’s purpose.

[1] Size is defined as half the sum of the lengths of all postings list in an index i.e. the inverted index of a word consists of docIDs and the frequencies of that word in them (as I implemented in the exercises). It really doesn’t matter that it is not exactly the size of the inverted index file it just needs to be proportional to it.

Regarding queries, product queries were discarded. Word queries of “number of mentions” and “number of reviews” are now all calculated from `getReviewsWithToken` output by summing on the result. Keeping those queries as separate files for a O(1) query time was not compatible with the deleting operation. [2] Review queries are still functional, including collection queries (e.g. total number of documents) so a language-model and vector-space query processing can work with this project. In any case, only the `getReviewsWithToken` was tested in the experiment below, as this is the main idea of an index.

In terms of environment, the experiment was conducted on a personal computer running windows 8.1 with 8GB RAM and a 2.2GHz CPU. In any case, stack size was limited to 1GB.

Regarding deletion, it is done by a list of docIDs (and not, as explained in footnote 2, by the documents’ data itself). docIDs are infinitely increasing as they are added to the index. This means that a document can’t change on the same ID but a deletion of the old docID and addition of a new document has to be done.

Words of length longer than 127 letters were completely discarded (i.e. also in review size) as they are barely significant for the scope of the project and introduce some troublesome aspects when included.

## Methods

In this section I will go over some additional details regarding the specific implementation of the methods and the project, mainly things I haven’t specified yet.

The raw data was taken from Stanford’s Large Network Dataset Collection (http://snap.stanford.edu/data/index.html) of Amazon movies reviews. So, from now on reviews and rIDs will be our documents and docIDs.

Next is a more specific description of the three parts constituting an index directory:

1. Dictionary – as explained earlier the dictionary maps words to pointer to their postings-lists. I have implemented Front Code dictionary of 8 words per row. Each word holds 4 values: Length of word, length of prefix of the word relative to the first word in the block, pointer to postings list in Inverted Index file and the length of this list. Each group of 8 words holds a pointer to the first word in the concatenated string file.
2. Concatenated String – holds a concatenation of words or suffixes of all words in the index. This saves disk space for storing words.
3. Inverted Index: Holds the postings list of a word: all the rIDs and frequencies of a word in those reviews. It is sorted from lower to higher rID. First the rIDs are listed using gaps and then their respective frequencies without gaps[3]. Both were also compressed using Length-Precoded Variable-Bytes Integer representation. This file is the largest as it is proportional to the size of the raw data.

[2] The alternative of storing this data in another file in construction is impractical for the deletion functionality: there is no efficient way to update a token’s number of mentions after deleting some document because the only way to know this token is in this document is to look for the token's postings list. So, this would result in reading all the posting lists even for just one deleted document. As for querying time during query processing (ex3) I have found that in both methods of query processing the whole postings list has to be queried any way.
One may also suggest deleting not only by docID but also by the document text itself. I have decided this is not realistic, especially if we start considering real websites with huge amounts of text.

[3] Since frequency is internal to a review and those have 100 words in average, frequencies will usually be equal or lower than a byte in Length-Precoded VarInt which is 64.

In the Simple method, *External Sort* was used for building the index to be able to scale with larger input.

Both methods were using the same merging logic: A Merging Moderator object is created to hold all the indexes' reader objects in a queue based on their first word and its inverted index. While there are still more words to write, the moderator takes the word of all indexes with the lowest lexicographical value. When the same word is the lowest in more than one index, it takes the one which has the lowest first rID in its postings list. [4] This goes on until all readers have finished reading all their words (and inverted index with them).

### *Deletion*

Both methods were also using the same deletion mechanism: Deletion is done by feeding the index writer a list of rIDs to delete. These rIDs are written to an invalidation file with the same as above VarInt compression. Also, a flag marks that the invalidation file is *dirty* so that time is not wasted when querying with no deleted rIDs. When the index is queried and the invalidation file is dirty, it filters the results with the deleted rIDs. When all the indexes are written to a merged index (i.e. merging), all the deleted values are dropped and the invalidation file is cleared and marked as *not dirty* again.

This results in a slight inaccuracy regarding the index size for the matter of merging in the Logarithmic method. The answer is the same as in footnote 1: it should just be proportional to the index size. In any case, some ways to rebalance this were used such as : sampling rIDs for deletion uniformly and not deleting too many reviews per delete as to not give it too much weight (because the Simple method is not affected by the number of deletions at all in the experiment I planned below but the Log method may do a lot of merges without increasing the size of the index if deleting too much).

## Experimentation

### *Experiment Design*

In this section I will explain the experiment designed to compare between the Simple and Logarithmic methods. Most of the details here (number of insertions, deletion and when to merge) stem from what I perceived to be the common case.

The experiment was conducted on two scales of input: 100,000 reviews and 1,000,000 reviews. Each method was run on each input. The Log method was additionally run on temporary indexes of sizes 2^X where X is in [10,13,16,19,25][5].

The experiment will compare between query time, construction (build) time and disk size[6] of the two dynamic indexing methods, given the same input sequence. All the results are written to a file named "LOG.txt". The experiment sequence is as follows:

1. Build from scratch on 40% of the total input review number.
    a. Measure the execution time.
    b. Query 50 words[7] and measure the average time for a query.
    c. Measure disk size.

[4] Since rIDs are monotonically increasing, we are guaranteed that once the first rID of a word in an index is the lowest, so would also be the rIDs following it in the same index. In other words, the entire postings list is minimal just by its first value.
[5] Regarding how the different sizes of temporary index where chosen, I have picked a range with significant results. Therefore, the temporary index is not too low and the scale goes in +3 exponentially.
[6] Only for the 1,000,000 part, as the results are not very surprising or interesting.
[7] The unique set of words for each experiment were gathered beforehand in a file named 'words.txt'

2. Then, for each insertion:
    a. Insert an input file of the size of 250 or 2,500 reviews (according to the input scale) meaning there are 240 insertions of data after the first build.
        i. Measure the execution time.
    b. Delete 10 random non-repeating rIDs.
    c. Query 50 words and measure the average time for a query.
    d. Measure disk size.
3. For Simple merge, merge after all the insertions and deletions and:
    a. Measure merging execution time
    b. Query 50 words and measure the average time for a query.
    c. Measure disk size.

## *Hypothesis*

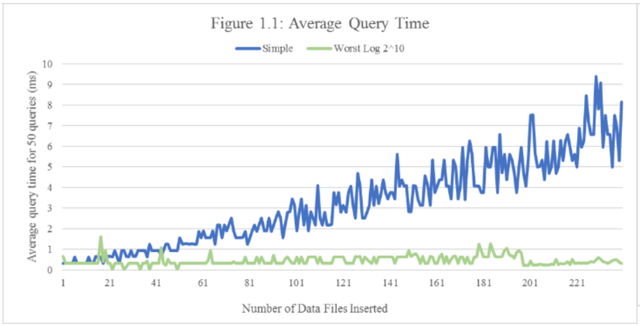
Log-Merge will get better results in query time on average, because the number of indexes to query is increasing logarithmically, and also might go to 1 after a big merge. The simple merge number of indexes will increase linearly and so will the average query time, but we expect the query time after merge to be much lower again. I also expect that the total runtime of the Simple method will be about the same as the Log method. I would also expect that a merge after the Simple run would drastically improve disk space utilization.

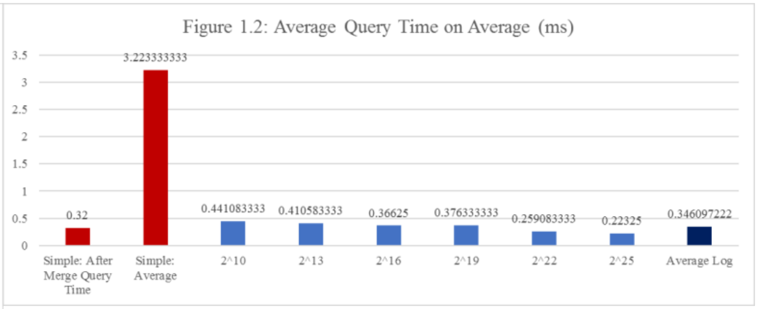
## *Results*

Note: In the following charts, "2^X" stands for the *temporary index size of the Log method* and "Average Log" is the average over the results of all temporary index sizes of the Log method (where X is in [10,13,16,19,25]).

### *100,000 reviews*

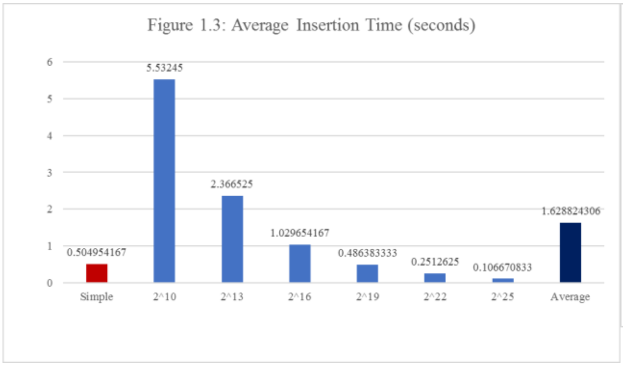
*Average Query Time*

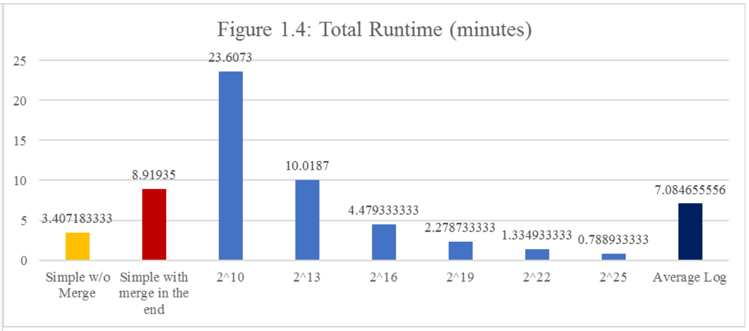




On Figure 1.1 we can see the Simple method did much worse than even the worst Log method. We can see that just after 50 files, the two plots are not touching even for the occasional peak of the Log method. For all insertions, we can see in Figure 1.2 that the worst Log method is ~8 times better on average than the Simple method, and that all Log methods are safely below the 0.5 milliseconds. We can also clearly see the benefit of merging on the Simple method when considering query time with it getting 10 times better after a single merge.

*Construction Time*



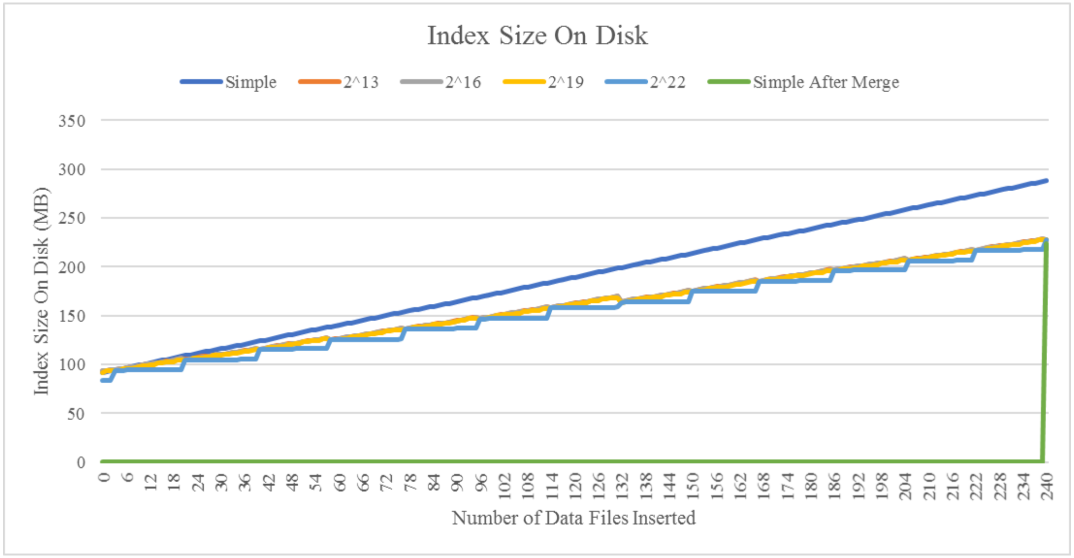


In Figure 1.3 we can see the price of using the Log method without hyper-parameter tuning: average insertion time of 3 out of 6 Log runs were worse than the Simple method, with the worst Log being 10 times worse. On the other hand, we see the Log method can still beat the Simple method with the right temporary index size where the largest got results 5 times better than the Simple Method. In Figure 1.4 we see similar trends in total runtime.

With this set of experiments, I should note that the 2^25 did not write even one index to the disk (and 2^22 wrote only 1) which obviously explains how these have gotten such good results such as far better total runtime. That being said, the 2^19 run had along its run a maximum of 4 indexes – so we know it definitely wrote indexes and merged at least 2^4 times – and still got better results than the Simple method – even without merging in the end. These artifacts are mitigated in the next 1 million reviews experiment, as all Log methods write (and merge) more than one time.

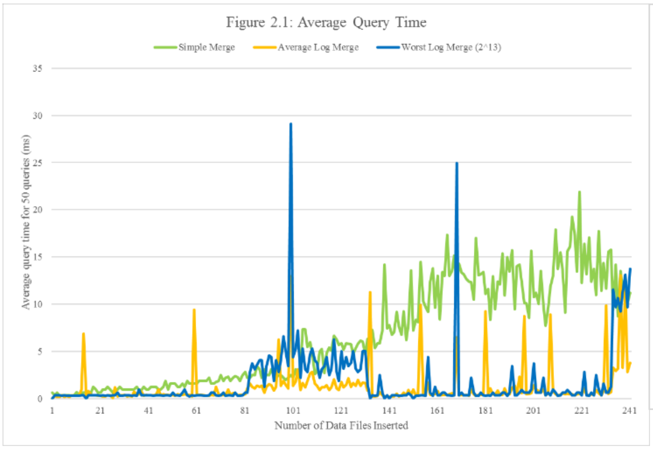
## ***1,000,000 Reviews***

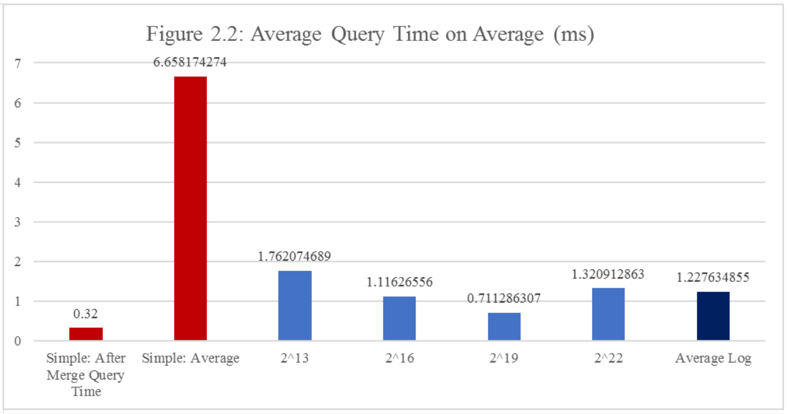
*Index Size on Disk*



We can see that all Log method plots behave almost identically regarding disk space. The main reason I wanted to test disk size is to compare it to the Simple method after merge. We see - to my surprise I must say - that a merge on the Simple index did not get better disk space utilization then the log method. We can see (only) a 30% increase in disk size when comparing the Log method to the Simple-before-merge method.

*Average Query Time*

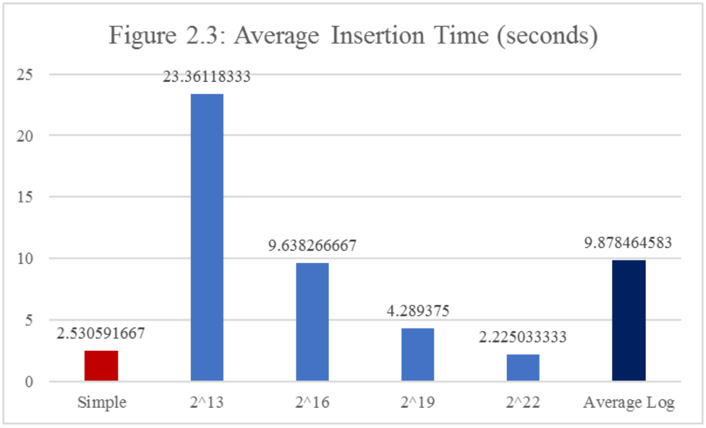




First, to clarify, this time not all temporary index sizes could be used. The 2^10 one took way too much time to finish (It was canceled mid run) and the 2^25 one has reached a heap overflow. I believe we still have a good range of results though.

Figure 2.1 shows us that this time the Simple method is only starting to create a serious gap around the 100th insertion. Figure 2.2 comparison is similar to Figure 1.2 with a difference that now the Simple merge outperforms all the Log methods after merge.

*Construction Time*



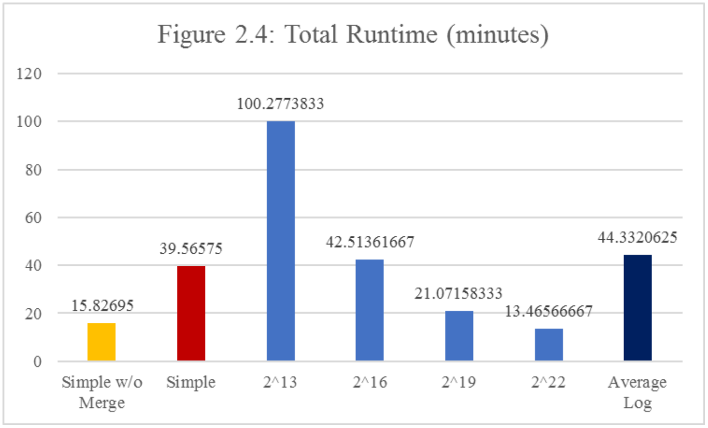


Figure 2.3 and Figure 2.4 draw the same conclusions as Figures 1.3 and 1.4: in the Simple method we might save time to execute insertions, but if we want to merge for better query time the Simple method becomes less attractive when comparing total runtime. For example, the 2^22 Log method beats the Simple method through all comparisons given here (although average query time is worse than the Simple run after being merged, total runtime in this case is much better).

## Conclusions

We can see that the Log method generally outperforms the Simple method if we are comparing by average query time of one word and in some cases also in insertion time and total runtime (and in my subjective opinion in technical maintenance and flexibility). Also, both methods don't vary much on disk space utilization. The trade-off lies within construction or insertion time, where the Simple method usually had better performance without merging. One might suggest that throwing in more merges in the Simple method might have gotten us more conclusive results in total runtime and maybe even better query time. I was also surprised to see that Log merge in the best cases still had equal or lower total runtime as I thought merging will incur more overhead.

As in any experiment the conclusions are only relevant to what was tested. Search Engines and other indexing implementations usually add much more meta-data for optimizing queries with runtime and best matching results. Things such as page ranking based on linking for example might add such overhead that we would want to merge as sparingly as possible, which may give priority to the Simple method or even for other ways to implement a dynamic indexing system.

## Resources

A much better-quality video and code examples for larger inputs can be found in the link below:
https://drive.google.com/drive/u/1/folders/1UjWqVZGhx3iKatM2-cwR1G9J1ZF28PuD

The link was set to be open to everyone, please contact me if there is a problem with it:
elad.musba@mail.huji.ac.il

Please have a look at the "How to run the code.pdf" file for running any of the code examples.