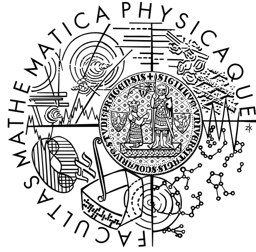


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Dávid Javorský

High-performance inverted index database

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Miroslav Kratochvíl
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In …….. date …………   signature of the author

I would like to thank my supervisor RNDr. Miroslav Kratochvíl, for his invested time, endless patient guidance and invaluable insights during my work on the thesis.

Many thanks also belong to my family and friends who support me all the time.

Title: High-performance inverted index database

Author: Dávid Javorský

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: The goal of this thesis is to implement an inverted-index database software that provides improvements in handling raw non-textual data, which is beneficial for several areas of research. The main internal structures of the library are designed to be cache-oblivious, also aiming to reduce the size of stored data. This thesis includes an overview of common inverted index implementation methods and describes retaled structures in a suitable cache-based model. This resulted in improvements of compression ratio, and performance similar to currently available highly optimized databases. The benchmark conducted on cheminformatic data has shown that the resulting software is applicable as an immediate, efficient replacement of the storage back-ends of specialized molecule databases.

Keywords: inverted index, database, cache-oblivious algorithm, feature search

Contents

Introduction **3**

1 Cache-efficient inverted indexes **5**
1.1 Ideal cache model . . . 6
1.2 Inverted indexes . . . 7
1.2.1 Simple inverted index search . . . 8
1.3 Skip lists . . . 9
1.4 Postings with perfect skip-lists . . . 11
1.4.1 Cache-efficient storage of posting data . . . 11
1.5 Balanced postings . . . 11
1.5.1 Cache efficiency of balanced posting traversal . . . 12
1.5.2 Storage characteristics of balanced postings . . . 14
1.6 Search with propagating buffers . . . 15
1.7 Efficient merge of postings . . . 19
1.8 Compression of ordered posting lists . . . 20
1.8.1 Block compression . . . 20
1.8.2 Skip list compression . . . 21

2 Implementation **23**
2.1 Data storage . . . 23
2.2 Implementation structure . . . 24
2.2.1 Indexer . . . 25
2.2.2 Searcher . . . 26
2.2.3 Optimizer . . . 27
2.2.4 Collector . . . 27
2.3 Low-level storage interface . . . 27

3 Results and discussion **29**
3.1 Benchmark setup . . . 29
3.2 Search performance . . . 30
3.3 Effect of compression . . . 31

Conclusion **33**

A Installation and usage **37**

A.1 Installation 37

A.2 Using the example programs 37

A.3 Running the benchmarks 38

Introduction

Inverted indexes are a powerful data structure commonly used for fast full-text search in document retrieval systems. The most important ability of inverted indexes is to quickly retrieve documents that contains a certain feature (called a term, usually a keyword), or their combination. Their performance is based on an efficient way of storing the contained data in a form suitable for quickly producing the required output.

More formally, the contained dataset is modeled as a system of subsets of the set of terms, where each subset represents one document. An inverted index is a data structure that stores this model as document sets for each term, where a document is a member of this set when it constains the corresponding term. Usual methods of storage (e.g. as sorted posting-lists) aid efficient computation of set intersections and unions of the term sets, which results in fast processing of Boolean queries usual in search engines.

Currently available inverted-index-providing software is typically oriented towards textual model of information. While this is the most common application of the indexes, the text-oriented interface is redundant in more specific usecases, where the additional text processing may unnecessarily waste resources. Common examples of such use-cases include e.g. spatial indexing [Fon+06], biopolymer sequence indexing [BTNK10], and structural search in molecules [KVG18].

This thesis aims to implement the obvious improvement: An inverted-index database with the text-adaptation layer removed. This allows to vastly simplify the database internals and focus on different aspects of the database, mainly the consumed space, cache efficiency and search performance. The thesis thus mainly concerns efficient implementation strategies of inverted indexes. Additionally, these are examined from the view of a suitable cache-based model, and the related data structures are adjusted to behave efficiently in this model. The implementation performance is analyzed on a cheminformatics-originated dataset, and compared with a similar existing database.

Related work

Many available databases use the inverted index structure: especially the search engines, such as Solr, Xapian, Lucene and Lucy [Smi+15; Bia+12a; Luc], but also the more traditional databases, including PostgreSQL with GiST and GIN[1] indexes. Inverted indexes are also widely used for Google-style search in World Wide Web contents [BP98].

Ongoing research aims at spatial indexing [Fon+06], the effort to compress existing inverted index implementations [AM05; YDS09; Che+10] and to increase the inverted index performance [Gan+16].

Thesis layout This thesis is organized as follows: Chapter 1 examines the inverted index fundamentals, their cache behavior and their possibke improvements for search performance. More precisely, we explain the ideal cache model in which we examine a simple implementation of inverted indexes. We continue describing the skip list data structure used for faster conjunctive query evaluation, a more efficient way of the inverted index implementation in the cache ideal model. Finally, we introduce compression techniques capable of a massive reduction in the inverted index size.

The implementation of the inverted index database is described in chapter 2, which includes the data storage, implementation structure and the low-level storage interface.

In chapter 3 we discuss the resulting library search performance and the measured effect of the inverted index compression.

The thesis is concluded by an overview of accomplished goals and possible applications of the new results.

[1] Generalized inverted Search Tree, Generalized Inverted Index

Chapter 1

Cache-efficient inverted indexes

Inverted indexes are a popular data structure widely used in document retrieval systems, especially in search engines. The usual purpose of an inverted index is to provide fast full-text search, used (among other) for Google-style search in World Wide Web contents. [BP98] The ability to quickly process conjunctive queries is also highly useful in research, e.g. for biopolymer sequence search (such as DNA and protein sequences [BTNK10]) and for structural search in molecules [KVG18].

The search performance of inverted indexes is derived from an efficient way of storing the contained data. Database contents are organized in simple, quickly reversible sequences of content identifiers (usually references to text documents) for each described content property (a text keyword), which allows e.g. fast retrieval of documents that contain a particular queried keyword, or a combination thereof.

In this chapter we define and describe the inverted indexes. For the reasons of performance analysis, we will review the ideal-cache model [Pro99], and examine the basic implementation of inverted indexes from the view of cache efficiency. This is motivated by the requirements on the usual inverted-index-based databases and used further in the thesis.

In the prepared framework, we describe skip-lists as the usual data structure that aids efficiency of inverted index traversal; together with their deterministic version suitable for high-performance and cache-efficiency-oriented environments. We continue by introducing balanced postings, which form a basis for database design and implementation discussed later.

1.1 Ideal cache model

In this section we describe the ideal-cache model popularized by Prokop [Pro99], which we will later use to evaluate the inverted index implementation.

(Z, L) *ideal-cache model* is a model where a memory is represented by two basic parts; an ideal data cache of Z words and an arbitrarily large main memory. Data are transferred between the main memory and the cache in blocks, called *cache lines*. Each cache line consists of L consecutive words.

In current computers, sizes of cache lines and words differ among different levels of cache. For example, CPU caches and RAM operate on several-byte words in 64-byte cache lines. Cache lines grow bigger when moving away from CPU; to several kilobytes (disk cache), megabytes (networked storage) or, in extreme, terabytes in cold storage.

Since the word size is basically fixed by CPU design, for the purposes of this thesis we will pragmatically assume that the word size is constant, and does not affect asymptotic analysis. Prokop also assumes that the cache is *tall*, i.e. the cache lines are relatively short when compared to large cache size:

$$Z = \Omega(L^2).$$

Processor can access only the data that are currently stored in the cache. If the referenced data are found in the cache, the data are transferred to the processor. This situation is known as a *cache hit*. The alternative situation, when the cache does not contain referenced data, is known as a *cache miss* and it requires to retrieve the data from the main memory. When the cache is full and the cache miss occurs, some heuristic is used to choose a suitable cache line which is replaced.

In the ideal-cache model, cache complexity of an algorithm with an input of size n is measured as the number of cache misses $Q(n; Z, L)$, where Z and L are the cache size and the cache line length, respectively. If the algorithm depends on some parameters (set at compile-time or run-time) that can influence the cache complexity for particular cache size and line length, we say the algorithm is *cache aware*. If the cache complexity is independent of parameters, the algorithm is called *cache oblivious*.

Thanks to the possibility of optimizing the parameters for a particular cache properties, the cache-aware algorithms usually provide best achievable cache-efficient behavior. Despite of that, there are many cache oblivious algorithms that are proved to be asymptotically as efficient as their optimized cache-aware counterparts. Moreover, the multitude of diverse (and often unpredictable) cache layers found in current computer environments favors use of cache oblivious algorithms that behave ideally without assuming any specific cache properties.

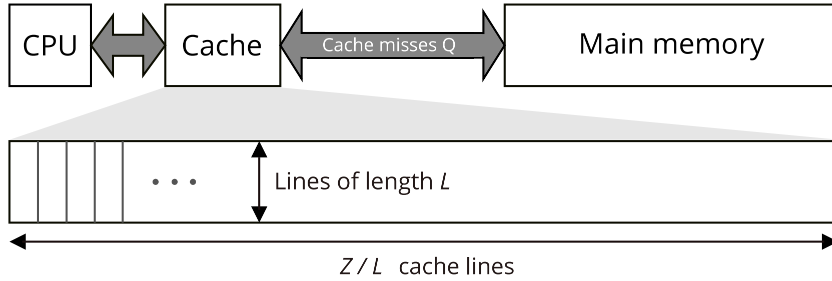


Figure 1.1: The ideal-cache model.

Figure 1.1 shows how the ideal-cache model is designed, which simplifies some aspects of a real cache (e.g. by reducing several cache layers to just one).

The analysis of inverted index implementation conducted further in this thesis is done with respect to this ideal-cache model.

1.2 Inverted indexes

For this section, we will consider a finite set of documents D, where each document is a sequence of occurrences of terms from a finite set T. For purposes of text search, we simplify each document to 'bag of terms', and mathematically represent the whole collection as a map $I : D \to 2^T$.

A database that stores data required to model evaluation of function I is usually called a *forward index*. Forward indexes allow fast search for all terms contained in a given document. A search for documents that contain a given term can be performed by testing the term occurrence in each document, which (without any supporting data structure) requires traversing all term occurrences in all document in $\mathcal{O}(|T| \cdot |D|)$.

If the collection function I is reversed as a mapping $J : T \to 2^D$ and stored in a database, we obtain an *inverted index*. This method of storage allows efficient evaluation of a common kind of query: Finding all n documents that contain a given term can be conducted in optimal time $\mathcal{O}(n)$.

Inverted indexes are the typical, widely-used data structures that power high-performance text search. Boldi and Vigna [BV05] describe the following storage of the function J:

> **Definition 1** (Inverted index and its contents). *Inverted index is formally built from following components:*

- *A* term *is an uniquely identifiable data item. In practice, terms are usually words (i.e. short text strings); in this thesis we consider mainly non-textual terms identified by unique integers.*

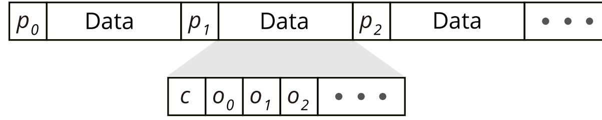


Figure 1.2: A posting where each item contains a document pointer p, count c and positions $\langle o_0, o_1, o_2, \ldots \rangle$.

- *A* document *is a finite sequence of terms. Terms are said to have* occurrences *in documents if they are contained in the corresponding sequences.*
- *A* document pointer *is a unique identifier of a document. In practice, we identify the documents by positive integers.*
- *Consider a document d that contains a term t. An* item *for the document d is a pair consisted of the corresponding document pointer, and a (possibly empty) collection of arbitrary additional data, such as the number of occurrences of term t in document d, or a list of all positions where term t appears in document d.*
- *A* posting *for term t is a sequence of items for each document where t appears; ordered by increasing value of the document pointer.*
- *An* inverted index *for collection D is a data structure made of one posting for each term t.*

A graphical example of a posting is shown in fig. 1.2.

Because the purpose of this thesis does not require any complicated processing of the additional item data, we will, for brevity, discuss postings that contain only plain document pointers. Nevertheless, all further data structures can be easily adjusted to work with the additional data.

1.2.1 Simple inverted index search

Since the purpose of postings is to retrieve a sequence of documents that contains a queried term, postings are accessed sequentially; a simple way of the posting implementation is to store its document pointers in a singly linked list, i.e. each document pointer is extended by an additional reference that points at the next document pointer.

A search for documents that contain a particular term is performed by iterating through the corresponding posting.

Theorem 1. *Considering posting P that contains n document pointers the cache complexity $Q(n; Z, L)$ of P traversal is $\mathcal{O}(n)$.*

Proof. Because posting P is accessed sequentially and its document pointers are not stored in one consecutive block, accessing n document pointers results in $\mathcal{O}(n)$ cache misses. □

However, conjunctive queries (formally defined later in section 1.6) such as a several words search in Internet search engine increase the requirements on the posting data structure. Instead of retrieving all documents from the posting, a verification of the document presence (in a posting) is required. Then, it would be highly inefficient to access every single posting item in a sequential manner. It is useful to add some extra information which allows to process the queries without reading the whole posting list contents. Skip-list-based data structures are commonly used for this purpose.

1.3 Skip lists

Boldi and Vigna [BV05] describe skip lists as a data structure where elements are represented as in an ordered linked list; each element holds several additional references that are used to skip forward in the list as needed.

The usual definitions are as follows:

Definition 2 (Linked lists)**.** *A* singly linked list *is a linear collection of data elements $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ where every element x_i that appears in the list contains a reference to the next element x_{i+1}. The last element is linked to a terminator used to signify the end of the list.*

An ordered singly linked list *of data elements $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ is a singly linked list where elements in the list are increasingly ordered w.r.t. some fixed order relation, that is, $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$.*

Definition 3 (Skip list, skip tower)**.** *A* skip list *of data elements $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ is an ordered singly linked list where each element x_i contains a certain number $h_i \geq 0$ of extra references that are called the* skip tower *of the element x_i. A t-reference contained in this tower references the first element h_j with $j > i$ such that $h_j \geq t$.*

Searching a skip list for an element x is performed by traversing forward references that advance to the highest elements that are still lower than the element we search for. Algorithmically, that is performed by scanning the first tower from the top, stopping before the first reference pointing at an element smaller than or equal to x; then we skip to the element that the reference is pointing at and repeat the process until x is found or no more elements are available.

Skip lists are usually implemented as a probabilistic data structure which is built in layers. The bottom layer is an ordered singly-linked list; each additional

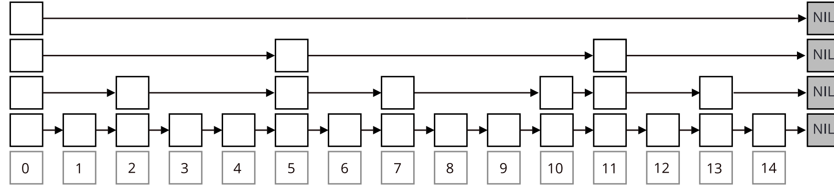


Figure 1.3: A skip list constructed from a singly-linked list in a bottom layer, and three extra layers of references.

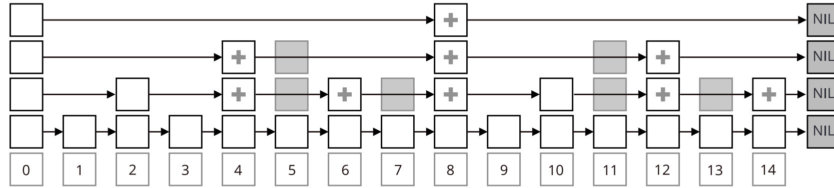


Figure 1.4: A perfect skip list. Gray references from the original skip-list (fig. 1.3) were replaced by the references marked by a plus sign.

skip-list reference acts as a shortcut for the list below and belongs to a separate layer. The layer where the reference belongs is chosen by tossing a p-biased coin ($0 < p < 1$ is fixed) until the outcome is positive. This construction ensures maintaining logarithmic access time on average.[BV05]

Although the usual probabilistic presentation of the data structure behaves well in practice, we do not require the dynamic properties granted by the probabilistic behavior. Instead, we use perfect skip list, a deterministic version of skip lists described in Boldi and Vigna [BV05]. Static nature of the perfect skip-list is more suitable for the postings implementation that behave well in the ideal-cache model.

Definition 4 (Perfect skip list)**.** *Let LSB*(x) *and MSB*(x) *be defined as the position of least significant bit in the integer* x *and the position of the most significant bit in the integer x, respectively, where* $x \in \mathbb{N}_0$*. We say that a skip list of* n *elements is* perfect *if the following condition holds on the height of the skip tower* h_i *at the position of item* x_i*:*

$$h_i = \begin{cases} \mathit{LSB}(i) & i > 0 \\ \mathit{MSB}(n) & i = 0 \end{cases}$$

A graphical example of the perfect skip-list is shown in fig. 1.4. We can see that each reference of a particular layer skip just one reference from the immediate lower layer.

1.4 Postings with perfect skip-lists

Intuitively, perfect skip lists can be used to implement the inverted-index postings by using the items of the posting (i.e. term occurrences) to form the contents of the bottom skip-list layer (i.e. the singly linked list).

This straightforward implementation, however, is not designed to be cache efficient (e.g. long postings are not guaranteed to fit the cache). To make better use of the cache, a suitable improvement is necessary.

1.4.1 Cache-efficient storage of posting data

To improve the utilization of the available cache, we group posting document pointers into blocks (called also posting blocks), and transform the bottom layer of the perfect skip list so that it contains only minimal information about the block content, namely:

- the value of the lowest element (document pointer) in the block, and
- a reference to the place where the block is stored.

Each block now contains a small interval of the original linked list. Since this list was ordered, the elements of consecutive blocks are ordered as well.

The resulting structure already reduces cache misses e.g. when traversing the index: Retrieving list elements from a block generates at most 1 cache miss if the block is cache-aligned; which is a great improvement when compared to the original worst-case 1 cache miss per each list element.

Similarly, the whole skip list can be improved: Either we decompose skip-list layers into blocks or we somehow force skip lists to be cache-efficiently stored in the memory. Because the aim of this thesis is to benefit from the cache-oblivious behavior, we choose the latter, in order to avoid using parameters that influence the cache complexity (as stated in section 1.1). The resulting parameterless structure is detailed in the next section.

1.5 Balanced postings

Definition 5 (Balanced postings). *Consider a posting that contains n document pointers divided into m blocks $\langle b_0, b_1, \ldots, b_{m-1} \rangle$ where $m \leq n$ and the size of each block is $|b_i| = \lceil n/m \rceil$. We denote* Low$(b_i)$ *to be the lowest value of b_i. Then, we create a perfect skip list S with the bottom layer containing pairs $($*Low*(b_i), *Ref*$(b_i))$ for each i, where* Ref(x) *represents a reference to x.*

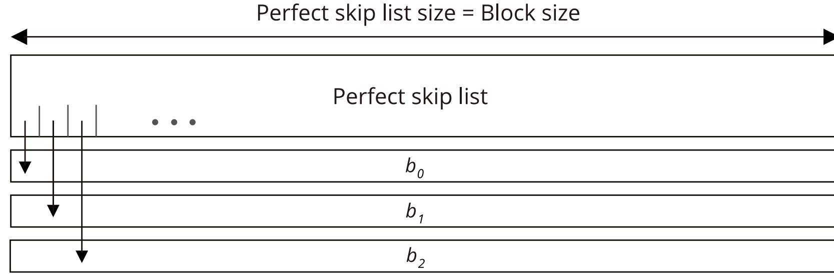


Figure 1.5: Illustration of the size equality between perfect skip list and blocks in a balanced posting.

The resulting posting is balanced *if*

$$\forall i : \lceil n/m \rceil = |b_i| = |S|,$$

where the size of the perfect skip list $|S|$ is equal to the number of all elements (references, document pointers) included in all layers.

Balanced postings are designed to make equal use of the cache for both the perfect skip list and posting blocks, as illustrated in fig. 1.5. Note that the definition does not require specific location of the referenced blocks and also does not specify the organization of each skip list layer. For purposes of simpler explanation of skip-list elements organization, we look at them as *binary trees*.

Binary trees can be represented in several different layouts [BFJ02] dependently on the order of nodes traversal. A *depth-first search* tree traversal is a process of visiting each child before going to the next sibling; we say that nodes are visited in *DFS order*. A *DFS layout* is a representation of a tree where nodes are stored in DFS order.

We represent perfect skip lists as binary trees in DFS layout: Each item of the bottom layer is a *leaf* and all items of other layers are *inner nodes*. A graphical transformation from a perfect skip list into a binary tree is shown in fig. 1.7.

A search for a document pointer in a balanced posting is performed by traversing a binary tree (i.e. a perfect skip list) until a leaf is reached. The leaf contains a reference to a block which is iterated till the document pointer is found or no more document pointers are available.

1.5.1 Cache efficiency of balanced posting traversal

Theorem 2. *The work required for finding a document pointer in a balanced posting with n document pointers divided into m blocks is*

$$\mathcal{O}(\log_2 m + \frac{n}{m}).$$

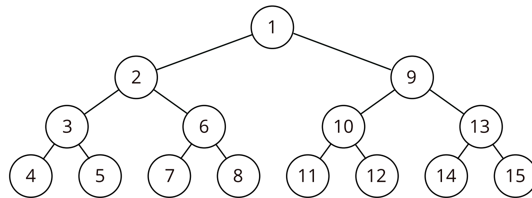
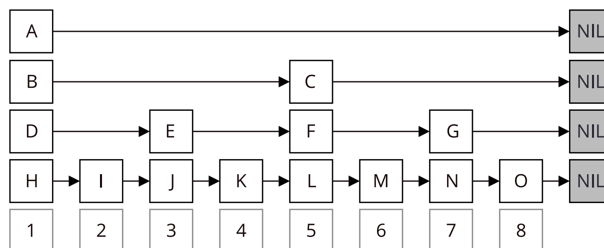


Figure 1.6: The example of the DFS layout of a complete tree with height 4 where each node is represented by a node record. All node records are stored in one array and the numbers in the graphical representation of a node designate their positions in this array.



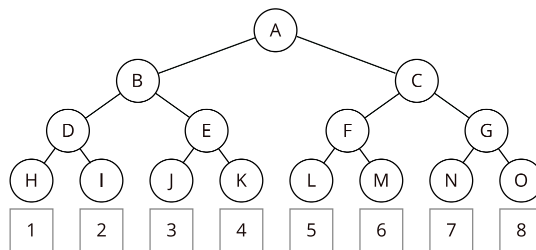


Figure 1.7: A perfect skip list transformation into a binary tree.

In ideal cache model with cache of size Z with lines of size L, the number of cache misses caused by the search for k document pointers at once is

$$\mathcal{O}(\frac{n + \frac{n}{m}}{L} + \frac{k}{L}) \qquad \text{if } Z \geq 2\frac{n}{m}$$
$$\mathcal{O}(\frac{k(n + \frac{n}{m})}{L} + \frac{k}{L}) \qquad \text{if } Z < 2\frac{n}{m}$$

Proof. The work complexity is straightforward; to find a pointer in the posting, we first search for the corresponding block that contains the pointer, by traversing through the tree-structured perfect skip list made of m leaves in $\mathcal{O}(\log_2 m)$ time Then, the block is searched for pointer p, causing $\mathcal{O}(\frac{n}{m})$ cache misses.

To analyze the cache misses, the cache complexity of balanced posting searching is divided into two basic cases: $Z \geq 2\frac{n}{m}$, that is, the perfect skip list fits the cache together with one posting block; or $Z < 2\frac{n}{m}$ if they do not.

In the first case, the cache misses occur only when the posting is loaded into the cache, which results in $\mathcal{O}((n + \frac{n}{m})/L)$ cache misses. All k queries are evaluated at once so all blocks of total size n are accessed at most once; additionally the perfect skip list of size $\frac{n}{m}$ is accessed.

In the second case, the skip list must be loaded for each searched document pointer, which may result in reloading the cache k times.

Furthermore, both cases require to load k searched document pointers, which takes $\mathcal{O}(k/L)$ cache misses. □

As a consequence, if at least 1 block and a the perfect skip list from the posting fit the cache, the cache complexity does *not* depend on the number of searched document pointers. Since the size of the block in a balanced posting of size n is effectively $\mathcal{O}(\sqrt{n})$ (see theorem 3); postings of practical sizes usually satisfy the conditions for the first case, and behave efficiently. This is the main motivation for grouping the searches for document pointers together, which is examined later in section 1.6.

1.5.2 Storage characteristics of balanced postings

The balanced posting definition provides a useful relation between the document count and block count.

Theorem 3. *A balanced posting with n document pointers divided into m blocks is tall:*

$$n = \mathcal{O}(m^2).$$

Before proving the theorem, we show that the skip list size linearly increases while adding new elements.

Lemma 4. *For a perfect skip list S with m leaves holds:*

$$|S| = \mathcal{O}(m),$$

where the size of the perfect skip list $|S|$ is equal to the number of all elements (references, document pointers) included in all layers.

Proof. We construct a skip list iteratively. Assume a perfect skip list S made of m leaves; let denote its size as C_m. Adding a new leaf l to the skip list S we distinguish two situations:

- m is odd; since the skip list is a binary tree, there is one space left in the lowest layer and we simply add l without creating any of inner nodes of the tree which costs 2 (a reference and a value), or

- m is even; there is no space left in the lowest layer and we need to add two inner nodes plus l and hence the cost is 6.

The size of S can be easily expressed by a recurrence $C_m = C_{m-1} + 2$ when m is odd and $C_m = C_{m-1} + 6$ otherwise; $C_1 = 4$; The solution of recurrences is $C_m = 8m + k$, where $k > 0$ is a fixed constant specific for odd and even m. Thus, $|S| = \mathcal{O}(m)$. □

With this, we can prove theorem 3.

Proof of theorem 3. From the definition of the balanced posting (definition 5) we have:

$$\forall i \in \langle 0, m-1 \rangle : |b_i| = |S|, |b_i| = \lceil n/m \rceil$$

from which we according to lemma 4 obtain

$$n = \mathcal{O}(m \cdot |b_i|) = \mathcal{O}(m \cdot |S|) = \mathcal{O}(m \cdot m) = \mathcal{O}(m^2)$$

. □

1.6 Search with propagating buffers

Common requirement on search engines is an ability to retrieve documents that satisfy a particular combination of terms, i.e. all queried terms must be contained in each document in the resultset. Such queries are usually called *conjunctive*; systematically described as follows:

Definition 6 (Query and results structure).

- *A* literal *is a small logical formula that describes a requirement on term occurrence in a document. For term t, we write t for a positive literal (requirement that the term is present in the document) and $\neg t$ for a negative literal (requirement that the term is* not *present).*[1]
- *A* query *is a non-empty conjunction of literals. A query describes a complex requirement on document contents that is satisfied if and only if all contained literals are satisfied. For practical purposes, we will additionally require all queries have at least one positive literal.*
- *A* resultset *is a set of all documents in an inverted index that satisfy a given query. Elements of resultsets are called* hits.

Each term of a non-empty query denotes a posting. The evaluation of a query simply means to take an arbitrary posting (that corresponds to one queried term), denote its document pointers as the *intermediate resultset* of the query, traverse through all the remaining postings (of the queried terms) and remove from the resultset each document pointer that is not contained in the remaining postings. At the end, the resultset contains only document pointers that match the query.

To adjust this simple algorithm to be cache efficient, we describe an algorithm that works with buffers, in a way similar to buffer handling in the funnel sort algorithm [Pro99].

We use the buffers to store fixed-size parts of the intermediate resultsets. As a result, cache misses occur only when the buffers are loaded into the cache. Next, instead of traversing all postings at once, we only work with limited amount of buffers, which directly limits the cache misses.

For demonstration of the process, consider a query Q of n terms $t_0, t_1, \ldots, t_{n-1}$ and a list of n postings $P_0, P_1, \ldots, P_{n-1}$; each posting P_i corresponds to term t_i. We denote r and r_{min} to be the buffer maximal and minimal size, respectively.

At the beginning, the initial buffer F_0 is filled by the first r document pointers from P_0. We traverse through the next posting P_1 and remove document pointers from F_0 that are not present in P_1 — this step is called a *propagation* of F_0 to P_1.

We continue by propagating F_0 to each posting $P_{i>1}$ until the size of the buffer F_0 decreases below r_{min}. At this point, further propagation of F_0 would cause disproportionate amount of cache misses to the little work done. Therefore, we fill a new buffer F_1 with up to next r document pointers from P_0 and continue by propagating F_1. During the propagation of the subsequent buffers $F_{>0}$, several (not necessarily disjunctive) scenarios may occur:

[1] For brevity, we will sometimes refer to literals as terms. This is supported by almost identical handling of both kinds of literals in most of the implementation.

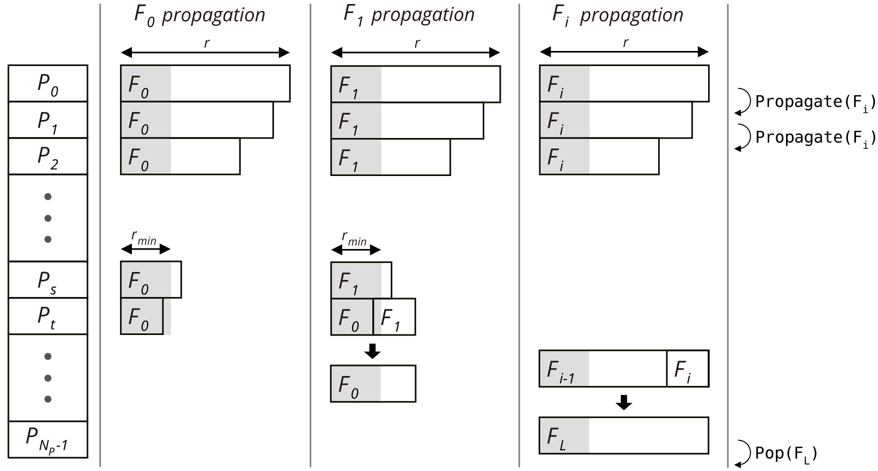


Figure 1.8: Buffer propagation process. Rows represent buffer propagation steps; states of individual buffers are grouped in columns. In the first column, F_0 propagates (moves down) until it is smaller than r_{min}. After that, F_1 is created and propagated (second column). Upon reaching F_0, the propagation of F_1 stops, and the content of F_1 is appended to F_0. After some effort, buffer F_i (last column) propagates to the last posting list, at which point its contents are moved to the search results.

- $|F_i| \leq r_{min}$, at which point we stop the propagation of F_i and continue with F_{i+1}, if available.

- F_i reaches F_j, i.e. F_i approaches posting P_j where the propagation of F_j stopped before. In that case, the content of F_i is added to F_j, and F_i is removed.

- F_i cannot propagate because it has reached the last posting; in this case F_i is a partial resultset of the query, and therefore reported as such and removed from propagation.

When all buffers are smaller than r_{min}, a new buffer is created by filling next r document pointers (if available) from P_0 and the largest buffer is propagated. This process continues until there are no more buffers available for the propagation.

We can see that one buffer propagation is directly implementable by the several-item-search subroutine from theorem 2 by setting $k = r$. Cache is thus efficiently used even for long postings.

The whole algorithm is summarized as algorithm 1. Illustration of the buffer propagation can be seen in fig. 1.8.

Algorithm 1 Buffered search algorithm

Require: $|P| > 0$, a set of postings

1: **procedure** BufferedSearch(P)
2: $\quad B \leftarrow \emptyset$ ▷ Initialize the set of buffers B
3: $\quad$ **repeat**
4: $\quad\quad F_L \leftarrow$ Pop(P, B)
5: $\quad\quad$ add F_L to search results
6: $\quad$ **until** $F_L = \emptyset$
7: **end procedure**

8: **procedure** Pop(P, B)
9: $\quad$ *last* $\leftarrow N_P - 1$
10: $\quad$ **while** $(|B| \neq 0) \vee \neg$End$(P[0])$ **do** ▷ An unfinished posting exists
11: $\quad\quad F_i \leftarrow$ ChooseBuffer(P, B)
12: $\quad\quad$ **if** $F_i = B[last]$ **then**
13: $\quad\quad\quad$ **return** $B[last]$
14: $\quad\quad$ **end if**
15: $\quad\quad$ Propagate(B, F_i)
16: $\quad$ **end while**
17: $\quad$ **return** $\emptyset$
18: **end procedure**

19: **procedure** ChooseBuffer(P, B)
20: $\quad F_i \leftarrow \max(B)$ ▷ Get the longest buffer F_i
21: $\quad$ **if** $(|F_i| < r_{min}) \wedge \neg$End$(P[0])$ **then**
22: $\quad\quad F_0 \leftarrow$ AddNewBuffer$(P[0])$ ▷ Create a new buffer F_0 from P_0
23: $\quad\quad$ **return** F_0
24: $\quad$ **end if**
25: $\quad$ **return** F_i
26: **end procedure**

27: **procedure** Propagate(B, F_i)
28: $\quad j \leftarrow$ NextPosting(F_i) ▷ Index of the following unprocessed posting
29: $\quad$ **while** $|F_i| > r_{min}$ **do**
30: $\quad\quad$ **for all** $p \in F_i$ **do**
31: $\quad\quad\quad$ **if** $p \notin P[j]$ **then**
32: $\quad\quad\quad\quad$ Remove$(F_i[p])$ ▷ Remove p from F_i
33: $\quad\quad\quad$ **end if**
34: $\quad\quad$ **end for**
35: $\quad\quad j \leftarrow j + 1$
36: $\quad\quad$ **if** Exists$(B[j])$ **then** ▷ If F_i reaches F_{i-1}
37: $\quad\quad\quad$ Join$(F_i, B[j])$ ▷ Items of F_i are added at the end of F_{i-1}
38: $\quad\quad$ **end if**
39: $\quad$ **end while**
40: **end procedure**

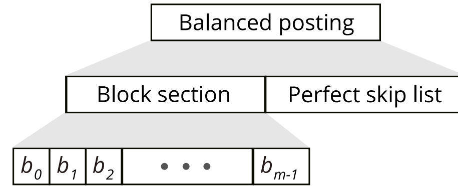


Figure 1.9: The memory layout of ordered postings.

1.7 Efficient merge of postings

Postings are designed to be static objects. Therefore, updates are performed by adding new postings for each newly added set of documents. To improve the search performance, multiple postings are usually combined into a single one, by an operation called *merge*. Because postings can be expected to be arbitrarily long (such that the size of usual available RAM memory is not sufficient), the common requirements on the merge operation include constant space complexity and linear time complexity.

For that purpose, we define ordered postings that designate the placement of blocks in a way efficient for a merge.

Definition 7 (Ordered posting)**.** *A balanced posting is* ordered *when all blocks are stored consecutively in the memory (denoted as a* block section *of the posting), directly followed by the perfect skip list.*

To merge two ordered postings, we first compute the size of the result posting (the sum of all document pointers from both postings) and its block count according to theorem 3. Second, we iterate through both block sections parallelly and create the result block section by merging them. During the iteration, we temporarily store lowest values of result blocks, which are used for creating a perfect skip list from them.

Despite the fact that a merge of block sections requires $\mathcal{O}(1)$ space, the number of lowest values depends on the block count which is definitely not a constant. Thus, we shall suppose that the block count can be bounded by a reasonably big constant which is usually true in practice.

Theorem 5. *The merge operation of two ordered postings of sizes n_1 and n_2 uses $\mathcal{O}(n_1 + n_2 + \sqrt{n_1 + n_2})$ work and requires $\mathcal{O}(1)$ memory.*

Proof. Merging block sections is performed in $\mathcal{O}(n_1 + n_2)$ work and requires $\mathcal{O}(1)$ memory space. The size of the result posting is $\mathcal{O}(n_1 + n_2)$, which corresponds to $\mathcal{O}(\sqrt{n_1 + n_2})$ perfect skip list size according to theorem 3, and lemma 4.

Since a perfect skip list can be represented as a binary tree traversed in the DFS order, the result perfect skip list made of $\mathcal{O}(\sqrt{n_1 + n_2})$ leaves is created in $\mathcal{O}(\sqrt{n_1 + n_2})$ time. Assuming that the result block count can be bounded by a constant, memory requirements are $\mathcal{O}(1)$. $\square$

1.8 Compression of ordered posting lists

A balanced posting list (as in definition 5) is divided into blocks, where each block contains an ordered sequence of document pointers. The density of the document pointers gets (asymptotically) higher with increasing document count, as two adjacent document pointer numbers get closer.

To exploit this behavior, we use delta-encoding [BB02]: Instead of storing the full document pointer integers, we compute and store only differences between them.

1.8.1 Block compression

Definition 8 (Deltas). *Consider two values $a, b \in \mathbb{N}_0$, $a < b$. A* delta *between them is computed as*

$$\Delta(a, b) = b - a.$$

Because computing deltas decreases the absolute values that occur in balanced postings, we can compress those numbers using a suitable encoding.

Definition 9 (VW encoding). *Let x_2 be a binary representation of a number $x \in \mathbb{N}_0$, $0 \leq x < 2^{56}$.* VW encoding *(variable-width encoding) encodes x as*

$$x_2 = \begin{cases} \texttt{0bbbbbbb} & 0 \leq x < 2^7 \\ \texttt{10bbbbbb bbbbbbbb} & 2^8 \leq x < 2^{14} \\ \texttt{110bbbbb bbbbbbbb bbbbbbbb} & 2^{14} \leq x < 2^{21} \\ \vdots & \vdots \\ \texttt{11111110 bbbbbbbb} \ \ldots & 2^{49} \leq x < 2^{56} \end{cases}$$

where $\texttt{b}$ is binary digit of x_2. Additionally, we define $\|x\|$ to be size of the encoded x_2 in bytes.

The VW encoding is similar to the widely used UTF-8 character encoding [All+12]. As the main difference, UTF-8 additionally uses control marks at the beginning of each byte that aid decoding error recovery; these are not required for our purposes.

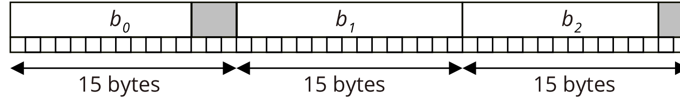


Figure 1.10: First three blocks of a plain posting, where each block occupies 15 bytes. Gray rectangles at the end of some blocks represent a termination mark.

1.8.2 Skip list compression

The bottom layer of a perfect skip list contains pairs, each formed by the lowest value of a particular block and a reference to this block (definition 5).

Despite the number of document pointers in every block is constant, the byte size of each compressed block varies, which would force the skip-list to use precise byte-references. Instead, by fixing the length of blocks we can compute block offsets without the references — because a perfect skip list can be seen as a full binary tree, skipping subtrees is simply performed by integer arithmetics on the reference, in $\mathcal{O}(1)$ time.

Moreover, since such block section followed by the prefect skip list has a suitable layout for the effective merge of ordered postings, the fixed size of blocks influences neither the cache complexity analysis nor the merge layout.

Definition 10 (Plain posting). *A balanced posting that contains m blocks $\langle b_0, b_1, \dots, b_{m-1} \rangle$ is* plain *if and only if this posting is* ordered *and the following condition holds:*

$$\forall i, j, i \neq j : \|b_i\| = \|b_j\|.$$

A free unused space left at the end of each block due to the alignment can be filled with a specific sequence of bits representing a special termination mark, as shown in fig. 1.10. Despite the fact that those marks cost extra memory, in result the complete postings occupy less memory.

Finally, in section 1.2 we have defined that the postings may store additional amount of attributes along the document pointers. Since these are typically represented by integers, we can compress them individually by the VW encoding (this is discussed more precisely in the following chapter).

Chapter 2

Implementation

This chapter describes the structure and inner workings of the resulting software implementation.

Following the naming convention of search engines that motivated this work (Lucene, Lucy), the resulting software library is called *Mary*.

We have chosen the C++ programming language for the implementation. The choice is substantiated mainly by the system-level interfaces of C++ that allow us to use UNIX system facilities directly and manage the memory effectively, which is necessary for achieving optimal cache behavior and performance.

2.1 Data storage

As mentioned in section 1.7, postings (and therefore inverted indexes as well) are static data structures, i.e each newly added set of documents creates new postings. A commonly used way to maintain inverted indexes is to decompose them into multiple subindexes called *segments*. Each segment is a fully independent inverted index and can be searched separately. Operations over the database are the following [Bia+12b]:

- read, i.e. a sequential traversal of each active (see later) segment, the resultsets are joined together,
- update, i.e. an addition of a new segment that contains newly added documents,
- merge, i.e. a process of compaction 2 segments into one, which is applied on each posting (as stated in section 1.7)

In general, we allow only expanding of the inverted index (from this point onward simplified as index). If a deletion of some documents is required, we

simply add all terms occurred in this document to a query as negative literals, which excludes them from the resultset (see definition 6).

Since the index is composed of several segments, index evolving can be performed independently of searching unless it modifies the index. Therefore, if we merge segments that exist, instead of deleting the old ones, we just mark them as inactive and they are removed when none of them is searched.

We distinguish operations over the index between these that modify the database and those that do not. The modifying operations cannot be run at the same time, in order to prevent write-write conflicts and data corruption. This is accomplished by placing an exclusive lock on the database. The operations that do not modify the index acquire a shared lock.

Locking is maintained by `flock` [McK+96], a UNIX syscall that provides an exclusive and shared lock exactly as the subroutines require. Since `flock` applies locks on files (not directories), we create two additional files; `write.lock` — for holding exclusive locks — and `read.lock` — for holding shared locks.

For the purposes of implementation, we first define the index structure and follow with description of the subroutines that operate on the index.

Index An *index* is an overarching abstraction of a database using an inverted index as its storing data structure. The database is completely identified by its root directory, which contains:

- a separate directory for each segment,
- `segments.alive` file, which contains identifications of active segments and their lengths.

Segment A *segment* is a fully independent index that contains two files:

- `postings.data` — the file that contains plain postings for each term, in increasing order of term;
- `lexicon.data` — the file that contains several attributes for each posting P: the beginning offset of P in `postings.data`, block section size, perfect skip list size and the length (the number of document pointers) of P.

2.2 Implementation structure

The index is maintained by subroutines implemented in the searcher, indexer, optimizer and collector classes. Before their thorough examination, we describe following classes that each subroutine uses:

Index (defined in `global/index.hpp`)

The index class holds and manages all necessary data about the active segments and provides locking, unlocking and a method `draft_segment()` which returns a draft of a new segment holding its `id` and `path`.

This draft is used when a segment builder (see below) creates a new segment. The index object is meant to be used as a singleton that provides access to global database properties.

Posting (defined in `global/posting.hpp`)

The posting class represents the plain posting implementation which can create, copy and merge postings. It manages block objects in its block section together with the skip list object, and provides a method `contains(d)` that reports whether the document pointer `d` is contained in the posting list.

Skiplist (defined in `global/skiplist.hpp`)

The skiplist class provides the functionality for creating skip lists as binary trees, implements skip-list operation on memory-mapped blocks and while searching term `t` determines the block which might contain `t`.

Block (defined in `global/block.hpp`)

The block class provides the functionality for creating blocks and implements the `contains(t)` method which returns true if term `t` is included in the block. For storing values uses a coder object.

Coder (defined in `global/coder.hpp`)

The coder namespace is responsible for encoding and decoding block values according to the VW encoding defined in definition 9.

2.2.1 Indexer

Given a set of documents an *indexer* is an object that exclusively locks the index, creates a new segment by flushing postings data of the set into the `postings.data` file and their attributes into the `lexicon.data` file. Then, modifies the `segments.alive` file and unlocks the index.

Indexer (defined in `store/indexer.hpp`)

The indexer environment provides two operations — `add(S, n)` and `commit(B)`, where `S`, `n` and `B` represent schema object, document count

and segment builder object, respectively. The `add(S, n)` operation prepares postings data of `n` documents, as described in the given schema `S`. The `commit(B)` operation creates a new segment using the segment builder `B`. If segment count reaches the maximum, two shortest segments are merged.

Builder (defined in `store/builder.hpp`)

The builder environment provides two operations — `create(SD, P)` and `merge(SL, SR, SD)`, where `P`, `SL`, `SR` and `SD` represent prepared postings data, left segment, right segment and segment draft, respectively. The `create(SD, P)` operation creates a segment from the draft `SD` by flushing the postings data `P`. The `merge(SL, SR, SD)` operation creates a segment from the draft `SD` by merging two given segments `SL` and `SR`.

Schema (defined in `store/schema.hpp`)

The schema provides an environment for running two operations — `next_document()` and `next_term()`, which describe an input (i.e. the representation of documents that are added to the database). The `next_document()` method returns a document pointer of next document and the `next_term()` method returns next term of this document.

2.2.2 Searcher

After giving a query, a *searcher* is an object that places a shared lock on all active segments recorded in the `segments.alive` file, sequentially traverse through them and retrieves the resultset of the query. Then, unlocks the segments.

Searcher (defined in `search/searcher.hpp`)

The searcher environment requires a query and provides two operations — `has_next()` and `search_next()`. The `has_next()` method indicates whether there are available active segments that have not been searched so far. The `search_next()` method traverse next active segment using segment walker object and returns a result represented by hit object.

Walker (defined in `search/walker.hpp`)

The segment walker environment requires a segment path in file system which is set by searcher object, and provides an operation `find(Q)` that search given query `Q` using hitheap object. Returns hit object.

Hitheap (defined in `search/hitheap.hpp`)

The hitheap environment requires detailed information about the queried terms — the posting offset, block section size, skip-list size and the posting

length — which is set by segment walker object. The purpose of the hitheap is to manage the search with propagating buffers as we have described in section 1.6. The only operation that the hitheap provides is `pop()` which returns the last buffer that represents a partial resultset of the given query.

Hit (defined in `search/hit.hpp`)

The hit environment requires a reference to hitheap object and provides an iterator that allows to loop through the hit. After obtaining resulting buffer from the hitheap, the iterator sequentially returns the hit content until the end of the buffer is reached. Then, the iterator calls `hitheap::pop()` per se and continues with returning document pointers of the next buffer. The process is repeated till the hitheap object has no more buffers available.

The whole searching process can be seen in fig. 2.1.

2.2.3 Optimizer

An *optimizer* is an object that exclusively locks the index, takes and merges two shortest segments. Then, it modifies the `segments.alive` file and unlocks the index.

Optimizer (defined in `store/optimizer.hpp`)

The optimizer environment provides an operation `optimize(B)` which takes a segment builder object `B` and merges two shortest segments.

2.2.4 Collector

A *collector* is an object that exclusively locks the index, takes inactive segments that are not locked by a shared lock and removes them. Then, unlocks the index.

Collector (defined in `store/collector.hpp`)

The collector environment provides an operation `collect()` which simply deletes those inactive segments that are not locked by a shared lock.

2.3 Low-level storage interface

During the search, all data loading is handled by `mmap` [McK+96], a UNIX syscall that provides mapping of file contents into memory.

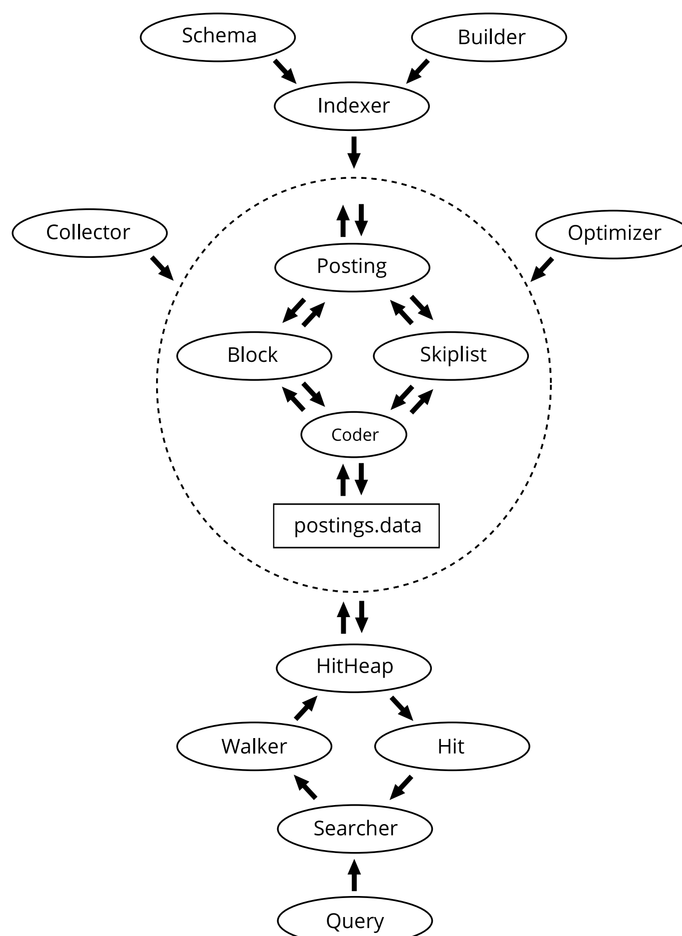


Figure 2.1: Simplified view of the interactions between main implementation classes.

`Mmap` maps pages that start at a given address, which is easily directly applicable to the postings — each posting has defined its beginning offset and size in the `lexicon.data` file.

The mapped memory is used for binary search over all offsets of postings (postings are sorted in increasing order of term).

Mmapper (defined in `global/mmapper.hpp`)

The mmapper environment provides an operation `get(O, S)` where `O` and `S` denote beginning offset of the file to be mapped and the size of mapping content, respectively. Moreover, the mmapper object behaves as a cache — given the maximal allowed size `n` to be mapped it holds the mapped content until `n` is reached. At that point, the oldest unneeded mapped content is unmapped.

Chapter 3

Results and discussion

In this chapter, we analyze search performance and compression capabilities of the implementation. The results are compared with the performance of Apache Lucy database [Luc], a search engine library that provides full-text search, which is a 'loose port' of the Apache Lucene search engine into C language.

3.1 Benchmark setup

The testing datasets were prepared to simulate a realistic workload in cheminformatics. The documents were prepared from compounds in ChEMBL database [Gau+11], which were converted to integer fingerprints using the same method as in Sachem database [KVG18]. For the fingerprinting algorithm, we used parameter GraphSize=5 and multiplicity encoding of up to 2^2 feature repeats [KVG18, section 'Fingerprint structure']. From the result we extracted random 10,000, 100,000 and 500,000 documents to obtain three testing datasets.

To obtain the queries, we prepared 250 queries by applying the same fingerprinting algorithms to random compounds from the ChEMBL database. The queries were then randomly reduced to lower number of query terms, which simulates the effect of query filtering in Sachem. Using that, we obtained five query sets for query sizes 1, 3, 10, 30, and 100, each with 50 queries.

Each query set was tested 10 times and 2 iterations were used to cache data.

All benchmarks were measured on Intel® Core™ i7-4790K CPU clocked at 4.00GHz with 16 GiB RAM running a Linux kernel version 4.15.54, using a 7200rpm rotational disk drive.

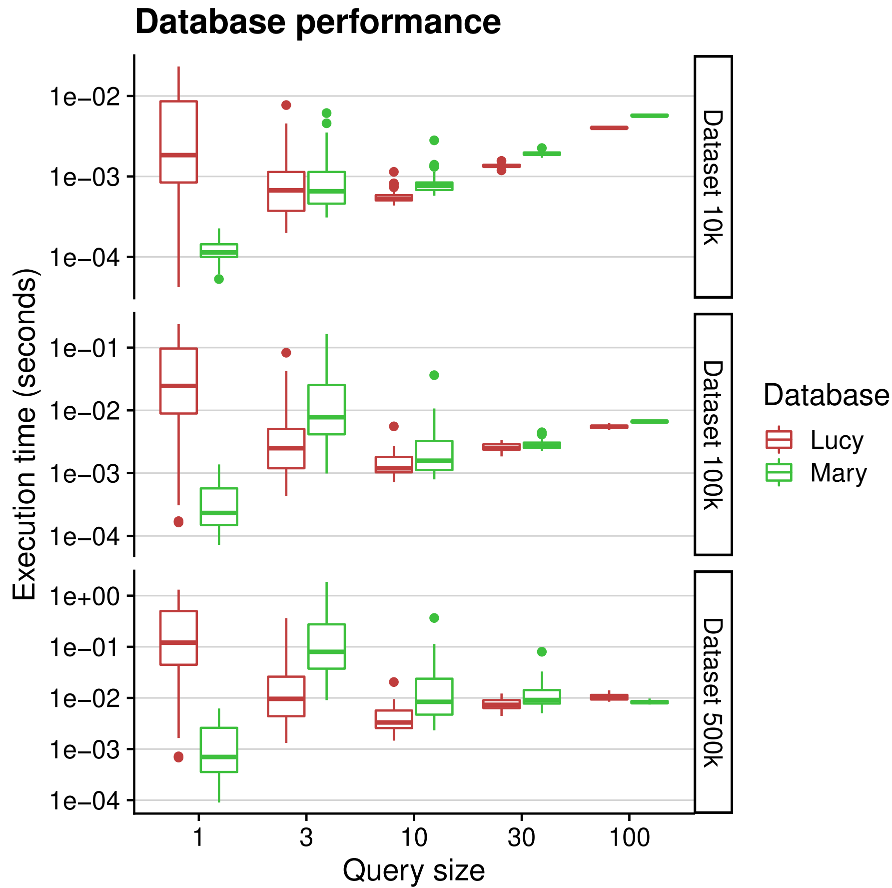


Figure 3.1: Search performance comparison of both libraries.

3.2 Search performance

The search performance of both libraries is shown in fig. 3.1. Overall, the performance can be viewed as similar.

We can see that the cache efficiency of Mary emerges while evaluating the queries of the length 1 (it is a simple access of the posting data) and the longest queries in 500k dataset (the effect of the balanced-posting cache-based model).

Otherwise, the result shows that the performance of Mary is either comparable or slightly slower than of Lucy. Since Mary was optimized only for cache efficiency and raw performance optimizations were not implemented yet, we consider the resulting performance to be a very good result that can be easily improved by focusing on the relevant parts of the implementation.

	Lucy	Mary	saved
10k	40.2 MiB	3.5 MiB	91%
100k	387.4 MiB	30.1 MB	92%
500k	2.0 GiB	152.7 MiB	91%

Table 3.1: Size comparison of the datasets stored in Lucy and Mary; percentages of saved space are rounded to integers.

	postings.data			lexicon.data		
	original	compressed	saved	original	compressed	saved
10k	22.2 MiB	3.1 MiB	86%	1.0 MiB	365.0 KiB	63%
100k	209.8 MiB	29.3 MiB	86%	1.9 MiB	731.6 KiB	61%
500k	798.9 MiB	151.5 MiB	81%	2.7 MiB	1.2 MiB	56%

Table 3.2: Differences between files `postings.data` and `lexicon.data` before and after compression with various stored datasets; percentages of saved space are rounded to integers.

3.3 Effect of compression

The space used for index storage by both libraries is compared in table 3.1. The results show that the saved space of Mary is 91% in average. The majority of occupied space of Lucy is caused by storing terms in the string format and, moreover, Lucy does not compress skip lists. Thus, to show the effectivity of the compression techniques described in section 1.8, we compute the compression ratio of single files (`postings.data` and `lexicon.data`), i.e the relative reduction in size of the files produced by discussed encodings.

In table 3.2 each test shows that saved space of `postings.data` files is greater than 80%, mainly as the result of powerful delta encoding described in section 1.8.1. `Lexicon.data` sizes are reduced by more than 50%, which is not as much as postings because the attributes are only encoded by VW encoding.

Generally, we can see that the storage requirements of Mary are more than 10×better than the requirements of Lucy, which appears notably useful, especially in large databases.

Conclusion

This thesis describes the design and implementation of a cache-efficient inverted-index database that processes general integer data, and is not complicated by the text processing layer usually present in current software.

The inverted index implementation was examined from the view of ideal-cache model, and the design of implemented data structures has been optimized to behave efficiently in this model. We specially focus on the most common performance-critical algoritms used in inverted indexes, including the parallel traversal of multiple postings, and merging of the inverted index segments.

The resulting implementation focuses on optimizing the cache efficiency and consumed storage space. This was verified by a benchmark conducted with a realistic dataset based on molecular structure-indexing data. The benchmark has shown the viability of implemented data compression (providing almost 10×improvement over the currently available databases). Although the raw performance optimization was not the main goal of the thesis, the result displays similar performance as current databases. In test cases that stress cache-efficient behavior, the other databases are outperformed.

Future work

The work on the thesis showed several possible directions of future research and implementation improvements:

- The buffered search algorithm described in section 1.6 currently consumes the majority of computation time in search. Better implementation of the buffer propagation, or perhaps another efficient way to speed up the evaluation of conjunctive queries, could massively improve the search performance.

- The current design of posting lists focuses on cache-oblivious behaviour, but does not take in account the computational efficiency of the traversal, or any of the properties of the underlying hardware. Optimizing the

data structure is the second main future goal that can bring performance improvements.

- The library API currently does not allow to store additional information in the posting lists (such as the bookkeeping required for efficient similarity search implementation). Designing a correct API and compression methods will allow more applications of the library.

- Since this thesis was originally motivated by the cheminformatics requirements of Kratochvíl, Vondrášek, and Galgonek [KVG18], the results will be applied to the ongoing research of the high-performance chemical structure search. For example, the space efficiency of the new software allows an immediate improvement of the existing deployments of Sachem cartridge, where terabytes of storage can be saved.

Bibliography

[All+12] Julie D Allen et al. *The Unicode Standard.* Vol. 8. Citeseer, 2012.

[AM05] Vo Ngoc Anh and Alistair Moffat. "Inverted index compression using word-aligned binary codes". In: *Information Retrieval* 8.1 (2005), pp. 151–166.

[BB02] Dan Blandford and Guy Blelloch. "Index compression through document reordering". In: *Proceedings DCC 2002. Data Compression Conference.* IEEE. 2002, pp. 342–351.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. "Cache oblivious search trees via binary trees of small height". In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics. 2002, pp. 39–48.

[Bia+12a] Andrzej Białecki et al. "Apache lucene 4". In: *SIGIR 2012 workshop on open source information retrieval.* 2012, p. 17.

[Bia+12b] Andrzej Białecki et al. "Apache lucene 4". In: *SIGIR 2012 workshop on open source information retrieval.* 2012, p. 17.

[BP98] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine". In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[BTNK10] Inbal Budowski-Tal, Yuval Nov, and Rachel Kolodny. "FragBag, an accurate representation of protein structure, retrieves structural neighbors from the entire PDB quickly and accurately". In: *Proceedings of the National Academy of Sciences* 107.8 (2010), pp. 3481–3486.

[BV05] Paolo Boldi and Sebastiano Vigna. "Compressed perfect embedded skip lists for quick inverted-index lookups". In: *International Symposium on String Processing and Information Retrieval.* Springer. 2005, pp. 25–28.

[Che+10] David M Chen et al. "Inverted Index Compression for Scalable Image Matching." In: *DCC.* 2010, p. 525.

[Fon+06] Marcus Fontoura et al. "Inverted index support for numeric search". In: *Internet Mathematics* 3.2 (2006), pp. 153–185.

[Gan+16] Abdullah Gani et al. "A survey on indexing techniques for big data: taxonomy and performance evaluation". In: *Knowledge and information systems* 46.2 (2016), pp. 241–284.

[Gau+11] Anna Gaulton et al. "ChEMBL: a large-scale bioactivity database for drug discovery". In: *Nucleic acids research* 40.D1 (2011), pp. D1100–D1107.

[KVG18] Miroslav Kratochvíl, Jiří Vondrášek, and Jakub Galgonek. "Sachem: a chemical cartridge for high-performance substructure search". In: *Journal of cheminformatics* 10.1 (2018), p. 27.

[Luc] *Apache Lucy*. Web page. 2017. URL: `https://lucy.apache.org/`.

[McK+96] Marshall Kirk McKusick et al. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[Pro99] Harald Prokop. "Cache-oblivious algorithms". PhD thesis. Massachusetts Institute of Technology, 1999.

[Smi+15] David Smiley et al. *Apache Solr enterprise search server*. Packt Publishing Ltd, 2015.

[YDS09] Hao Yan, Shuai Ding, and Torsten Suel. "Inverted index compression and query processing with optimized document ordering". In: *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 401–410.

Appendix A

Installation and usage

A.1 Installation

A C++17 capable compiler is needed for the library to compile. To compile all necessary sources that are attached to the thesis, simply run:

```
$ make
```

This also includes the installation of example programs `indexer.demo`, `searcher.demo`, `collector.demo` and `optimizer.demo`. To install only these subroutines, run:

```
$ make demo
```

A.2 Using the example programs

An inverted index database can be created by running:

```
$ ./indexer.demo documents-file
```

`Documents-file` contains a set of documents, each document occupies one row and contains its document pointer followed by terms. When the database exists, the same command can be used for adding new documents.

The database is implicitly stored in the directory `./index` which can be configured in the `defaults.hpp` file.

After this, to perform a search, run:

```
$ ./searcher.demo
```

To merge two shortest segments, run:

```
$ ./optimizer.demo
```

Otherwise, segments are automatically merged after reaching the MAX_SEGMENT_COUNT constant which can be adjusted in the defaults.hpp file as well.

Removing inactive segments can be done by running:

```
$ ./collector.demo
```

A.3 Running the benchmarks

To run all benchmarks, change directory to ./tests/ and run:

```
$ ./mary-analyze.sh # the results will appear in ./results/
```