Algorithms and Data Structures for In-Memory Text Search Engines

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie

genehmigte
Dissertation

von
Dipl.-Inf. Frederik Transier
aus Heidelberg

Karlsruhe, 2009

Tag der mündlichen Prüfung:	20. Januar 2010
Referent:	Prof. Dr. Peter Sanders
Korreferent:	Prof. Dr. Gerhard Weikum

Abstract

In the current information era, everyone has instant access to huge amounts of textual data. Unfortunately, the information is often unstructured and difficult to handle. So the use of search engines is indispensable.

Different aspects of search engines are well-established but still active areas of research. Across the relevant literature, inverted index data structures have been proved to be the key to fast text search. Classical inverted index-based search engines store at least major parts of their data structures on disk. But with growing amounts of RAM in recent computer systems it becomes possible to keep everything in main memory.

This thesis studies algorithms and data structures for in-memory text search in a bottom-up fashion. We investigate the predominant operation on inverted indexes, which requires intersecting two sorted lists of integers. We explore compression and performance of different list data structures in combination with suitable intersection algorithms. We present a new integer list data structure that comes along with an intersection algorithm called *Lookup*. It has interesting theoretical properties, and outperforms previous algorithms in particular for the very frequent case when the input lists have quite different lengths.

Using the new data structure, we develop the algorithmic core of a full text database that allows fast Boolean queries, phrase queries, and document reporting using less space than the input text. The system uses a carefully choreographed combination of classical data compression techniques and inverted index based search data structures. It has an efficient memory management that avoids fragmentation and that keeps down the space requirement close to the amount that we would expect by summing up the sizes of all its components.

As phrase queries are among the computationally most expensive operations of text search engines, we present a new flexible approach that allows a speedup of the most difficult queries, yielding a certain performance robustness. We also use this approach to design a new index scheme

for small text documents tailored to database-like scenarios. It outperforms classical positional indexes for both space requirement and querying time.

In contrast to other search engines that keep copies of documents for return as hits, our document reporting algorithm needs only to store the differences between the information contained in the indexes and the original documents. In most applications, reconstructing documents is faster than retrieving them from disk.

We extensively evaluate all our algorithms and data structures using real-world text collections and query logs. Our experiments show that our main memory text search engine can considerably improve query times of disk-based systems, and we show that inverted indexes are preferable to purely suffix array based techniques for in-memory (English) text search engines.

Zusammenfassung

Im heutigen Informationszeitalter hat jeder direkten Zugriff auf riesige Mengen von Textdokumenten. Sei es im Internet, im Firmennetzwerk oder am eigenen PC — überall sind Informationen leider oft unstrukturiert und teils unübersichtlich abgelegt. Um diese Datenflut überhaupt in den Griff zu bekommen, ist es unerlässlich, Suchmaschinen zu bemühen.

In Praxis und Forschung hat sich der *invertierte Index* als effiziente Datenstruktur für schnelle Textsuche durchgesetzt. Dabei wird zunächst jedem Textdokument einer zu indizierenden Dokumentensammlung ein ganzzahliger Bezeichner zugewiesen. Bei der Konstruktion des invertierten Index wird dann für jedes Wort der Dokumentensammlung eine Liste aller Bezeichner angefertigt, in deren Dokumenten dieses Wort auftritt. Um damit zum Beispiel nach Dokumenten zu suchen, die zwei bestimmte Wörter enthalten, muss einfach nur die Schnittmenge der entsprechenden Listen berechnet werden.

Klassische Suchsysteme speichern aufgrund der großen Menge von Daten den überwiegenden Teil ihrer Indexstrukturen auf Festplatten. Da aber aktuelle Computersysteme zunehmend mehr RAM besitzen, wird es (bald) möglich, alle Daten im Hauptspeicher zu halten. Der sich daraus ergebende Vorteil liegt nahe: Durch die um einen Faktor von 10^{-7} geringere Latenz und die um einen Faktor von 10^2 höhere Bandbreite bei Speicherzugriffen des RAM gegenüber einer Festplatte ist zu erwarten, dass sich die Suchzeiten erheblich verkürzen. So ergibt sich auch ein viel feinkörnigeres Optimierungspotential.

Beginnend mit einer der häufigsten Elementaroperationen von invertierten Indices, dem Berechnen der Schnittmenge zweier sortierter Listen von ganzen Zahlen, untersucht diese Dissertation Algorithmen und Datenstrukturen für Hauptspeichersuchmaschinen. Dabei steht nicht nur die reine Laufzeit im Vordergrund, sondern auch eine möglichst kompakte Repräsentation der zugrunde liegenden Indexstrukturen. Wir untersuchen

bereits existierende Arrayrepräsentationen in Kombination mit verschiedenen Schnittalgorithmen. Basierend auf der Idee, die ganzzahligen Dokumentenbezeichner randomisiert zu vergeben (und damit randomisierte Listen zu erhalten), stellen wir eine neue Arraydatenstruktur vor, die mit einem nicht ausschließlich vergleichsbasierten Schnittalgorithmus einhergeht. Durch Aufteilung der Listenelemente in Bereichsintervalle mit konstanter Größe einer Zweierpotenz und das Abspeichern von Einsprungspunkten in diese Intervalle, ist es während des Schnitts möglich, direkt oder kurz vor die Stelle zu springen, an der ein bestimmter Wert zu erwarten wäre. Dadurch reduziert sich die Anzahl der zu betrachtenden Werte erheblich, und durch die Randomisierung lassen sich interessante theoretische Eigenschaften feststellen. Der Algorithmus ist für den häufigsten Fall unserer Anwendung zugeschnitten, falls die Listen in ihrer Länge sehr unterschiedlich sind. Gegenüber dem klassischen Algorithmus, der die Listen sequentiell schneidet, ist er in den meisten Fällen sogar um Größenordnungen schneller.

Mit Hilfe unserer neuen Datenstruktur entwickeln wir den algorithmischen Kern einer Volltextdatenbank, die Boolsche UND-Suchen, Phrasensuchen und die Rekonstruktion indizierter Texte unterstützt. Die dabei benötigten Datenstrukturen belegen weniger Platz als der Eingabetext. Im Gegensatz zu anderen Suchmaschinen, die zusätzlich zu den Indexstrukturen auch die Originaltexte speichern, um diese als Resultate zurückgeben zu können, hält unser Volltextdatenbankkern nur die Differenzen zwischen diesen Originalen und den Informationen, die ohnehin schon im Index enthalten sind. In den meisten Anwendungen in der Praxis ist die Rekonstruktion der Texte aus unseren Datenstrukturen schneller, als wenn diese von der Festplatte nachgeladen werden müssten. Wir skizzieren auch, wie diese Indexdatenstrukturen schnell und platzeffizient aufgebaut werden können, indem Fragmentierung und häufiges Kopieren vermieden wird.

Die Suche nach kurzen Folgen von Wörtern (Phrasen) ist eine der zeitintensivsten Aufgaben einer Textsuchmaschine. Denn bei solchen Phrasensuchen muss nicht einfach nur bestimmt werden, in welchen Dokumenten die einzelnen Wörter einer Phrase vorkommen, sondern auch an welcher Stelle. Nur so kann eine Phrase eindeutig identifiziert werden. Dazu werden klassischerweise *Positionsindices* verwendet. Diese speichern sämtliche Positionen an denen ein gegebenes Wort innerhalb eines gegebenen Dokuments auftritt. Wir präsentieren einen Algorithmus, der die schwierigsten kurzen (Teil-)Phrasensuchen für eine Sammlung von Textdokumenten ermittelt und beschleunigt. Dabei schätzt er deren Aufwand durch ein fle-

xibles Kostenmaß, und speichert deren Ergebnis im Voraus, falls dieser zu hoch sein sollte. So ist es möglich, eine gewisse Robustheit zu erlangen. Wir nutzen diesen Ansatz auch, um einen auf sehr kurze Texte zugeschnittenen Index zu entwerfen. Dabei speichern wir die Wortpositionen nicht explizit wie in einem Positionsindex, sondern implizit, indem wir für jedes Dokument die Abfolge dessen Wörter in komprimierter Form ablegen. Für eine Phrasensuche führen wir dann zunächst eine UND-Suche nach den Teilwörtern durch, um mögliche Trefferdokumente vorzuselektieren. Schließlich identifizieren wir tatsächliche Treffer mittels sequentieller Suche auf den Wortfolgen der Kandidatdokumente. Um die Vorselektion zu verfeinern und damit diesen Suchvorgang zu verkürzen, speichern wir im invertierten Index zusätzlich kurze (Teil-)Phrasen nach obigem Schema. Als Resultat bietet unser *Kurztextindex* sowohl schnellere Durchschnittssuchzeiten, als auch bessere Suchzeiten unter den ungünstigsten Rahmenbedingungen und benötigt dabei weniger als dreiviertel des Platzes im Vergleich zu unserem Positionsindex.

Wir zeigen ausgewählte Details unserer hoch-modularen und dennoch effizienten Implementierung, die es erlaubt, auf verschiedenen Abstraktionsebenen schnell und einfach einzelne Komponenten auszutauschen. So ist es möglich, alle vorgestellten Algorithmen und Datenstrukturen auf einer gemeinsamen Basis zu evaluieren. Dabei greifen wir auf reale Textdokumente und Anfrageprotokolle zurück. Unsere Ergebnisse zeigen, dass unsere Hauptspeichersuchmaschine bis zu zwei Größenordnungen schneller als eine herkömmliche Festplatten-basierte Suchmaschine ist. Außerdem zeigen wir, dass unsere Hauptspeichersuchmaschine in praxisrelevanten Anwendungen selbst für Phrasensuchen Suffixarray-basierten Hauptspeichersuchsystemen vorzuziehen ist.

Danksagungen

Diese Arbeit wäre ohne die Unterstützung vieler nicht möglich gewesen. Dafür möchte ich im Einzelnen danken:

zuallererst Herrn Prof. Dr. Peter Sanders, der meine Arbeit betreut hat; Roland Kurz und Franz Färber vom TREX-Team der SAP AG, die das Projekt auf den Weg brachten, aus dem diese Dissertation entstand;

ebenso der Prüfungskommission, Prof. Dr. Gerhard Weikum für die freundliche Übernahme des Korreferats, dem Prüfungsvorsitzenden Prof. Dr. Jörg Henkel, den Prüfern Prof. Dr. Dorothea Wagner und Prof. Dr. Jörn Müller-Quade, sowie Prof. Dr. Peter Schmitt, der als Mitglied des Promotionsausschusses an meiner Prüfung teilgenommen hat;

dem gesamten TREX-Team, insbesondere Kai Stammerjohann und Nico Bohnsack für die Unterstützung bei der Produktisierung der entwickelten Algorithmen; Frank Renkes und Helmut Cossmann für ihre Hilfe und Unterstützung vor allem während der Anfangsphase des Projekts; Arne Schwarz für seine Bemühungen als Vermittler bei vertraglichen Angelegenheiten; im Besonderen Andrew Ross, der diese englischsprachige Arbeit Korrektur gelesen hat; außerdem allen Kollegen der Arbeitsgruppe in Karlsruhe; der Sekretärin Denise Albring bei TREX, sowie den Sekretärinnen Sonja Seitz und Anja Blancani in Karlsruhe;

und zu guter letzt meinen Eltern, meinem Bruder, meiner Oma und meiner Freundin Elisabeth für ihre Unterstützung.

Vielen Dank!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Searching in huge collections of text documents is a pervasive feature of modern life. Many search engines are deployed to execute queries on collections of documents or records taken from the web, from personal computers, or from large corporate intranets and databases. The amount of information is growing day by day.

In practice, modern computer systems use different types of memory to store their data structures. They are organized in a hierarchy around the CPU, as a tradeoff between performance and size. The lower a storage is located in this hierarchy, the lesser its bandwidth and the greater its access latency from the CPU. Traditionally, search engines store at least the major parts of their indexes on the hard disk, a low-level storage. But now, with growing amounts of RAM, it is possible to step up the hierarchy and to hold all the index data in main memory. Memory latency is about 10^7 times lower and its bandwidth about 10^2 times higher than disk. Nevertheless, the amount of main memory assigned to a single processing core is always lower than the size of a hard disk. But the data can be distributed over a cluster of low-cost machines each indexing a small part of the collection [19]. As a result, we can expect reduced times for answering queries associated with a finer-grained potential for optimization.

1.1 Basics

Index data structures for (disk-based) text search engines have been studied extensively in the past. Across the relevant literature, the *inverted index* has been proved to be the method of choice for fast query evaluation [126, 135].

Figure 1.1 shows a possible scheme for the indexing process of an inverted index based text search engine. Each document (tagged with an integer document identifier) is at first analyzed by the *preprocessor*. The preprocessor *normalizes* the document. It removes all *term separators* (spaces, commas, ...), replaces non-alphanumeric symbols, and unifies upper and lower-case characters. The preprocessor may also perform further analysis, for example linguistic extraction. But in any case, it returns a list of single terms that occur in the document. The set of all these terms in the index is called the *vocabulary*. It is stored in a *dictionary* that maps a unique integer identifier (ID) to each of the terms. The inverted index itself just handles IDs: for each term ID, there is an *inverted list* of document IDs showing in which documents the term appears. When a query has to be resolved, it is preprocessed in the same way as the documents, resulting in a sequence of query term IDs. Actually, it is not mandatory for inverted indexes to assign identifiers to terms. Instead, in some implementations, the dictionary directly stores pointers to the inverted lists. However, as we will see later, it is sometimes of great value to use IDs.

Such *document-level* (or *document-grained*) indexes can answer queries for uncoupled terms. But locating *phrases* in one or more documents needs additional positional information. More precisely, it is required to know the positions where each term occurs in each document. These positions are usually stored either *in-place* after each document ID in the inverted list, or in a separate structure. An index that contains positions is also called a *positional index*.

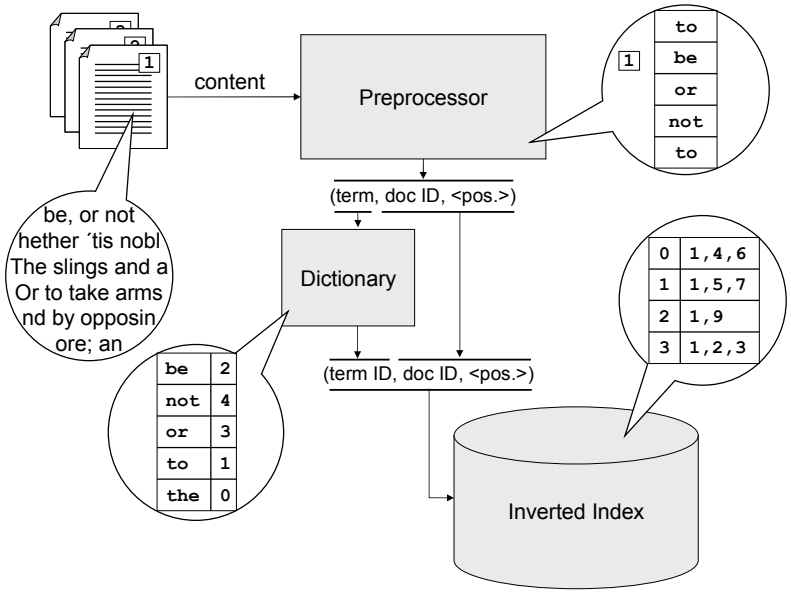


Figure 1.1: *Indexing process on an inverted index based search engine*

1.2 Overview of this Thesis

We summarize previous work dealing with different (low-level) aspects of text search engines in Chapter 2. With main-memory systems in mind, we investigate in Chapter 3 the predominant operation on inverted indexes, which requires intersecting two sorted lists of document IDs. We explore compression and performance of different inverted list data structures and suitable intersection algorithms. Furthermore, we introduce a new inverted list data structure together with a new intersection algorithm that has interesting theoretical properties, and which also outperforms previous approaches in many practical situations.

Using the results, we present in Chapters 4, 5, 6, and 7 the algorithmic core of an inverted index based full text database that allows fast Boolean queries, phrase queries, and document reporting using less space than the

input text. The document reporting is based on a new technique that allows us to reconstruct the document content by mainly using the information stored in our inverted index data structures. In Chapter 6, we present a method for speeding up phrase queries by adding carefully selected phrases to an existing inverted index. In contrast to competing approaches, our scheme is flexible as it can be adapted to fit the specific characteristics of a given text search engine implementation. Our approach outperforms others especially for difficult queries, and it enables us to derive theoretical performance guarantees for arbitrary phrase queries. We also use our phrase index to design a new index scheme for short text documents that completely dispenses with a positional index. It uses less space than a classical positional index, and it even outperforms the positional index for phrase queries.

In Chapter 8, we experimentally show by using real-world inputs that all our approaches maintain their theoretical advantages in practice. Moreover, our experiments give evidence that our inverted index is preferable over purely suffix array based techniques for in-memory (English) text search engines.

We outline methods for processing updates of indexed documents within our data structures in Chapter 9, and we give some implementation details in Chapter 10. Finally, we summarize all our results in Chapter 11.

CHAPTER 2

Related Work

The area of efficient text search has been extensively studied in the past. The most popular data structure for text search engines is the inverted index. This chapter illuminates different aspects of inverted indexes that are covered by previous work. Above all aspects, there is a recurring tradeoff between index compression and query performance. Besides the optimization of inverted indexes, we also outline previous work on storage management of index data structures, as well as on (query) caching, which is not a technique that is restricted to inverted indexes. Finally, we briefly list work dealing with suffix arrays. They have become an interesting alternative for text searching since there are also compressed versions.

2.1 Inverted Index Compression

In the past it was popular to exclude *stop words* or to prune the index (see Carmel et al. [37]). Nowadays it is usual to apply lossless compression to obtain exact and clear results (see Manning et al. [84]). The most common lossless approaches work on sorted inverted lists. They reduce the list of document IDs to a sequence of differences between consecutive document IDs, and apply one out of the multitude of standard techniques for compressing sequences of integers (see, for example, Salomon [104] or Trotman [117]). An index using such a standard compression needs about 10 % of the text size (see, for example, Witten et al. [126]).

Some authors investigated compression schemes adapted to the typical characteristics of document ID distributions in inverted lists. In real-world scenarios, document IDs have the tendency to appear in clusters. The *binary interpolative coding* of Moffat and Stuiver [87, 88] exploits this. It stores the middle element of an inverted list in binary, and handles recursively the two sublists in the resulting narrowed ranges, using only as few bits as needed. This coding scheme can reduce the size of an inverted index by more than 10 % compared to the most compact integer compression methods. Another coding scheme sensitive to variations in the density of inverted lists was proposed by Anh and Moffat [4, 5, 6]. It splits the list in small blocks of fixed-size binary words. The widths of the words in each block are indicated by additional *selector* bits. This requires about 2 % more space than the best standard techniques but can decompress inverted lists 54 % faster. Zhang et al. [130] refine the approach of Anh and Moffat. Zukowski et al. [137] propose to code integers in batches of fixed-width binary words. For each batch, they choose an independent word width b such that the majority of the values to be coded is less than 2^b, and consequently fit into the binary words. All greater values exceeding this range are encoded as *exceptions* using larger words. Blandford [25, 26], Silvestri et al. [111, 110], as well as Blanco and Barreiro [24] give methods to enhance the clustering property of inverted lists by reordering the document IDs. They achieve further compressions of up to 40 % for suitable encoding schemes.

Scholer et al. [108], Zhang et al. [130], and Yan et al. [128] compare different approaches regarding coding/decoding speed and compression.

All of these integer compression methods yield different tradeoffs between space and time. To get a reasonable parameter space in this thesis, we mainly focus on the extremes of this tradeoff. In particular, we use a highly optimized inverted list implementation that uses fixed-width binary words (see Section 3.2.1 and Chapter 10). In contrast to the fastest coding methods mentioned above that also use fixed-width words (combined with other techniques), ours has a simplified control structure, and therefore, it is supposed to be faster in coding/decoding values (probably at cost of compression). To cover the other extreme close-by the most compact representations, we consider Golomb coding, which is a variable-bit length encoding that has been successfully applied to inverted lists (see, for example, [126]). Furthermore, we also use an encoding scheme that falls between these two extremes called *Escaping* (see Section 3.2.1).

Especially the most compact inverted list compression techniques suffer from the constraint that simple lists have to be (almost) completely unpacked for accessing any single value. Therefore, Moffat and Zobel [90] introduced a technique for reducing the number of unnecessarily decompressed values. They divide the inverted lists into a sequence of blocks each containing a fixed number of document IDs. A *skip* pointer in front of each block stores the bit-length of the block. The document IDs within the blocks as well as the value offsets of the blocks can be encoded as differences. To access a value in a list, one can just unpack its block; all preceding blocks can be skipped. Another inverted list structure was proposed by Anh and Moffat [2] as an enhancement of Moffat et al. [92, 3]. They divide a list into subintervals each spanning a power-of-two range. A header indicates the number of document IDs falling in each interval. It is coded in unary, so that the interval of the i-th document ID can be determined by counting all zeros within the first i ones. The document IDs themselves are stored as differences relative to their interval offsets using a fixed-width binary code. Accessing the i-th value requires to add the interval offset to its difference coded in the i-th binary word. Like the representation of Moffat and Zobel

[90], the structure allows values to be skipped while accessing a list, but locating an arbitrary value still requires $\mathcal{O}(n)$ time.

Recent theoretical work of Barbay and Navarro [18] explores techniques to compress permutations of integer sequences. They apply their approach to positional inverted indexes by interpreting the concatenation of all inverted lists as a permutation of the absolute term positions. In addition, they store a pointer to each inverted list resulting in a full and very compact inverted index representation. They show that their representation enables them to execute classical inverted index operations in small time.

2.2 Inverted List Intersections

Intersecting inverted lists is one of the main operation of inverted index based text search engines (see Chapter 3). The problem of sorted set intersections has already been studied decades ago (see Hwang and Lin [64, 65]). Their goal was to reduce the required number of comparisons made between the elements. The simplest algorithm does a binary merge by scanning both sets. For two sets with n and m elements, it needs $n + m$ comparisons in the worst case. If $m \ll n$, it is better to binary search each element of the smaller set in the larger set, requiring $m \lg n$ comparisons which can be improved to $m \lg \frac{n}{m}$ (see Section 3.3).[1] The approach of Baeza-Yates [11, 13] binary searches the middle element of the smaller set in the larger one, and proceeds recursively on the two resulting subsets.

Several authors have also investigated in the general case of intersecting k sorted sets at once. The simplest approach repeatedly intersects the two smallest sets. Using multiple binary search, the algorithm takes $n \lg \frac{n}{k}$ comparisons in the worst case. Demaine et al. [47, 48] and Barbay et al. [17] give algorithms requiring fewer comparisons. The basic idea behind their various approaches is to locate an *eliminator* from one set in all other sets, using a variant of binary search. A greedy technique updates the eliminator *adaptively* after it has been processed.

[1] In this thesis we set $\lg x = \log_2 x$.

The number of comparisons as given in the works cited above is not always a good predictor of running time. In practice, binary search based methods have several disadvantages. In real implementations, binary search based algorithms beat merging only when $n > 20m$ although at $n \approx 20m$ they perform several times fewer comparisons (see Baeza-Yates and Salinger [13]). The main reason is that the linear access pattern of merging causes a good cache behavior. Furthermore, binary searching requires random access which is incompatible with many compression techniques.

While the previous works above do not study both space efficient representations and asymptotically optimal intersection algorithms, Sanders and Transier [106], Culpepper and Moffat [45], Moffat and Culpepper [86], as well as Claude et al. [43] compare different algorithms using different compressed inverted list data structures.

2.3 Index Storage and Caching

Search engines store traditionally at least the major parts of their index structures on hard disks. They use either storage managements that are similar to those of Zobel et al. [136] or the persistent object stores of databases as implemented by Brown et al. [29]. Luk and Lam [80] propose a main memory storage allocation scheme suitable for inverted lists of smaller size using a linked list data structure. The scheme uses variable sized nodes, so it is well suited for inverted lists that are subject to frequent updates. Typically, such approaches are used in combination with larger *main* indexes to overcome the bottleneck of updating search optimized index structures (see, for example, Cutting and Pedersen [46]). Due to the growing amount of random access memory in computer systems, recent work has also considered purging disk-based structures from index design. Strohman and Croft [113] show how *top-k* Boolean query evaluation can be significantly improved in terms of throughput by holding impact-sorted (document-precise) inverted index structures in main memory (see Anh and Moffat [7] for details on impact-sorted indexes).

The advantages of random access memory were also exploited by previous work for improving the query performance of search engines by caching. Brown et al. [29] use the buffer management of their object store for caching inverted lists in main memory. While they use a *Least Recently Used* replacement strategy, Jónsson et al. [67] compare different strategies in interaction with some buffer-optimized query evaluation algorithms. Zhang et al. [130] also investigate different replacement strategies, and show their impact on compressed inverted lists. Tsegay et al. [118] show results for caching only the parts of inverted lists that were actually needed during answering previous queries. Caching of complete query results is widely used and considered by several authors (see Lempel and Moran [71], Xie and O'Hallaron [127], Markatos [85], or Williams et al. [124]). Saraiva et al. [107] combines the approach of caching query results with caching of inverted lists. Long and Suel [78] explore a three-level approach of caching query results, inverted lists, and intersections of frequently occurring pairs. Baeza-Yates et al. [12] compare different caching approaches and give a good overview.

In this thesis, we focus on the basic (low-level) design of inverted index based text search engines. Therefore, our work is orthogonal to caching, and can even be combined with the techniques mentioned above.

2.4 Accelerating Phrase Queries

Decades ago, researchers began to create inverted indexes by indexing phrases instead of single terms (see, e.g., Salton et al. [105]). But building indexes that cover all phrases within a (large) document collection takes enormous amounts of space and time. So authors designed a variety of methods for selecting (short) phrases. All of these approaches focus on choosing phrases that are as *meaningful* as possible. They use syntactical and statistical analyses of a given text to extract suitable candidate phrases for the index (see, for example, Salton et al. [105], Fagan [50], or Gutwin et al. [60]). The main aim of such analyses is to improve the retrieval quality in the absence of a full-text index.

More recently, Williams et al. [123] proposed *nextword indexes*, in which for each term or *firstword*, a list of all successors is stored together with the positions at which they occur as a consecutive pair. With this approach, phrase queries can be answered four times faster than by a classical positional inverted index. However, this technique requires an additional 50 % to 60 % of the space occupied by a standard inverted index. This can be reduced to about 40 % to 50 % using the compression techniques of Bahle et al. [14] at little cost in query performance. To achieve further reductions in memory consumption, Bahle et al. [16] introduced *partial nextword indexes*. A partial nextword index contains only the most common words as firstwords. It can be used in combination with a conventional inverted index acting as fall-back for query terms that are not listed in the nextword index. A partial nextword index enables phrase queries to be evaluated in half the time using 10 % more space compared to an inverted index.

A combination of partial nextword index, partial phrase index, and inverted index was proposed by Williams et al. [124]. An existing query log is analyzed to optimize the indexes. This reduces the average query time to a quarter by indexing the three most common words as firstwords and the 10 000 most common phrase queries. The space overhead for the *three-way index* combination is about 26 %.

Another method, based on the analysis of query logs, was introduced by Chang and Poon [38]. Their approach divides the vocabulary into two sets: *rare* terms and *common* terms. Common terms are the most frequent terms found in the query log; all others are rare terms. For each common term, there is a tree whose root-to-leaf paths contain all phrases from the query log starting with that term and ending with a *terminal* term. Terminal terms are defined by a lexical analysis of the query log: any term that is not a preposition, adverb, conjunction, article, or pronoun is a terminal term. Each leaf points to the inverted list for its phrase. The resulting *common phrase index* shows 5 % improvement in average query time over a partial nextword index with only a 1 % extra storage cost. For long query phrases the query time improvement can reach 20 %.

All these approaches are designed for systems that hold major parts of their indexes on disks. So the cited values apply for disk-based systems.

2.5 Suffix Arrays

A further text index data structure held mostly in main memory, and becoming more and more popular is the suffix array. Originally designed for substring search, it stores all suffixes of a given text and can therefore answer queries consisting of (concatenated) terms very quickly.

Due to their high space consumption, they were first unsuitable for large text indexing. However, various authors have worked on the compression or succinct representation of suffix arrays. For an overview see Navarro and Mäkinen [96] or the Pizza&Chili website [54]. According to González and Navarro [58], they need about 30 % to 150 % of the size of the indexed text — depending on the desired space-time tradeoff. Moreover, they are *self-indexing*, i.e. they can reconstruct the indexed text efficiently. For comparison, a positional inverted index needs between 20 % and 100 % of the text size depending on compression and speed (see, for example, Witten et al. [126]). Unfortunately, inverted indexes can not reconstruct the text, so they require this amount of space in addition to the (compressed) text size.

Puglisi et al. [100] compared compressed suffix arrays and inverted indexes as approaches for substring search. They indexed q-grams, i.e. strings of length q, instead of words to allow full substring searches rather than just simple term searches within their inverted index. They found that inverted indexes are faster in reporting the location of substring patterns when the number of their occurrences is high. For rare patterns they noticed that suffix arrays outperform inverted indexes. Bast et al. have observed that suffix arrays are slower for prefix search [20] and for phrase search [21].

Ferragina and Fischer [51] investigated building suffix arrays on terms. They need just about 80 % of the space of character-based suffix arrays and are twice as fast during the construction. In recent work, Brisaboa et al.

[27] design a new word-based (large alphabet) compressed suffix array, and compare it to an in-memory inverted index implementation. They use an inverted index that is built over the compressed text. The (compressed) text is partitioned into equal-sized blocks rather than documents, such that their inverted lists contain block IDs. In addition, they include skip values in the list for speeding up intersections. For searching a word, they first obtain the list of block IDs from the index, and then they locate the occurrences within the compressed blocks by traversing them linearly. Moreover, they introduced an indexing scheme for their word-based suffix array that allows non-searchable to be detached from searchable content. That is, they indexed just a normalized version of the text and stored the differences between the original input and normalized text separately. Their result was that the word-based suffix array was faster for single-word and phrase searches, particularly when using little memory. However, when more memory is available, the inverted index can at least compete with word-based suffix arrays.

Chapter 3

Set Intersections

In this chapter, we consider sets of integers. We are motivated by *inverted lists* as used by inverted index based text search engines. They contain integer IDs each assigned to a unique indexed document. Whenever a user asks for documents that contain several terms at once (executing an AND query), the search engine has to intersect those lists. Therefore, set intersections are the predominant operation on inverted index based search engines, so they offer the greatest scope for optimizing such systems. In the following, we explore different representations of sets as well as methods for intersecting them. We show how the idea of *randomized sets* can lead to a simple but very efficient data representation. This chapter is mainly based on [106].

3.1 Related Set Operations

3.1.1 Unions and Differences

As mentioned above, set intersections are the most common operation on inverted indexes. Unions and differences are important as well, but they are less frequently requested due to users' querying behavior (see also [112]). Unions of inverted lists are required when resolving an OR query of different terms. Differences have to be calculated when a term should not appear in the results (due to a Boolean NOT operator).

3.1.2 Rank and Select

Rank and select are very basic operations on sorted sets. For a set S of integers they are defined as follows. The rank r of an element $e \in S$ is the number of elements in S which are not greater than e. Selecting i in S means retrieving the i-th smallest element in S. The two operations play an important role in the design of *succinct data structures* [99]. They can also be used for implementing intersections, unions, and differences [45, 44], and they are used for building compressed versions of suffix arrays.

3.2 Set Representations

We consider a set S of n integers, and list the set as a sequence of its sorted elements $\langle s_1, \ldots, s_n \rangle$ throughout this thesis. The simplest representation for S is a list of its n integers each stored in a machine word of its own. In this thesis, *list* denotes a continuous sequence in memory, i.e. an array, unless otherwise stated. Therefore, this uncompressed format requires n machine words, i.e. $32n$ or $64n$ bits on modern computer systems.

3.2.1 Compression

A simple approach to reduce the space consumption of such integer arrays is *bit-compression*. It uses only the bits actually needed to represent the inte-

gers under consideration. As s_n is the maximal value in S, bit-compression reduces the space consumption of S to $n \lceil \lg s_n \rceil$ bits.

We can use Δ-*encoding* to reduce the maximum value to be coded. Instead of plain values, we store differences: the sorted sequence of the elements of S is coded as $\langle d_1, \ldots, d_n \rangle := \langle s_1, s_2 - s_1, s_3 - s_2, \ldots, s_n - s_{n-1} \rangle$. Sometimes, Δ-encoding is referred to as encoding of *d*-gaps. If the elements in S are uniformly spread over the range $[1, s_n]$, bit-compressed Δ-encoding can reduce space consumption to $n \lg \frac{s_n}{n}$ bits. Of course, differences can get as big as $s_n - n + 1$. Thus, in the worst case, Δ-encoding with bit-compression is no better than bit-compression alone.

Variable Bit Length Encodings. This type of coding addresses the above mentioned drawback of bit-compression. Suppose we could encode an integer d using $\lg d$ bits. Then Δ-encoding of S would need $\sum_{i=1}^{n} \lg d_i \leq n \lg \frac{s_n}{n}$ bits. There is a multitude of encoding schemes with different tradeoffs between space consumption and coding/decoding cost [117, 122]. But none of them can reach this overoptimistic space consumption. In the following, we describe two of the most popular schemes.

Escaping. Consider a parameter b. A k-bit integer is split into $K = \lceil k/(b-1) \rceil$ pieces of $b - 1$ bits each and encoded into K blocks of b bits each. All but the last block are marked by a 1 in the most significant bit.

Let us assume that all n integers of the set S are uniformly distributed over the universe U. Then $p = \frac{n}{U}$ is the probability for a given value to be contained in S, and $1 - p$ is the probability *not* to be contained in S. A value d in the Δ-encoding of S means that there is a gap of $d - 1$ integers that are not contained in S. Hence, its probability of occurrence is approximately given by the *geometric distribution* $\Pr(X = d) = (1 - p)^{d-1} p$. We can obtain the probability of occurrence for a value requiring w bits by summing up all single probabilities of the w-th power-of-two range, resulting in the

frequency distribution

$$f_n(w) = n \sum_{d=2^{w-1}}^{2^w-1} \Pr(X = d) = n\frac{p}{1-p} \sum_{d=2^{w-1}}^{2^w-1} (1-p)^d.$$

The latter sum is the difference between two finite geometric series $s_m = \sum_{k=0}^{m} q^k = \frac{q^{m+1}-1}{q-1}$ with $q = 1 - p$. Therefore,

$$\begin{aligned} f_n(w) &= n\frac{p}{1-p} \sum_{d=2^{w-1}}^{2^w-1} (1-p)^d = n\frac{p}{q}\left(\frac{q^{2^w}-1}{q-1} - \frac{q^{2^{w-1}}-1}{q-1}\right) \\ &= n\frac{p}{q}\frac{q^{2^w} - q^{2^{w-1}}}{q-1} = \frac{n}{1-p}\left((1-p)^{2^{w-1}} - (1-p)^{2^w}\right) \\ &= n\left(\left(1-\frac{U}{n}\right)^{2^{w-1}-1} - \left(1-\frac{U}{n}\right)^{2^w-1}\right). \end{aligned}$$

We can determine the space $M_n(b)$ required to store the sequence of the n uniformly distributed Δ-values of S by summing up the space according to the frequency distribution f_n. The maximum possible Δ-value is $U - n$, and therefore,

$$M_n(b) = \sum_{w=1}^{\lceil \lg(U-n) \rceil} f_n(w) \left\lceil \frac{w}{b-1} \right\rceil b.$$

Using $M_n(b)$, the optimal parameter $\hat{b}_n$ for the Δ-encoding of S is given by

$$\hat{b}_n = \min_b M_n(b).$$

Note that in practice b and $\lceil \lg(U-n) \rceil$ are bounded by the width of a machine word. Therefore, the computational complexity for determining $\hat{b}_n$ is low. Escaping with $b = 8$ is also called *variable byte coding* [122]. Due to its use of fixed size pieces, Escaping is amongst the fastest to decompress.

Golomb Coding. Again, consider a parameter b. An integer value $v \geq 0$ is coded in two parts: the quotient $q = \lfloor v/b \rfloor$ and the remainder $r = v - qb$.

The quotient q is coded in unary followed by the remainder r in *truncated binary encoding* over the finite alphabet $\{0, 1, ..., b-1\}$. Truncated binary encoding is a generalization of binary coding where the number of all codewords is not necessarily a power of two (for details see [120]). Golomb coding is very space efficient for the geometrically distributed Δ-values of equally distributed sets (see [104]). Therefore, it is a very popular coding scheme for inverted lists. However, decompression of Golomb codes is much more expensive than decoding escaped lists. According to [126], the parameter b for the Δ-encoding of a uniformly distributed set $S = \{s_i | i = 1..n\}$ with $\langle d_1, \ldots, d_n \rangle := \langle s_1, s_2 - s_1, \ldots, s_n - s_{n-1} \rangle$ is optimally chosen as

$$b = \ln 2 \Big(\frac{1}{n} \sum_{i=1}^{n} d_i \Big) = \ln 2 \frac{s_n}{n}.$$

Using this parameter, Golomb codes have clearly limited memory requirements. A complex analysis is given by [89], but the following theorem and its proof already show this nice property.

Theorem 1. *Let $S = \{s_i | i = 1..n\}$ be a set of n integers and $\langle d_1, \ldots, d_n \rangle := \langle s_1, s_2 - s_1, s_3 - s_2, \ldots, s_n - s_{n-1} \rangle$. Storing the Δ-values $d_{1..n}$ of S using Golomb encoding with parameter $b = \lceil \ln 2 \frac{s_n}{n} \rceil$ requires not more than $n(1 + \lceil \lg \frac{U}{n} \rceil + \frac{1}{\ln 2})$ bits.*

Proof. Let v be an integer value, $q = \lfloor v/b \rfloor$, and $r = v - bq$. The unary coding of q requires $\lfloor v/b \rfloor + 1$ bits and the truncated binary encoding of r is not larger than $\lceil \lg b \rceil$ bits. Therefore, the Golomb representation of v (using parameter b) is not larger than $q + 1 + \lceil \lg b \rceil$ bits. Hence, the maximum space

in bits required to store the sequence of Δ-values $\langle d_1, d_2, d_3, \ldots, d_n \rangle$ is

$$\begin{aligned}
M_{max} &= \sum_{i=1}^{n} \left(\left\lfloor \frac{d_i}{b} \right\rfloor + 1 + \lceil \lg b \rceil \right) = n(1 + \lceil \lg b \rceil) + \sum_{i=1}^{n} \left\lfloor \frac{d_i}{b} \right\rfloor \\
&\leq n(1 + \lceil \lg b \rceil) + \sum_{i=1}^{n} \frac{d_i}{b} = n(1 + \lceil \lg b \rceil) + \frac{\sum_{i=1}^{n} d_i}{b} \\
&= n(1 + \lceil \lg b \rceil) + \frac{\sum_{i=1}^{n} d_i}{\left\lceil \ln 2 \frac{1}{n} \sum_{i=1}^{n} d_i \right\rceil} \\
&\leq n(1 + \lceil \lg b \rceil) + \frac{\sum_{i=1}^{n} d_i}{\ln 2 \frac{1}{n} \sum_{i=1}^{n} d_i} \\
&= n(1 + \lceil \lg b \rceil + \frac{1}{\ln 2}) \leq n(1 + \left\lceil \lg \frac{U}{n} \right\rceil + \frac{1}{\ln 2}).
\end{aligned}$$

□

For implementing a Golomb encoder, Theorem 1 can be very helpful. It allows us to predict the maximum memory requirement of a given set in advance, so that no resize (and thus no copy) operations are required while encoding a list.

3.2.2 Two-Level Representations

The big drawback of Δ- and variable bit length encodings is that they do not provide random access. Since the width of a value is not known until it has actually been decompressed, accessing the i-th element requires us to decode all $i - 1$ preceding values. Two-level representations can be used to overcome this limitation of *linear* encoded lists. In addition to the encoded values of a set S, a (random-accessible) *top-level* data structure t stores every B-th element of S together with its position in the encoding (see, for example, [2, 106, 45]). We can now access the i-th element of S by proceeding to the position stored in the $\lfloor i/B \rfloor$-th field of t and decoding just the next $i - B \lfloor i/B \rfloor$ values. The smaller B is chosen the faster random accesses are, but also the more space is required for storing t.

3.3 Selected Intersection Algorithms

In the following, we consider two sets R, S of integers represented by a sequence of their elements with $m = \|R\|$ and $n = \|S\|$. Without loss of generality, we assume $m \leq n$.

3.3.1 Zipper

The simplest intersection algorithm scans both sets as in a binary merging operation. It needs time $\mathcal{O}(n + m)$ with a quite small constant factor. Hence, Zipper is very good if the sets have about equal size. Moreover, Zipper harmonizes well with linear encodings.

3.3.2 Variants of Binary Search

A simple, (suboptimal) algorithm that is sublinear in the size of the greater set S scans the smaller set R and locates each element e of R in S using binary search. This takes time $\mathcal{O}(m \lg n)$ with constant factors worse than in algorithm Zipper. Binary search can be improved to an asymptotically optimal $\mathcal{O}\left(m \lg \frac{n}{m}\right)$ by exploiting that R is sorted [64]. If $R[i-1]$ was located at position j in S, i.e., $j = \max\{k \mid S[k] \leq R[i-1]\}$, then $R[i]$ must be located at a position $j' > j$ in S. If $j' = j + \delta_i$, we can find an upper bound on δ_i in time $\mathcal{O}(\lg \delta_i)$ using *exponential search* [23]—we double an estimate δ' of δ_i until $S[j + \delta'] \geq R[i]$. We can then locate $R[i]$ exactly using binary search in time $\mathcal{O}(\lg \delta_i)$. Overall, we get execution time $\sum_{i=1}^{m} \mathcal{O}(\lg \delta_i) \leq \mathcal{O}\left(m \lg \frac{n}{m}\right)$. Since we perform two searches for each element, the constant factors are even worse than for binary search. A somewhat faster variant of binary search by Baeza-Yates [11, 13] therefore uses the *divide-and-conquer* principle. The middle element $S[\lfloor n/2 \rfloor]$ of S is located in R using binary search and then intersection proceeds recursively on the respective left and right halves of R and S. All these algorithms have the disadvantage that binary search is incompatible with linear encodings.

3.3.3 Skipper

Using a two-level representation we can also run a variant of Zipper that scans the top level data structure and delves down into the lower level whenever necessary. This avoids the complicated control structure of binary search and allows us to use linear encodings also for values and references stored in the top level data structure. A similar scheme where all the information is embedded into a single bit sequence has been proposed and implemented in [90] (see Chapter 2).

3.4 Randomized Sets

3.4.1 Representation

Consider a set $S = \{s_i | i = 1..n\}$ where all elements are uniformly distributed over $[1, U]$. We split the range into *buckets* based on the most significant bits. Let B denote a parameter that roughly controls the average bucket size and $w = \lceil \lg \frac{UB}{n} \rceil$. Then, the i-th bucket b_i stores the sequence of value residues $\langle d_j \bmod 2^w : \lfloor d_j 2^{-w} \rfloor = i \rangle$. Due to randomization, and because of the ceiling function in the term above, the average size of each bucket is between $B/2$ and B. There can also be smaller and larger buckets, but basic balls-into-bins theory guarantees with high probability that no bucket size is very much larger than B. The encoding of all buckets can be stored consecutively in one stream of bits b using an arbitrary coding scheme. A top level data structure t provides the information leading from the most significant bits $i = \lfloor d 2^{-w} \rfloor$ of a value d to the i-th bucket. In the simplest case, t is an uncompressed array of indexes into b. We call this two-level data structure a *Lookup list*.

Using a fixed-width encoding in the bottom level, Lookup lists also support rank and select operations in acceptable time. Selecting the k-th element takes $\mathcal{O}(\lg(n/B) + B)$ time by binary searching the top level for its bucket position i, and taking the $(k - i)$-th element in the bucket. Ranking an element requires $\mathcal{O}(B)$ time by branching into the bucket and scanning it linearly. Using variable-width encodings, Lookup lists supporting fast

rank and select operations are easy to obtain by adding rank information to the top level. That is, we add $r_i = \sum_{j=0}^{i} \|b_j\|$ for the i-th bucket. As a side effect, we can also use *variable Golomb coding* for the buckets without storing any parameters. The range of a bucket is given by 2^w and the number of values coded in a bucket is given by $r_i - r_{i-1}$. Due to uniformly distributed values, we can estimate a suitable Golomb parameter (in accordance with Section 3.2) by approximating the average Δ-value through $(r_i - r_{i-1})/2^w$. We do not need to store this parameter as we can make exactly the same estimation while decompressing a bucket.

3.4.2 Intersection

Figure 1 gives pseudo-code for the intersection of two Lookup lists. The smaller set R is traversed by the outermost loop. The current element d of R is disassembled into its most significant bits h and its least significant bits ℓ such that h can be used as an index into the top level data structure of S. Thus h determines the bucket of S where we have to look for elements matching ℓ. We now have to scan the h-th bucket of S for an element ℓ' that coincides with ℓ. If this is not the bucket we have been scanning before, we have to scan it from the start. Otherwise, we continue scanning from where we left off. There are three ways to stop scanning: First, if we reach the end of the bucket. Second, if we encounter an element $\ell < \ell'$. Then we know that d cannot be in S and go to the next iteration of the outer loop. Third, we have found d in S (indicated by $\ell = \ell'$), we output it, and we continue with the next element of R. We call this algorithm *Lookup*. Note that R can also be present in another compressed (or uncompressed) form.

3.4.3 Analysis

In the following, we give time and space analyses of Lookup lists and their intersection algorithm.

Algorithm 1 High level pseudo-code for our intersection algorithm

Require: list R, list S

```
O ← ∅                                              {output}
i ← −1                                             {current bucket in S (now a dummy)}
for all d ∈ R do                                   {unpack R on-the-fly}
    h ← ⌊d2^{−w_S}⌋                                {d would fall into the h-th bucket of S}
    ℓ ← d&2^{w_S} − 1                              {least significant bits of d}
    if h > i then                                  {do we enter a new bucket?}
        i ← h                                      {set h-th bucket of S as current bucket}
        j ← t_S[i]                                 {read start of current bucket from toplevel of S}
        e ← t_S[i + 1]                             {read end of current bucket from toplevel of S}
    end if
    while j < e do                                 {current bucket not yet exhausted}
        ℓ′ ← b_S[j]                                {unpack value from bottom of S}
        if ℓ ≤ ℓ′ then
            if ℓ = ℓ′ then
                O ← O ∪ d                          {d is in R and S}
            end if
            break while loop
        end if
        j ← j + 1
    end while
end for
return O
```

Theorem 2. *Algorithm* Lookup *runs in expected time* $\mathcal{O}(m + \min\{n, Bm\})$.

Proof. (Outline) The total time spent outside the inner loop is $\mathcal{O}(m)$. Since each bucket of S is scanned at most once, there is also a bound of $\mathcal{O}(m + n)$ on the time spend in the inner loop. Another bound for the time in the inner loop is $\sum_{d \in R} \left\| b_{\lfloor d 2^{-w} \rfloor} \right\|$ which makes the conservative assumption that for each $d \in R$ the entire bucket of S corresponding to d has to be scanned. □

Theorem 3. *Using bit-compressed* Δ*-encoding and assuming a uniform random distribution over* $[1, U]$*, our representation needs*

$$n\left(\lg\frac{U}{n} + \lg\min\{B, \lg n\} + \mathcal{O}\left(1 + \frac{\lg n}{B}\right)\right)$$

bits for storing a list of size n *with high probability.*

Proof. (Outline) There are $\Theta\left(\frac{n}{B}\right)$ buckets. Storing a pointer to a bucket needs $\mathcal{O}(\lg n)$ bits. The largest difference between two bucket entries can be bounded in two ways — the bucket width $\Theta\left(B\frac{U}{n}\right)$ and the largest difference between any two consecutive list entries. The latter quantity is well known from probability theory: We throw n balls into U (different) bins and ask for the longest consecutive stretch of unoccupied bins. Using standard techniques, it can now be shown that the largest difference is $\mathcal{O}\left(\frac{U}{n}\lg n\right)$ with high probability. The two bounds can be combined to a bound of $\lg\min\left\{\Theta\left(B\frac{U}{n}\right), \mathcal{O}\left(\frac{U}{n}\lg n\right)\right\} = \lg\frac{U}{n} + \lg\min\{B, \lg n\} + \mathcal{O}(1)$ bits per list element. □

In this bound and its proof we can see that using $B = \Omega(\lg n)$ avoids empty buckets (and their dispensable pointers in the top level) with high probability. For smaller B, it is hence also quite likely that there are some buckets with only one element, which will set a maximum difference close to the range of the residues (and the width of an entire bucket). For bit-compression, this means that using Δ-encoding in this case actually makes no sense.

Theorem 4. *Using Golomb Δ-encoding and assuming a uniform random distribution over $[1, U]$, our representation needs*

$$n\left(\lg\frac{U}{n} + \mathcal{O}\left(1 + \frac{\lg n}{B}\right)\right)$$

bits for storing a set of size n.

Proof. (Outline) There are $\Theta\left(\frac{n}{B}\right)$ buckets. Storing a pointer to a bucket needs $\mathcal{O}(\lg n)$ bits. The average Δ-value is $\Theta\left(\frac{U}{n}\right)$. Together with Theorem 1, we get $n\left(\lg\frac{U}{n} + \mathcal{O}(1)\right) + n\mathcal{O}\left(\frac{\lg n}{B}\right)$. □

By comparing Theorems 3 and 4, we can see that a variable bit length encoding can harness the full power of Δ-encoding. It replaces the term $n \lg \min\{B, \lg n\}$ by $\mathcal{O}(n)$. The term $n \lg \frac{U}{n}$ is unavoidable since it already shows up in the information theoretic lower bound $\lg\binom{U}{n} \approx \lg\frac{U^n}{n!} = n\lg\frac{U}{n} + \Theta(n)$. The term $\mathcal{O}\left(n\frac{\lg n}{B}\right)$ can be made arbitrarily small by choosing a sufficiently large bucket size. However, this comes at the cost of increased running time when intersecting lists of asymmetric size. Still, for all but the longest lists the term $n \lg \frac{U}{n}$ will dominate space consumption.

For very long lists which need only a few bits for encoding each entry of a bucket, it might pay off to replace our two-level data structure with a three-level data structure. We could use *metabuckets* of size $\mathcal{O}(\lg n)$ and bottom-level buckets of size $\mathcal{O}(1)$ whose position within a metabucket can be encoded using $\mathcal{O}(\lg\lg n)$ bits with high probability. In practice, this might boil down to using a single byte for encoding the relative position of a bottom-level bucket.

3.4.4 Achieving Randomization

All of the ideas concerning Lookup lists above rely on the assumption of uniformly distributed values. But in practice, document IDs are not necessarily uniformly distributed. Therefore, we show how to achieve (pseudo) randomization.

Pseudo Random Permutations

For simplicity, we explain the algorithm for the case that $U = 2^{2u}$ for some integer u. The general case was explained in [79] and [49]. Consider the *Feistel permutation* $\pi_i(u \circ v) := v \circ (u \oplus f_i(v))$ where $\circ$ denotes the concatenation of u-bit bitstrings, $\oplus$ denotes bit-wise exclusive-or, and where f is a random mapping $0..2^u - 1 \rightarrow 0..2^u - 1$. Note that f can be efficiently implemented by filling a lookup table of size $\sqrt{U}$ with random values. As [79] showed, chaining four Feistel permutations gives a permutation which is pseudo-random even in a cryptographic sense. Our implementation chains only two Feistel permutations.

Load Balancing by Randomization

Any large full-text database will nowadays use multiple processors. Currently, a typical configuration is a cluster of low-cost servers each equipped with one or two multi core processors. The easiest way to exploit p processing elements (PEs) is to assign about U/p documents to each PE [36, 1]. Now queries simply go to all the PEs. Doing this naively, based on ordered document IDs, will lead to poor load balancing, because document sizes, and term appearances will be correlated with the document IDs. Randomization largely dissolves all these problems. We can just map document i to PE $i \bmod p$ (round robin). We get strong probabilistic guarantees not only that every PE is responsible for a similar overall amount of data but also that *every* list of the inverted index is split almost evenly. Note that trying to do this deterministically would be more complicated. Using balls-into-bins arguments one can see that a given list of length n gets split into sublists of size $\frac{n}{p} + \mathcal{O}(\sqrt{\frac{n}{p} \lg p})$ with high probability [102]. In particular, long lists, which cause the largest query times, will get split very evenly.

Stick to Determinism

Randomization gives us certain guarantees for storing and intersecting inverted lists. But as shown in [87], non-uniformity present in real-world

inputs can also have advantages. In practice, terms tend to appear very often in consecutively indexed documents before they disappear for a while, and finally they occur again. This *clustering* effect can be exploited to improve inverted index compression. Moreover, [25] shows that by carefully choosing document IDs, inverted lists can be made even more clustered, and hence more compact. Also, several authors have mentioned that load balancing can be achieved without randomization of documents by replicating the index data on several PEs [19, 35].

Using a deterministic distribution of document IDs, it is wasteful to store in each Lookup list a top level that covers the entire universe U of all document IDs. Instead, we build the Lookup list of a set $S = \{s_i | i = 1..n\}$ according to the range $[0, s_n - s_1]$, and store the offset s_1 in addition. To make Lookup lists even more useful for deterministic distributions, we can also use variable Golomb encoding as described in Section 3.4.1. It adapts the parameter of buckets depending on whether they span sparse or dense regions. The theoretical guarantees do not hold for deterministic data, but our experiments show that Lookup lists perform well on non-randomized values too.

CHAPTER 4

Compressed In-Memory Indexes

In this chapter, we describe how to build an in-memory text search engine using inverted index based data structures. Our text index mainly focuses on the following three operations: Boolean AND queries, phrase queries, and *document reporting*. The first two operations are well known from everyday usage of web search engines. The third operation, document reporting, is less established, as many search engines do not store the indexed texts itself. In contrast, they contain just pointers to documents stored at an external location (for example a disk or the network). But document reporting out of the search engine is very important in database-like environments. Furthermore, we show how document-grained indexing can be applied to suffix arrays. This chapter is mainly based on [114].

4.1 Compressed Inverted Indexes

Our inverted index design is modular, and consists therefore of several parts. Besides space efficiency, we attach great importance to query performance. Because our index is highly (and carefully) optimized, and its success often depends on details, we also give, on some points, practical considerations achieved through our implementation.

4.1.1 A Document-grained Inverted Index

Zipf [131, 132] has observed that term occurrences in text collections follow an inverse power law with an exponent close to one. The frequency p of a term is inversely proportional to its rank r in the frequency table: $p \propto 1/r^{\alpha}$ with $\alpha \approx 1$. Figure 4.1 shows the distribution of term frequencies ordered by the rank for an example text collection. We can see that Zipf's law does not necessarily match the characteristics of a real-world collection, but for some ranges it provides a good approximation. Roughly speaking, *Zipf's law* says that on the one hand, a major part of all term occurrences are distributed among just few terms, and on the other hand, there are many terms that occur very rarely. This suggests usage of different types of inverted lists depending on their very different lengths [114, 86].

Therefore, we use the following scheme. For each term ID we store either a pointer to an inverted list or — if the term occurs in a single document only — just a document ID. We distinguish between two types of inverted lists. The first type is used for all terms that occur in less than K documents where K is a tuning parameter. In our implementation we use Golomb Δ-encoding to store these *small* lists very compactly. For the remaining terms, we use the Lookup data structure, and call those lists *large* inverted lists. The distinction between single values, small lists and large lists offers two advantages. First, the majority of inverted lists are highly compressed (without any auxiliary information). Second, very large lists do not have to be scanned completely whenever they are involved. The

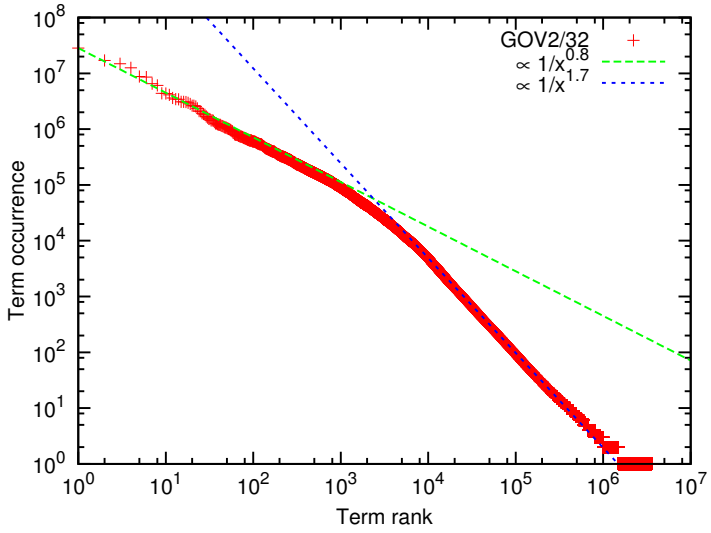


Figure 4.1: *Term occurrences*

following lemmas stress the importance of different list types, and show that there is actually a major portion of single values and small lists.

Lemma 1. *Assuming the classical Zipfian distribution of M terms over U documents, our index contains at least $M - \frac{U}{\ln(M)}$ single value entries.*

Proof. The classical Zipfian law says that the r-th most frequent term of a dictionary of M words occurs $\frac{U}{rH_M}$ times in a collection of U documents, where H_M is the harmonic number of M. Claiming an occurrence frequency of 1, we get $r_1 = \frac{U}{H_M}$ as the order of the first term that occurs just once. As the total number of terms in our index is M, we have $M - \frac{U}{H_M}$ unique terms. According to [129], $H_n - \ln(n) - \gamma < \frac{1}{2n}$, where γ is Euler's constant. Therefore, we have $M - \frac{U}{H_M} > M - U(\frac{1}{2M} + \ln(M) + \gamma)^{-1} > M - \frac{U}{\ln(M)}$. □

Lemma 2. *Our document-grained index contains at least $(1 - \frac{1}{K})\frac{U}{\ln(M)}$ small lists.*

Proof. As in proof of Lemma 1, we can obtain the order of the first term whose occurrence frequency is equal to K using the Zipfian law, $r_K =$

$\frac{N}{KH_M}$. By subtracting this rank from that of the unique term r_1, we get the number of terms that occur more than once but less than or equal to K times. Thus, $r_1 - r_K = (1 - \frac{1}{K})\frac{U}{H_M}$ and according to [129] $r_1 - r_k > N(1 - \frac{1}{K})(\frac{1}{2M} + \ln(M) + \gamma)^{-1} > (1 - \frac{1}{K})\frac{N}{\ln(M)}$. □

The tuning parameter K controls a tradeoff between index size and querying performance. The larger K is, the smaller the index is, but also the slower query response times are. In practice, K should be chosen large enough, such that in any case the top level contains more useful data than it produces space overhead, i.e. $\frac{K}{B} \lg K > c$ where c is the implementation specific overhead in bits. It is essentially given by the difference between small and large list object sizes. In our implementation we use $c = 96$.

4.1.2 Storing Positional Information

As an extension of our document-grained inverted index we store the positional information of the terms in a separate structure. The separation between the two data structures yield different advantages: for cases in which we do not need the functionality of a positional index, we can easily switch off this part (to save space) without any influence on the document-grained data. Furthermore, Boolean queries are very cache efficient as the inverted lists are free of other information than the document IDs themselves (see also [8]). In the following, we propose two methods for indexing the positional information.

Positional Indexes. We use a *positional index* that contains for each term-document pair (t, d) a list $l_{(t,d)}$ of positions where t occurs in d. Similar to the document-grained index, our positional index distinguishes between different list data structures depending on the number of term positions. Our index stores for each term t the lists $l_{(t,i)} \forall i : t \in i$ in a contiguous stream. Again, we use a vector where we store pointers to such a stream, or a single position (if the term occurs only once within the index). Positions of terms that occur more often, but in just one single document, are stored in a Δ-Golomb coded stream. Positions of terms that appear in

multiple documents, but just once in each, are stored (random accessibly) bit-compressed. They are ordered according to the ranks in the corresponding inverted list of the document-grained index. For all the other terms, we use *indexed lists*. An indexed list is a two-level data structure whose top level points to buckets at the bottom level containing an arbitrary number of values each. In our case, such a bucket contains a list of positions. We again use the inverted index ranks for accessing the buckets: for each term t, the i-th entry in the top level points to the list $l_{(t,d)}$, where d is the i-th document in the inverted list of t. Because we work in main memory, we can jump to any term positions *actually* needed at little cost.

Term Lists. Maintaining a positional index for a collection that contain just short documents is prohibitive in space. In those cases, the overhead for keeping a huge amount of very small position lists is tremendous. Instead, we propose to store the term positions implicitly by keeping a sequence of term IDs for each document. Our implementation uses Golomb coded lists with individually chosen parameters. This is not only more space efficient for short documents, but also its document reporting is simpler (and faster). We give details for that in Chapter 7. The phrase query evaluation using term lists is very different than using positional indexes. We describe it for both approaches in Chapter 6.

4.1.3 Further Issues

A Simple Dictionary

As usual for inverted index based text search engines, we use a dictionary that maps normalized terms to term IDs and vice versa. Since this part of a search engine is not our main focus, we use a simple uncompressed dictionary. It stores the normalized terms in a memory-optimized hash map [109] using the hash function of [63]. The inverse mapping is done via a separate vector that is indexed by the term IDs and contains pointers to the respective terms. Certainly, our dictionary is not very fancy, but as it holds uncompressed terms, it certainly does no harm to our reconstruction

process in Section 7. A more sophisticated dictionary was for example proposed in [55].

Index Construction and Memory Management

The inverted index construction on in-memory search engines is straightforward. We assign all IDs in the order of the term and document appearance in the text collection to be indexed. We loop through the terms of each document, append the current document ID to their inverted lists, and add each term occurrence to the positional index (or to the term lists). We use temporary data structures during the indexing process for the inverted index as well as for the positional index. Both contain lists that are growing whenever new documents are added. So a bit-compressed vector storing Δ-values that dynamically adapts its word width to fit its largest value seems to be a suitable choice. It provides a good tradeoff between updating performance and compression.

Memory management turns out to be crucial for the speed and the peak memory usage during index construction. To avoid wasteful fragmentation and to reduce the number of expensive system memory allocation calls, we largely organize the memory occupied by inverted lists in blocks of 512 KB each. Figure 4.2 shows a coarse scheme of our memory management. Analogous to the static index structure, we distinguish between different-sized inverted lists. The lists are growing during construction, and we adapt their memory organization accordingly. The smallest possible representation is a plain value stored in a simple array — the *inverted list directory*. For lists that contain more than a single value, the array stores a pointer to a list object. Each of these objects has a data field containing the actual list values. The field changes depending on the size of the list. Lists that require less than the size of a pointer (in our case less than 8 B) use *in-place* storage within the objects. Lists that are larger than that get one or several continuous memory snippets with a power-of-two size between 16 B and 512 KB, where we combine small equal-sized snippets less than 16 KB in 512 KB blocks. These blocks maintain a free list consisting of a bit for each

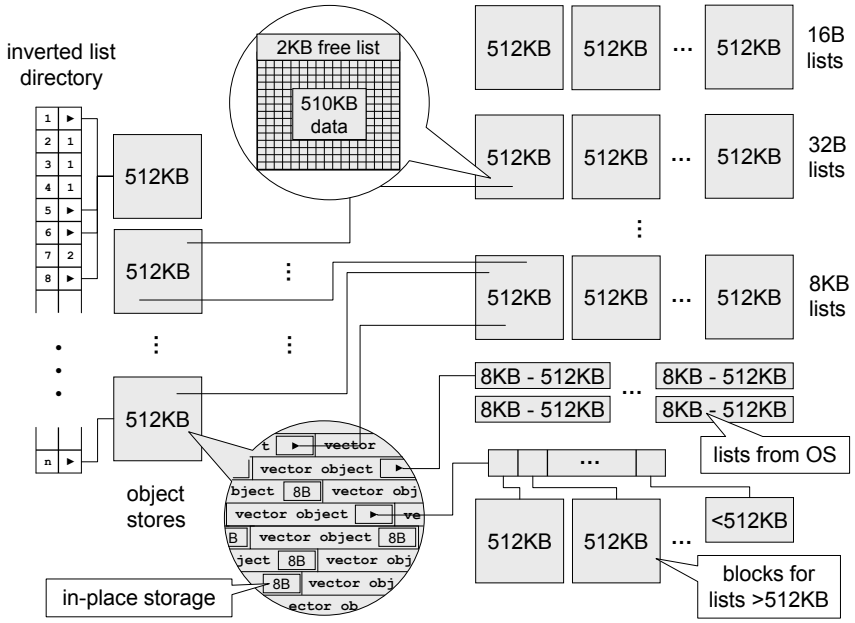


Figure 4.2: *Organization of inverted list memory during index construction*

snippet indicating whether the snippet is currently in use or not. Snippets larger than 8 KB are managed directly by the operation system. Whenever the data field of a list has to be expanded, we copy the old content into a free snippet with the next larger power-of-two size (possibly marking the old snippet as freed). Lists larger than 512 KB are organized using several 512 KB memory blocks and an additional smaller snippet that can grow (not exceeding 512 KB). This avoids the need to copy all the content of a very large list when its capacity is exhausted. We choose the size values and the types of the data fields so that they fit the Zipf-like distribution of the inverted list lengths: typically, there are many small lists and just a few large lists.

After the last document has been added, the temporary lists are converted into the static data structures, and one block after the other is freed. The memory management of the static structures turned out to be crucial for obtaining *actual* space consumption close to what we would expect from summing the sizes of the many ingredients. Most data is again stored in blocks of 512 KB each. Lists that are greater than such a block get their own contiguous memory snippet. Within the blocks, we use word-aligned allocation of different objects. We tried byte alignment but did not find the space saving worth the computational overhead.

4.2 Indexing Documents into Suffix Arrays

Suffix arrays do not support multi-document indexing innately. Therefore, we index the normalized text of all documents as a continuous string. We separate the normalized terms by spaces and the documents by a pair of spaces. This allows us to identify document bounds and to exclude hits that overlap document bounds while querying for phrases (see Section 6.5). The use of two space separators (in the following indicated by '␣') is an advantage over inserting a single special separator (for example '$') for marking documents: we do not need to distinguish between the cases of a hit at the bounds or within a document during query time. For instance, if we want to search for a single term t, we can just search for '␣t␣', and we do not have

to search for '_t_', '\$$t$_', and '_$t$\$' simultaneously. In the example, we see that we search for '_t_' instead for 't'. This is because we do not want to get results where t is postfix or prefix of another term ('st_' or '_ts'). As suffix arrays return absolute text positions as query results, we store the start positions for all documents in a Lookup list. Using their rank functionality, we can map absolute suffix array positions to document IDs (assigned in indexing order) very quickly.

CHAPTER 5

Boolean AND Queries

Inverted indexes are by design well suited to answer AND queries. As already mentioned in Chapter 3, they are the predominant operation of text search engines, and they correspond to an intersection of a set of inverted lists. Chapter 2 has shown that the problem of intersecting inverted lists has been studied extensively. Furthermore, we have explored some intersection algorithms in Chapter 3. This chapter briefly describes how we evaluate AND queries consisting of more than two terms on our index structures and on suffix arrays.

5.1 Query Evaluation on Inverted Indexes

We evaluate AND queries of $k > 1$ terms by a series of $k - 1$ pair-wise intersections. Beginning with the two smallest lists, the remaining $k-2$ lists are intersected in increasing order with the current result set. This approach is widely used in practice [48, 17, 45, 86]. In Chapter 2, we have seen more sophisticated algorithms to intersect a set of k inverted lists. However, we believe that this algorithm has a decisive advantage, because it is more cache-efficient than the others. Depending on the involved inverted list data structures, we use Zipper or Lookup for the pair-wise intersections.

5.2 Boolean AND Queries on Suffix Arrays

Suffix arrays do not support AND queries innately. Therefore, we use the following algorithm to resolve those queries. First, we locate the occurrences for each query term by a substring search in the suffix array. Second, we map the result list of the least frequently occurring term to a list of document IDs. Third, we insert the list in a binary search tree (`std::map`) and incorporate the results of the next frequently occurring term by performing a tree lookup for each of its items. Finally, we put all hits in a second tree and swap them as soon as we have checked all of the items. We repeat this until all term IDs are processed.

We also tried two other approaches for Boolean AND queries. The first one was to build a hash table from the shortest document ID list using `stdext::hash_map` and to check there for the IDs of the remaining lists. The second one was to sort all lists and to intersect them like the inverted lists. However, these two alternatives could not compete with the previous one. Anyway, as we will see later in Chapter 8, the major part ($> 99\,\%$) of the querying time is spent during suffix array operations.

CHAPTER 6

Phrase Queries

The advantage of entering a phrase query in a search system is that it specifies certain relations between the words in the phrase, and this constrains the result set more narrowly than individual AND-combined search terms can do. In the following, we explain how phrase queries can be resolved using the data structures of the previous sections. Moreover, we show different speed up techniques for accelerating phrase queries on inverted index based search engines. We also show how we can profit from these techniques while building an efficient index for short documents. Parts of this chapter are based on [115].

6.1 Query Evaluation using Positional Indexes

The classical approach for resolving phrase queries is to use a positional index as also described in Section 4.1.2. It stores for each term and document a list of all positions where the term occurs in the document. A phrase query can be answered by intersecting all these position lists — n lists per document for a query with n terms — regarding the offset given by the term positions in the query phrase.

As we did for Boolean AND query evaluation, we use a pair-wise processing strategy. We start a phrase search $t_1 \cdot t_2 \cdots t_n$ by sorting the phrase terms $t_{1..n}$ according to their frequency, i.e. their inverted list lengths. In the following, $\pi(i)$ denotes the position in the query phrase of the term with the i-th smallest list. We intersect the document-grained inverted lists $l_{t_{\pi(1)}}$ and $l_{t_{\pi(2)}}$ of the two least frequent terms $t_{\pi(1)}$ and $t_{\pi(2)}$ (using algorithm *Lookup* if applicable). During the intersection, we keep track of the current *rank* of each list. That is, for each result document d, we also record the positions r and s indicating where d is located in $l_{t_{\pi(1)}}$ and $l_{t_{\pi(2)}}$, respectively. The list of all those result triples (d, r, s) contains also at least all hits of the phrase query (and probably some other occurrences). To identify actual hits in this set of candidates, we need to match their position lists. Therefore, we iterate through the list of triples (d, r, s), and use the ranks r and s to retrieve the position lists $l_{(t_{\pi(1)},d)}$ and $l_{(t_{\pi(2)},d)}$ for the document d from the positional index structure according to Section 4.1.2. We intersect the lists $l_{(t_{\pi(1)},d)}$ and $l_{(t_{\pi(2)},d)}$ where we add an offset of $o_1 = 1 - \pi(1)$ to all values in $l_{(t_{\pi(1)},d)}$ and an offset of $o_2 = 1 - \pi(2)$ to all values in $l_{(t_{\pi(2)},d)}$. We use a simple binary merge algorithm similar to Zipper for this intersection. The result is a list of pairs consisting of a document d and a position list l showing where the terms $t_{\pi(1)}$ and $t_{\pi(2)}$ occur in the same order and with the same distance as in the query phrase. Finally, we continue with intersecting the list of result documents with the document-grained inverted list of the next frequent term $t_{\pi(3)}$ and with matching their position lists. We repeat this procedure until all terms are processed.

6.2 Partial Phrase Indexes and Nextword Indexes

Unfortunately, query evaluation using positional indexes is computationally expensive. Therefore, attempts have been made to speed up phrase queries on inverted indexes by adding further information to their data structures (see Section 2.4). These attempts have the disadvantage that they consume more space. This disadvantage is especially critical for main-memory search engines. Although modern hardware configurations feature ever more random access memory, memory capacity is still significantly more restricted and more expensive than disk capacity in classical search engine systems. Thus, it is essential to optimize the "return on investment" when investigating approaches that need extra space.

Long ago, Gutwin et al. [60] observed that storing pre-calculated results of a set of phrase queries can significantly reduce the average response time of a search system. But this applies only if queries currently entered were actually considered before. Using all possible phrases is prohibitive in space and time [124]. An alternative is to store only a subset. The problem of selecting the most useful phrases leads to the concept of caching previous query results, which then involves updating strategies. Section 2.3 has summarized previous work that addresses this issue for any type of query. A more simple but competitive approach for phrase queries proposed in [124] is to use a *partial phrase index*. This is essentially a positional inverted index whose vocabulary is enriched by the most frequently occurring phrases of some query log. The single-term part of the positional inverted index acts as fall-back if a phrase that is searched for was not indexed. However, examining query logs, we can observe that often *hype words* appear for a while and quickly disappear again [112]. This dynamical aspect of user behavior regarding search terms may be unpredictable and makes it difficult to optimize partial phrase indexes sufficiently. This means that the (static) partial phrase indexes can get suddenly out-of-date resulting in a miserable worst-case behavior of the search system.

The *nextword index* proposed by Williams et al. [123] is (in a high-level point of view) a phrase index that stores all phrases with a length of two

terms. However, such an index is still prohibitive in space and time for large amounts of data. The *partial nextword index* of Bahle et al. [16] contains, therefore, only the phrases whose first word is a *common* or a top frequently occurring word.

Williams et al. [124] showed that a *three-way index combination* of an inverted index, a partial phrase index, and a nextword index can improve on the performance of its components.

6.3 Two-Term Phrase Indexes

Whereas many approaches for speeding up phrase queries mainly use query logs, we present a flexible framework that works out of the box and needs only the (static) information contained in the index of single terms. This is an advantage over other approaches because query logs (a) are not available for many real-world applications and (b) are just a snapshot of user behavior, which often changes unpredictably. Despite this, we show in Section 8.5 that our method is competitive in terms of query performance and can even improve on other approaches at least for *difficult* queries. Also, our approach enables us to derive theoretical performance guarantees for some queries. Our main focus is on search engines that hold all their data structures in main memory, but we give evidence that the results can be applied to classical disk-based systems as well.

Our phrase selection scheme is confined to a subset of the two-term phrases contained in a document collection to be indexed. Two-term phrases are the most widely used form of text queries [66]. They are also the slowest to resolve [114]. Once they are in the index, they can be used to speed up queries of arbitrary length greater than two [15]. Furthermore, as we outline in Section 6.3.4, our two-term phrase indexing approach can easily be generalized for phrases consisting of more than two terms.

6.3.1 Constructing Phrase Indexes

Generalizing the nextword indexes of [123, 16], the idea behind our phrase indexing scheme is simple. Assuming there is an inverted index I, we define a function $C(s,t)$ estimating the real costs caused by evaluating a term pair query $s \cdot t$ through I. We require C to be determined by properties that are also available at query time. We choose a suitable threshold T_I and add all the phrases $s \cdot t$ such that $C(s,t) \geq T_I$ to the index I. Depending on how large the phrase index will be (indicated by the threshold T_I) and whether the performance of the base index is skewed to indexing or querying, we propose two different methods for adding phrases. The naive way is to make an extra pass over the data, and to rerun the indexing process extracting (two-term) phrases rather than single terms. We call this *re-process phrase indexing*. However, if preprocessing is very expensive, the threshold T_I large enough, and the query performance not too bad, it might pay off to use *search-based phrase indexing*. That is, we firstly make a list of all two-term phrases that occur during the standard indexing process. Then, we search for all of the phrases whose cost is above the threshold, and add the results to the index.

We can exploit our phrase selection scheme to speed up the handling of some phrase queries with empty result sets. We estimate the cost of each two-term phrase occurring in the query and check whether it is above the threshold for the index. If it is, and if there are results for the phrase, the phrase is in the index and we return the results. Otherwise, we know the result set is empty and we are done.

6.3.2 Phrase Selection and Cost Models

There are various options to define suitable cost functions for a two-term phrase $s \cdot t$. In the following, let $\|s\|$ be the length of the inverted list, i.e. the occurrence frequency, of some term s. For simplicity, we use the number of documents rather than positions.

Nextword Indexes. In its original design, the partial nextword index of Bahle et al. [16] is a three-level data structure where all distinct two-term phrases of the indexed text are organized as *firstword-nextword* pairs. Each firstword is associated with a set of nextwords, and each of the nextwords has its own inverted list. Firstwords are selected according to their occurrence frequency. That is, a partial nextword index is just a different representation of a two-term phrase index that implicitly uses the cost function

$$C_0(s,t) := \|s\| .$$

Minimum. It is not hard to see that the cost of a phrase query $s \cdot t$ is not suitably rated by the frequency of the term s, in particular because an intersection is a commutative set operation. In Section 3.4 we have seen that on main-memory systems the running time of an inverted list intersection is essentially linear in the size of the smaller list, when a form of *per-list* index is used. Therefore, the minimum of the inverted list lengths may be more closely related to the real cost of an intersection:

$$C_<(s,t) := \min\{\|s\| , \|t\|\}.$$

Sum. On external memory systems, we would rather take the sum of the lengths of the inverted lists as this reflects the amount of data to be retrieved from an external storage for intersecting them:

$$C_+(s,t) := \|s\| + \|t\| .$$

Further Criteria. Of course, there are other — more or less complex — cost functions that might be used. For instances, the phrase query is often implemented as a (fast) AND query followed by a check for consecutive term positions. In this case, a cost function can be defined as the number of entries in the result set of the AND query. Queries that do not profit from the phrase index can be still evaluated in nearly the same time.

Performance Guarantee

All of these selection methods have their practical application, and all of them yield a kind of performance robustness. However, the sum-based cost function is of theoretical importance because it matches the formal worst case of intersecting two inverted lists.

Proposition 1 (Performance guarantee)**.** *Let I be a two-term phrase index and $C(s,t) = \|s\| + \|t\| \geq T_I$ its selection criterion. Then, the cost of a two-term query $s \cdot t$ on I is $O(T_I)$.*

6.3.3 Query Plans

Bahle et al. [15, 16] have studied different *query plan* generators that use simple algorithms for choosing suitable combinations of two-term phrases. Also observing that it is useful to sort the components of the query plan, they conclude that phrase queries can be evaluated simply but efficiently, as follows. Let $t_1 \cdot t_2 \cdots t_n$ be a phrase query consisting of n terms and $l_1, l_2, ..., l_n$ their inverted lists. Starting at l_1, we replace each list l_i with the inverted list of the phrase $t_i \cdot t_{i+1}$ where possible, because $\|t_i \cdot t_{i+1}\| \leq \|t_i\|$. Then, we sort the resulting set of inverted lists by increasing length. Taking the smallest list as the initial result set, the others are intersected (taking phrase offsets into account) with it successively. We skip inverted lists of two-term phrases that are already covered by previous lists during the intersections.

6.3.4 Generalization

We can easily generalize our approach to phrases consisting of more than two terms. The two-term phrase index replaces virtually each frequent term pair by a new single word (consisting of the pair). Therefore, we can repeat our two-term phrase indexing process incrementally on an already existing two-term phrase index. Physically, this adds triples and quadruplets of terms to the index.

The query plan calculation on generalized phrase indexes is more complex than on simple two-term phrase indexes. But it can also be extended in an incremental manner. However, for an l-level generalized phrase index this adds a factor of $l - 1$ to the running time of the query plan calculation.

6.4 Indexes for Small Documents

The result of a phrase query is always a subset of the AND query assembled out of the terms within the phrase. Therefore, a phrase query can be implemented in two steps. In step 1, preselect possible documents by executing the AND query. In step 2, select the results from them by using the string matching algorithm of [68] on the result documents term ID lists (which we introduced as an alternative for a positional index in Section 4.1.2). Because this implementation is generally slower than evaluating a phrase query using a positional index, we add a phrase index that makes the preselection process more precise. This gives a better tradeoff between compression and query performance (see Section 8.5). We call this modified index design a *short-text index*.

6.5 Phrase Queries on Suffix Arrays

Phrase querying on suffix arrays is straightforward. Here, a phrase search is equal to a substring search of the normalized query phrase. Due to the two separator characters between the documents, it is impossible to get a result phrase that overlaps document bounds (see Section 4.2). As the result, the suffix array returns (unsorted) positions where the phrase occurrences start. They have to be remapped to document IDs. We use a Lookup list for this.

CHAPTER 7

Document Reporting

The query result of an inverted index is a list of document IDs which are associated with pointers to the original documents. Traditionally, the documents are archived in a separate storage location, i.e. in files on disk or on the network. They are retrieved from there when the search engine returns the results. However, since we want to exploit associated advantages of main memory, our space is scarce. So instead of storing the documents separately, we use the information about the indexed documents we have already stored. In fact, from the inverted index we know in which document a term appears, and from the positional index we know where it appears within a document. The main ideas of this chapter are also published in [116].

7.1 Reconstructing Term Sequences

According to both ways of storing the positional information in the index (see Section 4.1.2), we give two variants of reconstructing the term sequences.

7.1.1 Bag of Words

Using a positional index, we could restore the sequence of (normalized) terms of a document by traversing the inverted lists, gathering the term IDs and placing their dictionary items at the correct positions. However, reconstructing a term sequence this way would take too much time. Instead, we store a *bag of words* for each document in addition to our existing data structures. A bag of words is a set of all the term IDs occurring in a document (without duplicates). We store them sorted in increasing order of IDs using Δ-Golomb encoding. As sequences of consecutive value IDs appear often in the bags, we encode them as a 0 followed by the length of that series. Using bags of words, we can build term sequences without traversing through all inverted lists. We just have to position all terms of a bag.

7.1.2 Term Lists

The second approach of Section 4.1.2 was to store all the positional information as sequences of term IDs as they occur in the documents. In this case, the reconstruction of a documents' term sequence is very simple: unpack the list of term IDs and replace each ID by the term from the dictionary.

7.2 Document Reporting on Suffix Arrays

All compressed or succinct suffix arrays of [54] (and the word-based suffix array of [27]) have the functionality of extracting snippets of any length starting at an arbitrary position in the text. Therefore, it is easy to implement a document reporting algorithm on suffix arrays: for returning

a document with the ID d we just extract the text starting at the beginning position of d and ending at the beginning position of $d + 1$ (if it exists, or at the end of the complete text). The Lookup list that maps absolute suffix array positions to document IDs stores all these starting positions. However, it cannot efficiently return the values in this reverse direction. Therefore we store an additional vector that stores for each document ID, its starting position in the suffix array text.

7.3 Text Delta

Because of the normalization, there are differences between the sequence of normalized terms and the original document we have indexed. For that reason, we store all changes made to text during the normalization process. We refer in this section to inverted indexes. However, with minor modifications, our algorithms can also be applied to suffix arrays. For word-based suffix arrays, [27] gives a further approach.

7.3.1 Term Escapes

In our dictionary, all terms are normalized to lower-case characters — the most frequent spelling of words in English texts. Any differences to this notation in the original document are recorded in a list of *term escapes*. An item in that list consists of a pair of a position where the variation in the sequence occurs and an escape value. The escape value indicates how the term has to be modified. A value of 0 means that the first character has to be capitalized and a value of 1 means that the whole term has to be spelled with capital letters. Each greater escape value points to a replacement term in a separate *escape dictionary*.

7.3.2 Term Separators

In the original text of the indexed documents, there is at least one *term separator* between each consecutive pair of terms (for example a space or a

comma). We store the arrangement of separators and terms of each document as a (Golomb-coded) list of values: a 0-value followed by x indicates that there are x terms separated by spaces. A 1-value indicates a term. Recall that the normalized text already tells us *which* term. A value of 2 or larger encodes the ID of a separator (stored in a dictionary). We use a simple binary merge algorithm to reconstruct a document from its term sequence and its list of separators.

CHAPTER 8

Experiments

This chapter evaluates the compressed inverted index as proposed above, following the results of the theoretical section. First, we give a short overview of our experimental setting and the data we used in our experiments. Then, we investigate inverted list data structures with respect to compression and running time of different intersection algorithms. Using the results, we compare our compressed inverted index against a recent in-memory indexing technology, the compressed or succinct suffix arrays (and trees). They are often proposed as an all-in-one device for text indexing, and they are becoming (rightly) more and more popular. We use publicly available implementations from the Pizza&Chili website [54]. As the alphabet size of those (character-based) implementations is bounded by a universe of eight bit, we also explore the word-based suffix array implementation of [27] that indexes word IDs rather than characters.

8.1 Experimental Setting

8.1.1 Implementation

We have implemented the data structures and algorithms as described in Chapters 3-7. The implementation was done using C++. To be able to switch easily between different combinations of data structures and algorithms, and to reduce redundant coding, we made heavy use of generic programming using C++ templates. Nevertheless, our code has more than 30 000 physical source lines of code (SLOCs).

8.1.2 Environment

The experiments were done on one core of an Intel Core 2 Duo E6600 processor clocked at 2.4 GHz with 4 GB main memory and 2×2048 KB L2 cache, running OpenSuSE 10.2 (kernel 2.6.18). The program was compiled by the gnu C++ compiler 4.1.2 using optimization level `-O3`. Timing was done using PAPI 3.5.0 developed by Mucci et al. [94]. PAPI uses hardware performance counters.

8.1.3 Test Collections

Our experiments are based on three different text collections. All collections were normalized for filling the indexes. We removed HTML tags with the `HTMLparser` of [75]. We split the text into words using the token separators of Table 8.1. All characters were folded to lowercase, and non-alphanumeric characters and symbols were replaced where possible using a subset of the translation table of [69]. Table 8.2 summarizes the properties of our test collections.

WT2g. The WT2g test collection was introduced by [62] as a 2 GB subset of the larger *VLC2* collection. The VLC2 collection consists of 100 GB of a web-crawl carried out by the Internet Archive in 1997 (see [61]). Although its WT2g subset is by now considered a 'small' input, it seems to be the

							!	”	#
0x07	0x08	0x10	0x11	0x12	0x13	0x20	0x21	0x22	0x23
$	%	&	’	(	)	*	+	,	-
0x24	0x25	0x26	0x27	0x28	0x29	0x2a	0x2b	0x2c	0x2d
.	/	:	;	<	=	>	?	@	[
0x2e	0x2f	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f	0x40	0x5b
\	]	_	‘	{	|	}	~		
0x5c	0x5d	0x5f	0x60	0x7b	0c7c	0x7d	0x7e	0xa0	0xb0

Table 8.1: *Token separators with Unicode positions*

	WT2g	WT2g.s	GOV2
documents	247 491	11 272 769	25 205 179
terms	2 008 964	1 823 710	n/a
maximum list length	212 974	5 742 167	>21 424 402
volume [MB]	2 105.32	2 105.32	436 224
volume w/o HTML [MB]	1 583.49	1 583.49	n/a
normalized volume [MB]	1 454.20	1 415.4	n/a

Table 8.2: *Properties of the text corpora used*

right size for the amount of data assigned to *one processing core* of a main memory search engine based on a cluster of low-cost machines. Using more than 1–2 GB of data on one core would mean investing much more money in RAM than in processing power.

WT2g.s. This test collection was introduced by [106]. It contains short documents and is derived from the WT2g corpus by interpreting each sentence of the plain text as a single document. The total size of the normalized text of WT2g.s is about 1.5 GB. The average document size of the test collection is 18.5 terms. However, about 2000 of its more than 10 million documents were much larger than that, so we truncated them to a size of 1024 terms each. This reduced the size of the corpus by almost 40 MB in total.

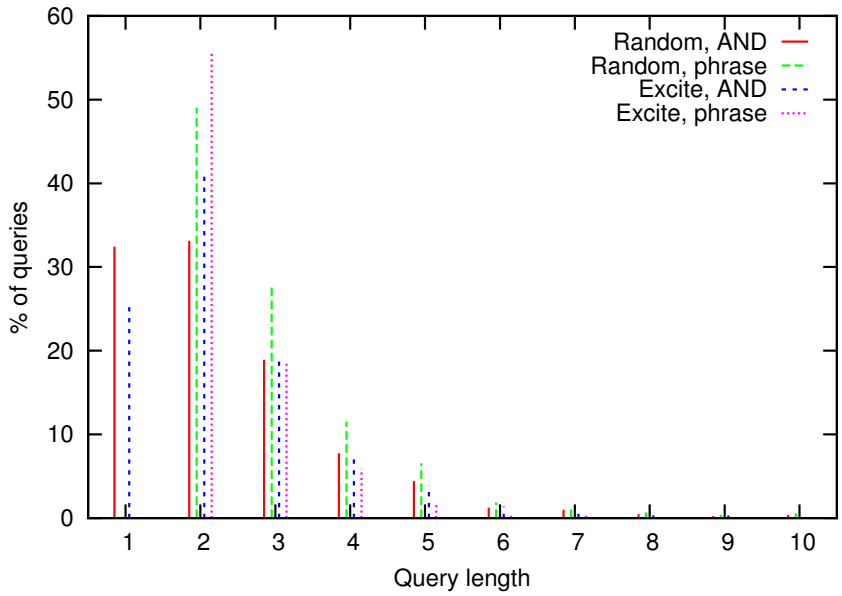
GOV2. The GOV2 test collection from TREC is a crawl of a large proportion of the websites from the `.gov` domain in early 2004 (see [42]). This

corpus contains 25 million documents filling 426 GB. We mainly use this large corpus for the external memory experiments in Section 8.5.4. In the appendix, we also give bottom-line results using GOV2 for other experiments. For processing the GOV2 data with our in-memory search engine, we split the corpus into 64 roughly equal-sized and contiguous parts. After normalization, each part had about 2 GB. This is again the amount of data normally assigned to a single processing core of an in-memory text search engine. For our main-memory measurements, we randomly selected one of the parts. It would have been much more expensive to average over all the parts, and the sizes of the result sets suggested that the chosen portion fit the query logs in the same manner as the complete corpus.

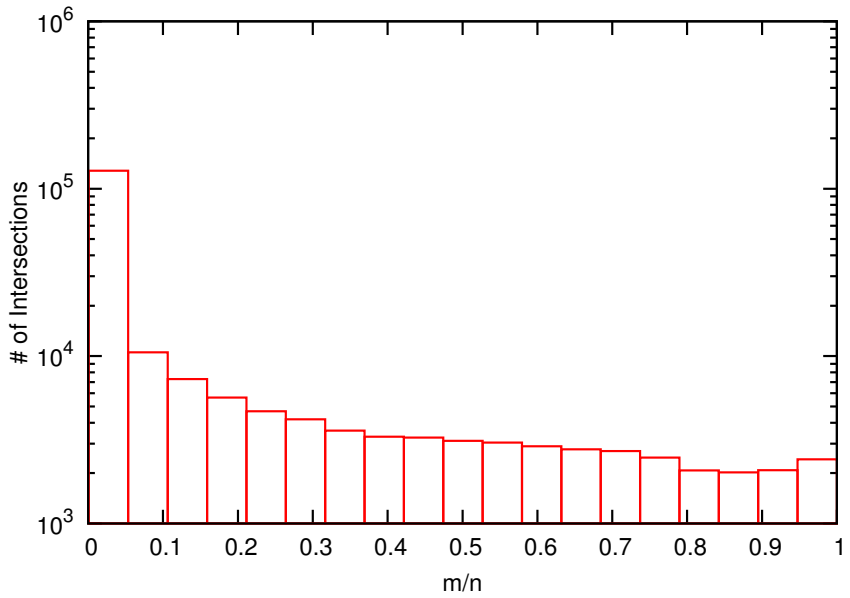
8.1.4 Query Logs

We used four different query logs for our experiments. One of them was artificially generated, and the others were real-world inputs. All logs contain AND queries as well as phrase queries consisting of between one and at least ten terms. Figure 8.1(a) shows some properties of the logs.

Random Hits. The Random query log contains pseudo real-world queries generated by selecting random hits [114]. From a randomly chosen document of the underlying corpus, we randomly selected different terms to build an AND query. Similarly, we chose a random term from such a document as the start of a phrase query. In each case, we built queries varying in the lengths according to the distribution given by [66]. They reported that over 80 % of real world queries consists of between one and three terms. Of course, the result sets of the queries contain at least one document, but in fact, they are much larger. Figure 8.1(b) shows what percentage of the intersections caused by a pair-wise evaluation of the random AND queries on WT2g fall into some range of list lengths ratios. We can see that for nearly 90 % of the intersections this ratio is below 0.1. We therefore turn our attention in particular to those ratios in our experiments.

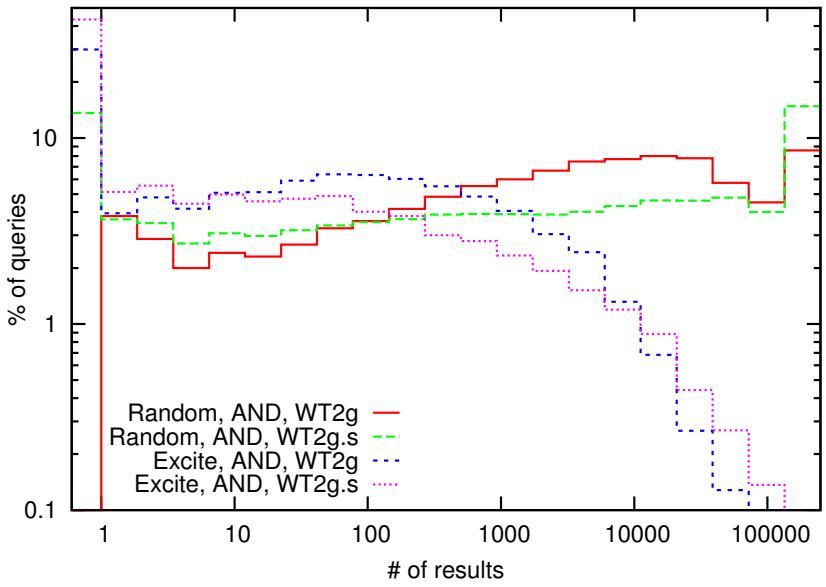


(a) Query lengths distribution of the Excite and WT2g Random log

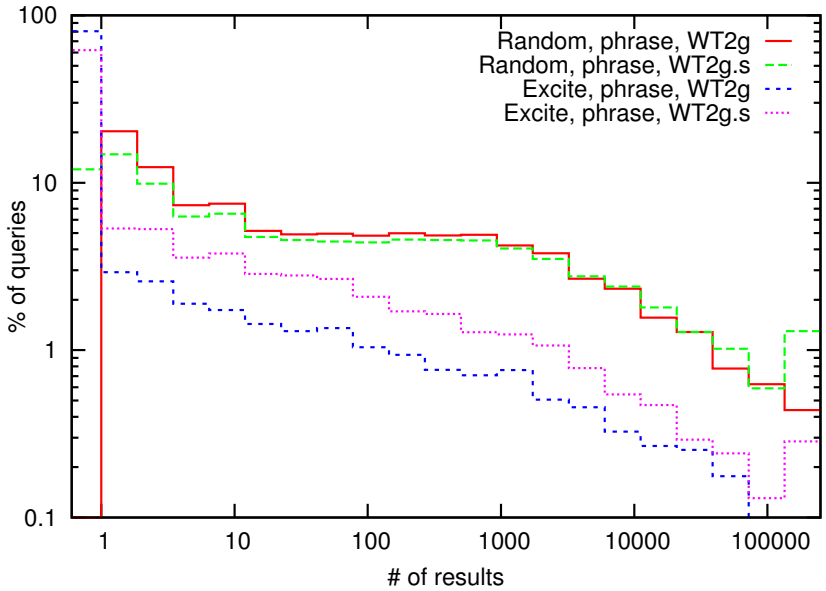


(b) Histogram of pair-wise intersections caused by the AND queries of the Random log on WT2g

Figure 8.1: *Properties of our query logs*



(a) Result set sizes of AND queries



(b) Result set sizes of phrase queries

Figure 8.2: *Result sizes on WT2g and WT2g.s*

Excite. Our real-world query log was taken from [112]. We extracted queries that were explicitly indicated as phrases by means of quotation marks, and considered all the others as AND queries. In fact, the Excite query log can be processed much more quickly than the Random log, as it contains many terms that occur either rarely or not at all in WT2g. This is because the queries were originally addressed to the whole web and not just to the sites included in WT2g. Figures 8.2(a) and 8.2(b) confirm this. The fraction of queries that do not return any results is very high for the Excite log. In contrast, the Random log has also a high fraction of queries that produce many hits. Therefore, the Excite log can be seen as an 'easy' counterweight to the Random log. We expect that many interesting applications fall somewhere between these two extremes.

TREC Efficiency Task 2005 and 2006. Finally, the last two logs consist of the queries from the TREC Efficiency task topics from the years 2005 (see [40]) and 2006 (see [30]). We call them 05eff and 06eff respectively. They were extracted from a real-world query log to compare the performance of different search engine systems on the GOV2 test collection. In fact, the 2005 query log does not match GOV2 very well, so it can be processed much more quickly than more realistic queries because it contains many terms that occur either rarely or not at all in GOV2 (or in the documents of the `.gov` domain). For this reason, the 2006 queries were selected more carefully to fit the data [41, 31]. The queries were not especially indicated as AND or phrase queries. Although some of them were obviously phrases, some were not. For our experiments, we used all of them for testing AND queries as well as for testing phrase queries. As the TREC logs were generated for GOV2, we use the logs exclusively with that corpus.

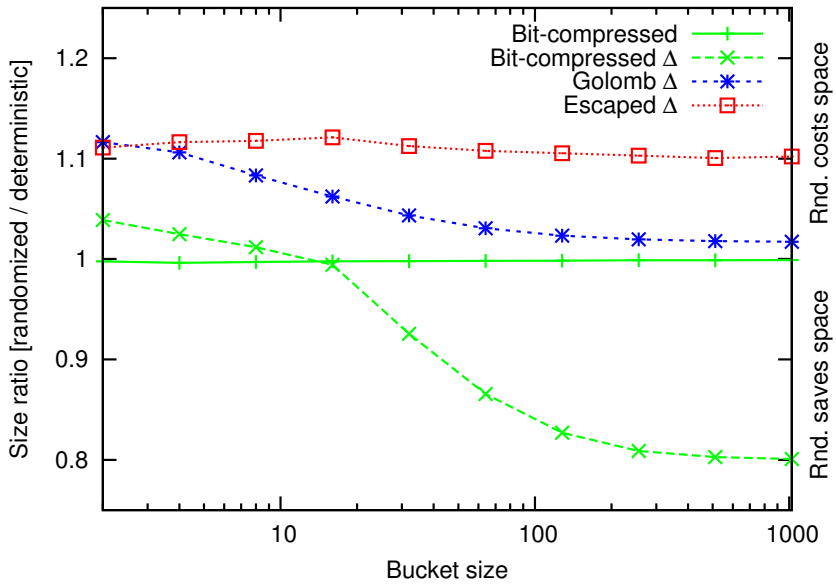
The partial phrase index of Section 6.2 requires a training log. Therefore, we adopt the procedure of [124] and split each query log into two halves. We used the first half for training and the second half for measuring.

8.2 Inverted List Compression and Intersections

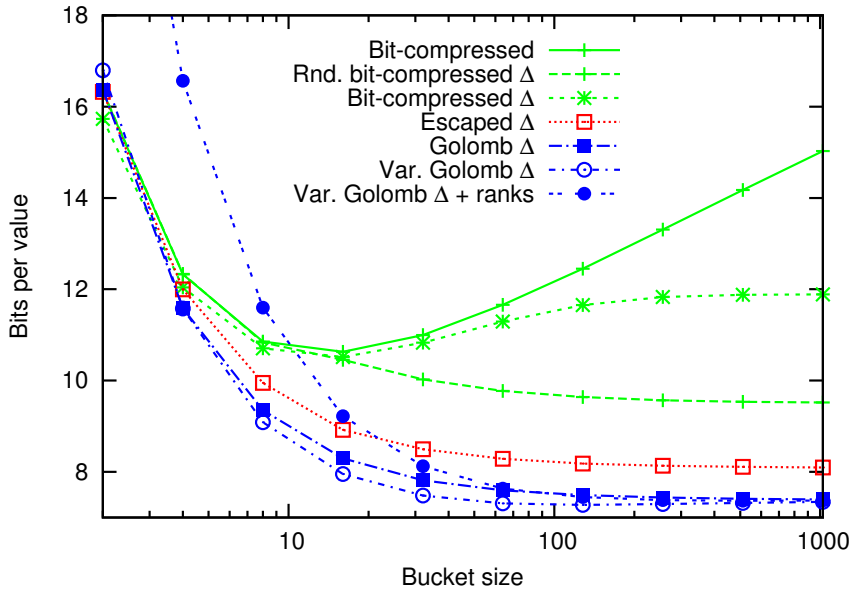
8.2.1 Space Consumption of Inverted Lists

To compare the space consumption of Lookup lists (of Section 3.4.1) and Skipper lists (of Section 3.2.2), we built inverted indexes of the WT2g collection preprocessed as described in Section 8.1.3. We varied between different compression schemes for the bottom levels of both data structures. Because algorithms Lookup and Baeza-Yates need random access to their top levels, we encoded them using bit-compression. Throughout our experiments, we denote the case in which document IDs were assigned in the order of indexing as *deterministic* (or *det.* for short), and we denote the cases where we used pseudo-randomized data meeting the requirements of the theoretical sections in Chapter 3 as *randomized* (or *rnd.* for short).

Figure 8.3(a) shows the impact of randomization on the space consumption of the inverted index. It displays the compression that can be achieved by randomization against the bucket size for different encodings. A compression ratio of one means that the randomized inverted index occupies the same amount of space as the deterministic version. A ratio smaller than one implies that randomization yields a more compact representation. All values larger than one show the cases in which randomization actually costs space. We can see in the figure that randomization does not have an influence on bit-compression. But when using bit-compression on Δ-values in combination with large bucket sizes, randomization can achieve respectable compression ratios. In this case, it eliminates a disadvantage of the deterministic data: in larger bucket ranges it is more likely that there is at least one larger Δ-value amongst the deterministic IDs that forces all others to be encoded using a larger word width. However, Figure 8.3(b) shows the average size per encoded document ID against the bucket size, and relativizes the advantage of randomization for Δ-bit-compression. For larger bucket sizes all variants of bit-compression become quite unattractive, as Golomb codes and Escaping perform significantly better. Furthermore, getting back to Figure 8.3(a), we see that these schemes suffer from random-



(a) Impact of randomization on space consumption



(b) Space consumption of different representations

Figure 8.3: *Space consumption of different Lookup lists on WT2g*

ization while they can actually profit from a deterministic non-uniformity. We can conclude that randomization is not good for the space consumption of Lookup lists.

Figure 8.3(b) compares different encodings using deterministic data. Coinciding with our analysis of the Lookup list, bit-compression does not need Δ-encoding for the very small bucket size of two. In this case, both bit-compressed encodings are the best choice and are preferable to the other compression schemes. But with increasing bucket sizes, bit-compression requires more and more memory, as more of the most significant bits of the list entries have to be stored explicitly in the bottom level. Using Δ-encoding this effect can be weakened. But beginning at a bucket size of four, using Escaping or Golomb Δ-encoding is more compact.

Lookup lists can profit even more from non-uniformity when using a variable and bucket-wise Golomb encoding as described in Section 3.4.1. But as also described, we need to store additional rank information for estimating the Golomb parameters of each bucket. As shown in the figure, this can pay off only if the rank information is required anyway, i.e. if we also use a positional index (see Sections 4.1.2 and 3.4.1). It is not a good idea to store additional ranks just to enable usage of variable Golomb coding.

Figure 8.4 compares Lookup with Skipper lists using the most promising encoding schemes. It shows again the average size per encoded document ID against the bucket size of both two-level data structures. We immediately see evidence for a higher potential of the Lookup data structure in compressing inverted lists. A reason for this is that the most significant bits of the values encoded in the bottom level do not need to be stored explicitly, as they are given implicitly by the index of the top-level. So there is no redundant information in the top level that is also contained in the bottom. However, each bucket has a certain overhead in space for both two-level representations, because each bucket has a pointer from the top to the bottom level or stores the bucket offset. So, very small bucket sizes result in considerably higher memory requirements for Lookup and Skipper lists. Practical values for Lookup lists seem to be bucket sizes greater

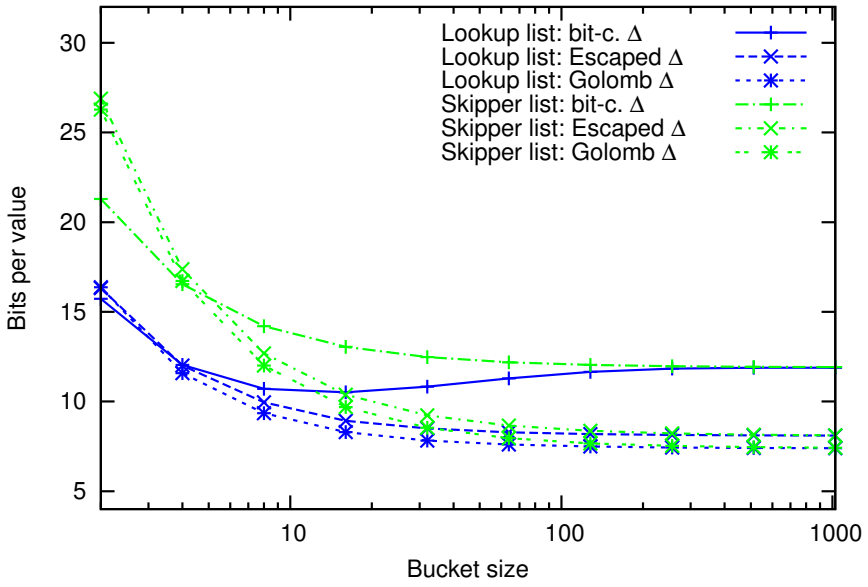


Figure 8.4: *Space consumption of different representations on WT2g*

than or equal to eight. For Skipper lists, bucket sizes of 16 or greater seem to provide acceptable space consumptions.

As to be expected, Skipper lists do not work well with bit-compressed encodings. The reasons were already mentioned above: all values in the bottom level require explicit storage of all of their most significant bits. Nevertheless bit-compressed Δ-encoding is again best for a small bucket size of two. Escaping and Golomb coding can again exploit the non-uniformity of the input for larger sizes, and can achieve a clearly smaller space consumption compared to a bit-compression of Δ-values. The overhead in space of the top level of a variable-width coded Skipper list is higher as it needs to store the first entry of each bucket in addition to the pointer of its successor. This disadvantage would be smaller in a representation optimized for algorithm Skipper, because it could actually work with a top level using Δ-encoding.

We conclude again that compact representations of Lookup lists can profit from deterministic inputs, and Lookup lists can achieve higher compression ratios than Skipper lists. Skipper lists do not work well with bit-compression, and regarding space consumption, a bucket size of eight or greater is good for Lookup lists, and a bucket size of 16 or greater is good for Skipper lists.

8.2.2 Intersecting Inverted Lists

To qualify the performance of Lookup and Skipper lists for intersections, we executed the AND queries of our Random query log on the WT2g corpus, each as a series of pair-wise intersections (see Chapter 5). We recorded the running times of the first intersections of the series, as both input lists of those intersections are in the same compressed form. Lookup and Baeza-Yates are asymmetric algorithms that could have advantages (compared to Zipper and Skipper) by passing one list in an uncompressed version. Besides, we also observed that intersections between compressed and uncompressed lists had the same characteristics, but were slightly faster for both, Lookup and Skipper lists.

Our figures show the results for the most 'difficult' intersections involving long lists. We investigate the range of length ratios $[0.001, 1]$ by subdividing it into 100 intervals with the same width on a logarithmic scale. This stresses small ratios that occur very often and that are therefore the most important ones (see Section 8.1.4). Our plots use for each interval the three most difficult intersections (with different inputs), where we average the running time for intersections that had to be evaluated more than once (with the same input).

Again, we investigate the impact of randomization on algorithm Lookup and Lookup lists. Because bit-compression needs about equal space on randomized and deterministic data (see Figure 8.3(a)), it seemed to be a good choice for comparing the running time of the intersections on both document ID distributions. Again, we varied the bucket size between the power-of-two values from 2 to 1024. For each different intersection,

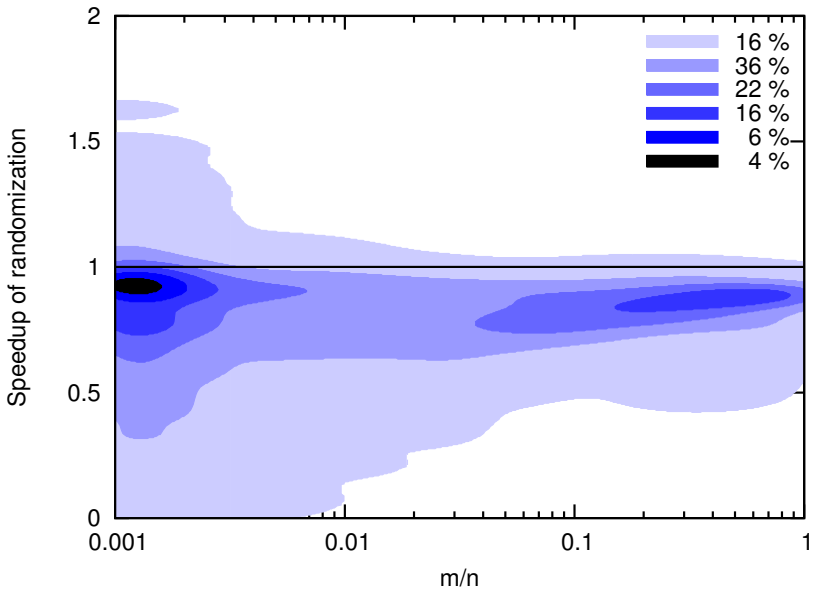
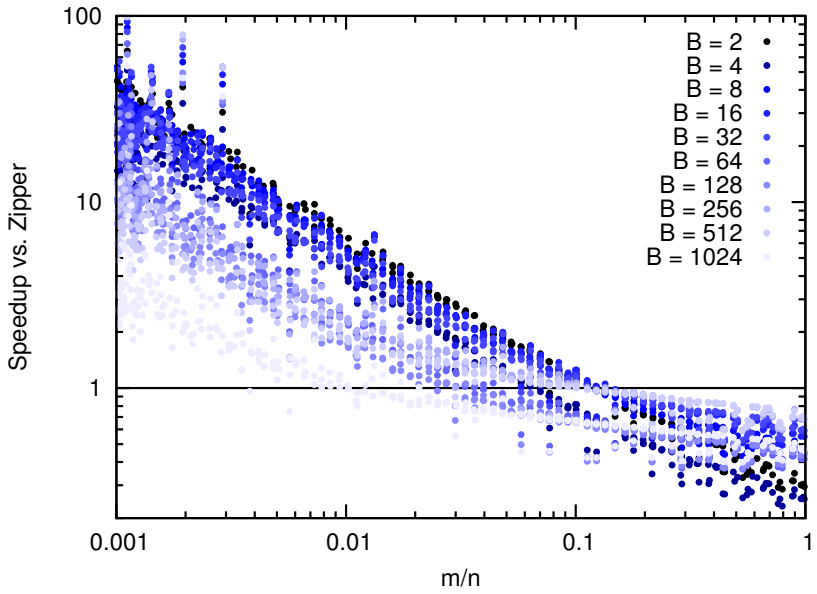
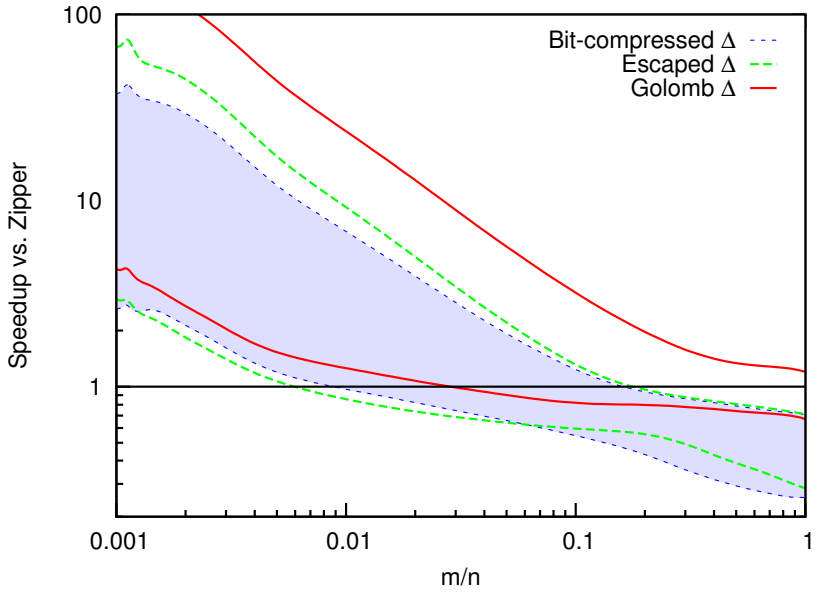


Figure 8.5: *Impact of randomization on running time*

we determined the speedup possibly achieved by randomization. That is, we divided the running time required on deterministic data by the running time that elapsed using randomized data. As our experiments did not show any influences of the bucket sizes on the speedup, we combine all results in a single plot. Figure 8.5 shows a density map of our measuring points recording the speedup against the length ratio. The density levels were estimated using the normal kernel estimator `kde2d` of the R software package [101, 119], where darker regions indicate dense areas. The legend shows what percentage of all measuring points are located in the specified levels. We can see that for roughly 90 % of the intersections (covering the complete range of ratios), randomization is actually a disadvantage. Therefore, we can conclude that randomization does not help for the performance of Lookup. Moreover, Lookup can profit from the non-uniformity of the input, and is faster on deterministic data than on randomized data.



(a) Speedup of Lookup against Zipper for different bucket sizes using Δ-bit-compression



(b) Speedup range of different encodings

Figure 8.6: *Impact of the bucket size on the running time of algorithm Lookup*

Figure 8.6(b) investigates the impact of the bucket size on algorithm Lookup. To weaken the influence of the input sizes, it shows the speedup versus algorithm Zipper (rather than the running time) against the length ratio of the input lists. Again, we used Δ-bit-compression while we varied the bucket size between 2 and 1024. We used simple Δ-bit-compressed lists (without buckets) for obtaining the reference values from Zipper. The plot shows that using Lookup lists with small buckets is better for small length ratios and worse for nearly equal lengths. However, the speedup achieved in the lower range is a magnitude higher than the slow-down that Lookup incurs in the upper range. A bucket size of between 8 and 32 seem to be a good choice for Lookup using Δ-bit-compression: those representations are very fast for small ratios and not too bad for nearly equal lengths. The results for Escaping and Golomb codes were nearly the same. Figure 8.6(b) compares the shapes of their measurement points. It displays the minimum and maximum values within our 100 measurement intervals using the gnuplot option `smooth bezier`. We can observe that using a more compact encoding, the speedup (versus algorithm Zipper executed on the same encoding) increases. The reason for this shift is that the more complex the encoding is, the more time can be saved by being able to skip values during an intersection. Before we explore more details on the impact of encodings, we briefly conclude our current results. Harmonizing with the experiments investigating the space consumption, a bucket size of 8, 16, or 32 seem to be a good choice for Lookup.

As mentioned above, we also investigated the impact of the encoding scheme on the running time of Lookup. Figure 8.7 compares Lookup lists with different encodings and bucket sizes. For the plot, we divided the running times of Lookup on compressed representations by the running time using uncompressed lists (in each case with the same bucket size). Following our previous experiments, we used bucket sizes in the range between 8 and 32 such that the index sizes did not differ too much. As expected, the higher the compression is, the slower algorithm Lookup runs. Bit-compression is the fastest but it also requires the most space. Escaping is just slightly faster than Golomb coding, but not so much as to justify a

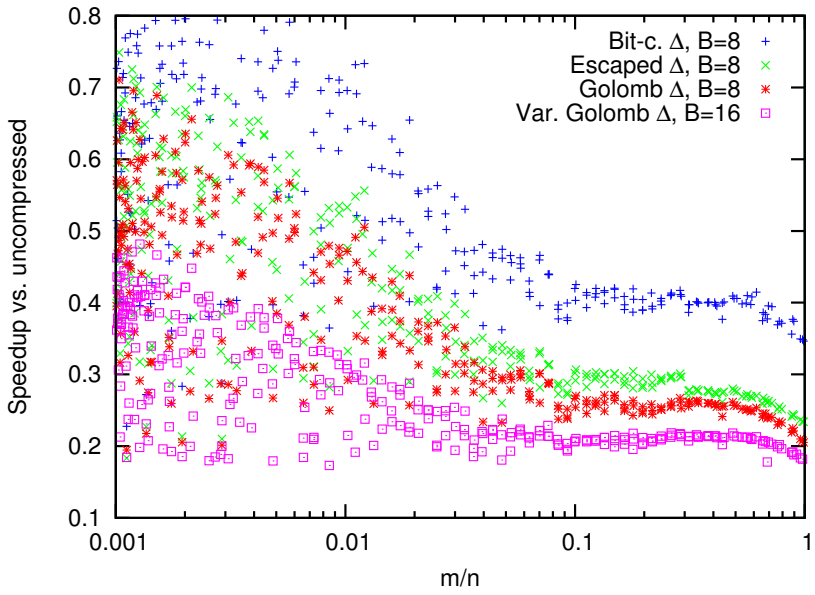


Figure 8.7: *Impact of encoding on running time*

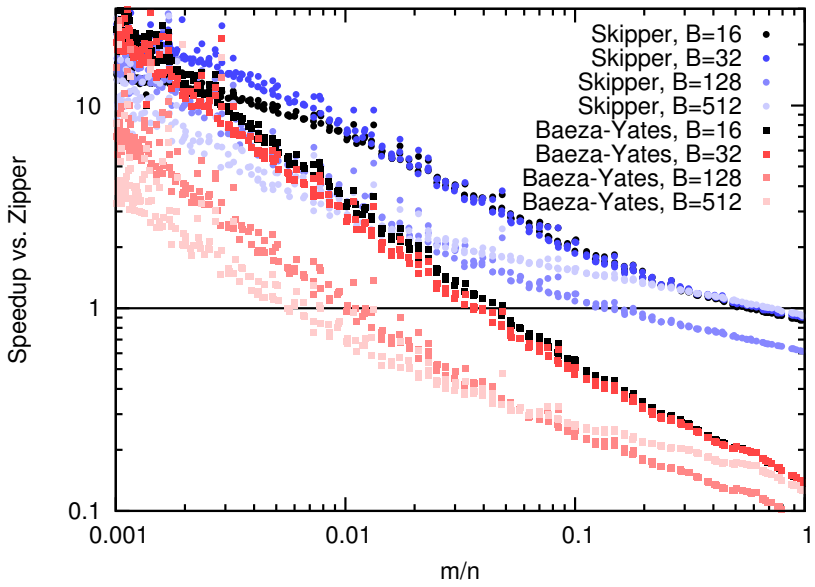


Figure 8.8: *Performance of Skipper and Baeza-Yates on Skipper lists*

higher space consumption (see Figure 8.4). Variable Golomb coding does not cost too much compared to Golomb codes with a global parameter and seems to be a good alternative when space is more restricted than time. We can conclude that algorithm Lookup is fastest using bit-compression. Golomb codes or variable Golomb codes are good for very space efficient indexes, because they yield a better space-time tradeoff than Escaping.

Analogous to our experiments with Lookup lists, we investigated the impact of the bucket size of Skipper lists on the running time of algorithm Skipper and Baeza-Yates (see Chapter 3). Figure 8.8 shows the speedup of both algorithms versus Zipper for different bucket sizes. We used Escaping (of Δ-values) for this experiment because it seemed to be roughly competitive compared to the compressed representations of Lookup lists in terms of space requirements. Indeed we also tried other encodings, but all of them yield similar results. Algorithm Skipper shows also dependencies on varying the bucket sizes. In particular, neither very large nor very small buckets give the best choice. This is because we have a delicate compromise between two evils — scanning big buckets or scanning many entries on the top level. A bucket size of 32 seems to be a good compromise. Algorithm Baeza-Yates cannot compete with Skipper for any but very small length ratios. There is again a fine balance between two evils. Larger buckets increase bucket scanning overhead but also require more expensive levels of recursion.

Figure 8.9 compares the best inverted list configurations from above using similar amounts of space. To allow a comparison on a basis independent of input lengths, encodings, and bucket sizes, the plot shows speedup values versus algorithm Zipper using simple uncompressed lists. Lookup lists (together with algorithm Lookup) clearly perform best for the very important small list length ratios. For larger ratios, Lookup lists are also well positioned and the difference to the competitors is marginal. Skipper lists using Skipper are good for very small ratios, but cannot compete with Lookup. However, they are slightly ahead of Lookup for ratios greater than about 0.1. As expected, algorithm Zipper, which does not use two-level data structures, is good for nearly equal lengths. Especially on uncom-

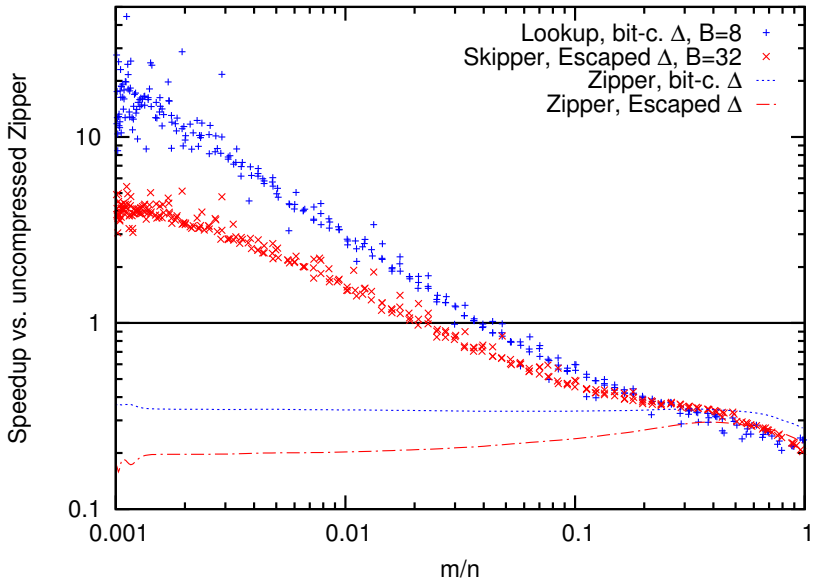


Figure 8.9: *Comparison of Lookup and Skipper*

pressed lists where the speedup of Lookup and Skipper falls below one, Zipper is better than the others. But as indicated by the (smoothed) lines in the figure, using (higher) compression, the advantage of Zipper shrinks. This suggests that it does not necessarily make sense to use algorithm Zipper on the two-level data structures for list lengths ratios greater than a certain threshold.

Concluding all our results, we can say that an index that uses bit-compressed Δ-encoded Lookup lists with a bucket size of eight is a good choice for a system that allows very fast intersections but that do not require too much space (see Figures 8.4 and 8.9). If space is even more restricted than time, then Golomb coded (or variable Golomb coded for systems with positional indexes) Lookup lists with a size of 16 are a good choice (see also Figures 8.4, 8.6(b), and 8.7).

We also made experiments for WT2g.s and a smaller index consisting of the title fields of a collection of computer science bibliographies. The results were similar. For details see [106].

8.3 Document Indexing

8.3.1 Compressed Inverted Indexes

In Section 8.2 we determined practical configurations for the large lists of our compressed inverted index (CII). It turned out that Lookup lists with a bucket size of 8 or 16 using either bit-compression or Golomb code provide a good tradeoff between memory requirement and execution times of intersections.

To address the positional index as described in the theoretical sections, we also need Lookup lists to allow fast rank operations (see Section 4.1.2). Our experiments above suggested two practical configurations for *large* inverted lists when using positional indexes. Bit-compressed Lookup lists with a bucket size of 8 yield the best running times, and Lookup lists with variable Golomb codes and a bucket size of 16 resulted in particularly good compression. We use the faster bit-compressed configuration in the following.

Consistent with the bucket size of 8, we used a threshold of $K = 128$ to switch between small and large inverted lists (see Section 4.1.1). All small lists were encoded using Δ-Golomb coding.

The first two columns of Table 8.3 summarize the properties of our two test collections and their compressed indexes. The WT2g index requires 726.7 MB. For reconstructing the normalized text, we need the bag of words, which uses 116.4 MB and gives a compression of 58 % for the normalized text. The text delta requires additional 334.3 MB, resulting in a total of 1176.1 MB for the original content of WT2g (without HTML). This still provides a compression of 74 % for the index that includes the text.

For comparison, we also give the amount of space that is required to store the input text using the compression library `zlib` [56] as an alterna-

	WT2g	WT2g.s	WT2g*
Documents	247 491	11 272 769	50 000
Terms	2 008 964	1 832 710	609 362
Single values	1 107 046	969 913	316 063
Small lists	864 679	807 208	277 093
Large lists	37 238	46 589	16 206
Dictionary	74.1	74.1	21.7
Document index	136.6	376.6	33.9
Positional index	516.0	658.2	139.6
Sum / index	726.7	1108.9	195.2
Bag of words	116.4	421.5	23.1
Text delta	(334.3)	(606.3)	(86.5)
Sum / index, text	**843.1 (1177.4)**	**1530.4 (2136.7)**	**218.3 (304.8)**
Input size	2105.3	2105.3	562.4
Input size w/o HTML	1454.2 (1583.5)	1426.9 (1533.6)	420.5 (452.3)
`zlib` w/o HTML	527.3 (620.3)	n/a	143.78 (169.8)
Compression	0.58 (0.74)	1.07 (1.39)	0.52 (0.67)
Indexing time [min]	**5.5 (6.3)**	**6.3 (7.2)**	**1.4 (1.5)**
Peak mem usage [GB]	1.6 (1.9)	2.9 (3.5)	0.4 (0.5)

Table 8.3: *CII on WT2g and WT2g.s*

tive way replacing the bags of words to report result documents. To allow document-grained access within the compressed stream (comparable to the reporting with bags of words), we added synchronization points after each document using the `Z_FULL_FLUSH` flag of `zlib`. The compressed stream of the original content occupies 620.3 MB. However, the bags of words in combination with our text delta need just 450.7 MB — roughly 27 % less. But note that we need the indexes for decompression.

As indicated by the column of WT2g.s, an average document length of 18.5 already exceeds the limit of the compression potential of the bag of words approach (the ratio is $1.07 > 1$). The shorter the document lengths are, the more values have to be stored in the bags while indexing equal contents. Furthermore, the positional index has to store many pointers to very small position lists. We show in Section 8.5.4 that in the case of short documents, it is more efficient to follow Section 6.4 and to dispense with a positional index and to extend the bags of words into sequences of term IDs as they occur in the documents.

8.3.2 Comparison with Suffix Arrays

We compare our inverted index against different suffix array (and suffix tree) implementations listed in Table 8.4. The peak memory usage while indexing the first 50 000 documents of WT2g exceeded the limits of our physical main memory for some of the suffix array implementations. So our comparison is based on this subset (called WT2g* in Table 8.3). It has a volume of roughly 452 MB. After normalization, the size shrank to about 420 MB. We indexed the normalized text of all documents as a continuous string, separating the normalized terms by spaces and the documents by a pair of spaces (see Section 4.2). Table 8.3.2 summarizes the results of the indexing processes.

The peak memory requirement during construction was measured using the Linux tool `libmemsuage.so`. The sizes given for the Pizza&Chili indexes were determined by their API function `index_size`. We used default parameters for all Pizza&Chili indexes except for RPSA and LCSA2.

Implementation	Name	References
SSA_v3.1 SSA2_v1.1	Succinct Suffix Array	Mäkinen and Navarro [83]
af-index_v2.1	Alphabet-Friendly FM-index	Ferragina et al. [53]
CCSA	Compressed Compact Suffix Array	Mäkinen and Navarro [81]
RLFM	Run-Length FM-Index	Mäkinen and Navarro [82, 83]
fmindexV2	FM-Index	Ferragina and Manzini [52]
LZ-index7 LZ-index4	LZ-Index	Navarro [95], Arroyuelo et al. [9]
RPSA LCSA2	Locally Compressed Suffix Array	González and Navarro [58]
sada_csa	Compressed Suffix Array	Sadakane [103]

Table 8.4: *Pizza&Chili implementations used*

	Construction		Result	
	Time [min]	Mem. [GB]	Size [MB]	Compr. %
CII	1.417	0.426	218.264	51.90
SSA_v3.1	24.753	4.324	412.431	98.07
SSA2_v1.1	24.644	4.324	412.431	98.07
af-index_v2.1	38.225	3.409	308.173	73.28
CCSA	32.450	3.614	542.903	129.01
RLFM	24.610	2.592	341.966	81.32
fmindexV2	8.682	2.509	215.613	51.27
LZ-index7	6.607	2.688	513.880	122.19
LZ-index4	7.284	2.616	465.768	110.75
RPSA	> 73.9	> 6.1	764.720	181.84
LCSA2	> 180	> 6.1	684.050	162.66
sada_csa$_1$	19.373	3.709	264.180	62.82
sada_csa$_2$	19.353	3.707	218.173	51.88
WCSA$_1$	3.794	1.087	223.202	53.07
WCSA$_2$	3.797	1.087	179.643	42.72

Table 8.5: *Index properties*

Their poor indexing speed forced us to activate the Ψ heuristics (see [27]). For Sadakane's compressed suffix array, we explore two different configurations. The first one indicated by sada_csa$_1$ has default parameters. The second configuration sada_csa$_2$ was adapted to match the size of the CII, and therefore, uses the sample rates $D_A = 24$ and $D_\Psi = 92$.

For the word-based compressed suffix array from [27], we also built two different configurations. One of them (WCSA$_1$) uses the parameter set $\{t_\Psi, t_A, t_{A^{-1}}\} = \{8, 8, 8\}$, and occupies roughly the same amount of space as the CII. The other (WCSA$_2$) uses $\{t_\Psi, t_A, t_{A^{-1}}\} = \{16, 8, 16\}$, and was adapted to match the size of the CII without its document-grained index. This case explores the approach of using a combination of a word-based suffix array and a document-grained inverted index (details are discussed later). We do not explore further configurations of the WCSA needing even less space, because our experiments showed that especially their worst-case query times are quite unattractive. Concerning the different parameters used in our configurations, we refer to the respective papers.

Comparing the indexing times of Table 8.3.2, it turns out that there are big differences between the different implementations. Some indexes need just a few minutes for construction while others (RPSA and LCSA2) need hours. The construction time of many character-based suffix arrays is about half an hour. With 20 minutes, the compressed suffix array (sada_csa) is slightly faster. But the fastest Pizza&Chili indexes need only about seven minutes. The WCSA is about twice as fast, and beats all character-based suffix arrays (and trees) with a time of less than four minutes. However, it cannot compete with the compressed inverted index (CII), which requires less than two minutes.

We can make clear statements about the peak memory usage during index construction. The compressed inverted index needs less than the size of the input text for construction when the text is streamed from disk or network. Even the most economical character-based Pizza&Chili index (fmindexV2) need more than six times than that. The WCSA requires more than twice the amount of the CII.

The index compression is heavily dependent on the selection of its parameters. With default values, there are two character-based Pizza&Chili indexes (fmindexV2 and sada_csa$_2$) that achieve roughly the same compression as the CII. They need about half the size of the input text. All the others occupy more space than that. Some need even more than the input size.

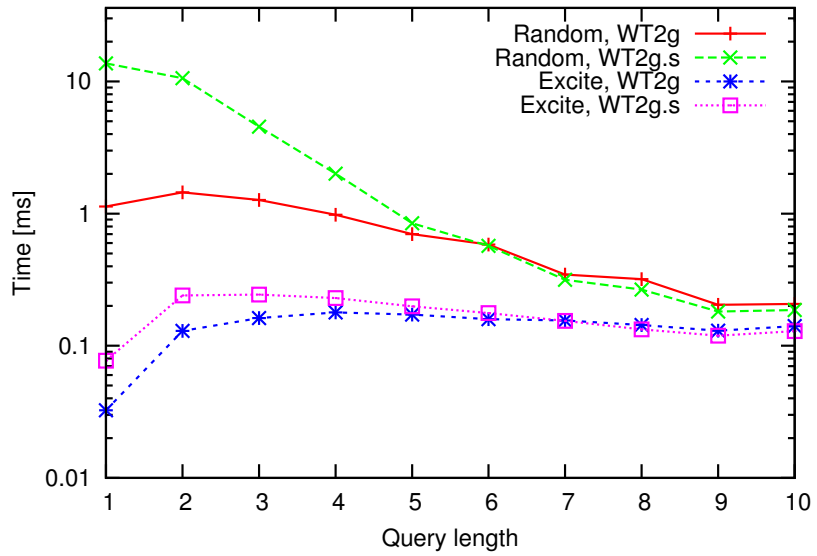
Even if other configurations are possible for each suffix array, none of the Pizza&Chili implementations seem to be promising enough such that it could provide a better tradeoff than the CII or the WCSA just by modifying its parameters.

8.4 AND Queries

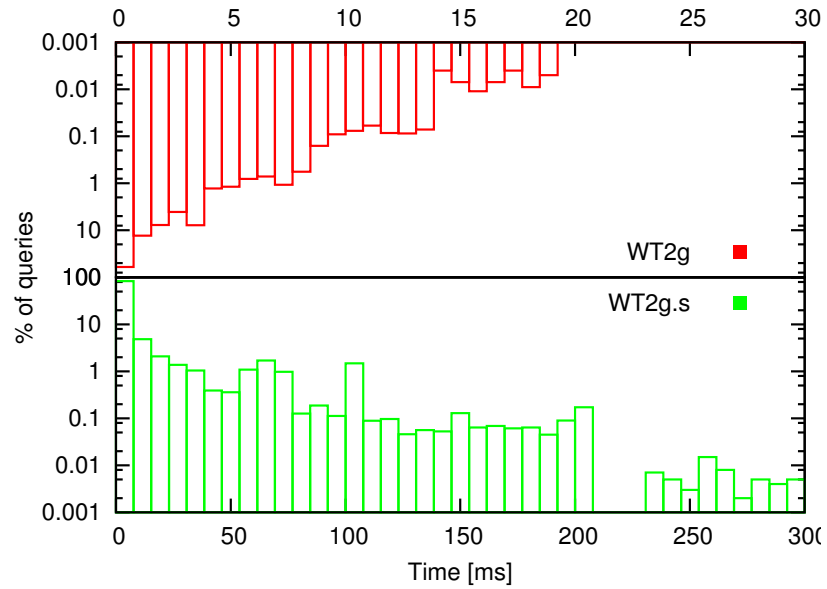
8.4.1 Compressed Inverted Index

Figure 8.10(a) shows the average query time against the query length for the AND queries of our two test logs on WT2g and WT2g.s. As expected, the Excite log can be processed much more quickly with its very small result sets than the Random log which produces many results. The reason can already be seen at a query length of one where just a single inverted list has to be unpacked: the Random log involves clearly more frequent terms. For larger queries, the CII benefits from an increasing number of query terms, as this makes it more likely that the query contains a rare term with a short inverted list. Since the running time of Lookup is essentially linear in the smaller list, this decreases processing time.

AND queries for small query lengths are slower on WT2g.s, as there are about 45 times more documents and hence more results. Because the Random log was generated for the larger documents of WT2g, it is less likely that larger queries return a hit in the shorter documents of WT2g.s. This results in smaller result sets and compensates the negative impact on the running time of the larger number of documents. So the querying times on both test collections are about equal for larger queries.



(a) Average AND querying times



(b) Histogram of AND query running times

Figure 8.10: *AND query performance of CII on WT2g and WT2g.s*

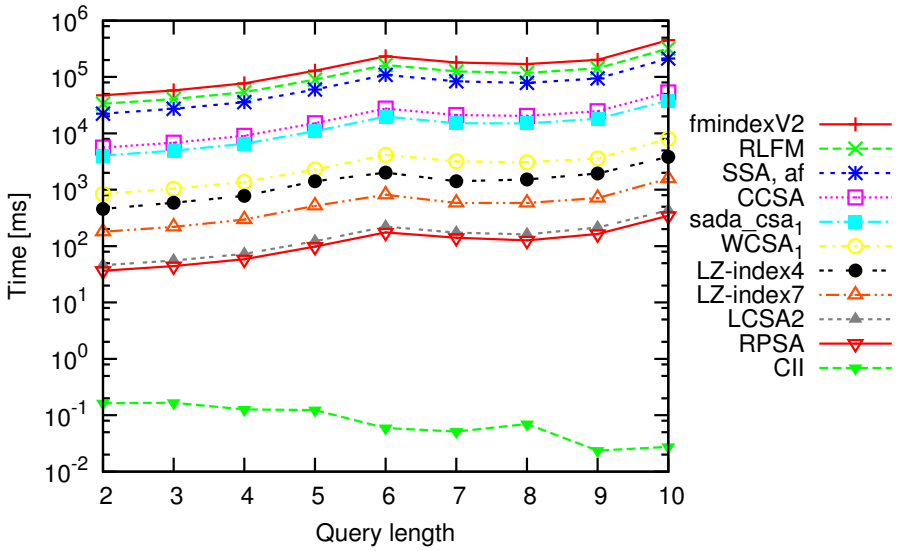


Figure 8.11: *Average AND query running times on WT2g**

Figure 8.10(b) shows the histogram of AND query execution times for the Random log on WT2g and WT2g.s. In both cases, the major part (over 80 %) of the queries is processed within the period of the first boxes. The worst case for WT2g is less than 20 ms, and for WT2g.s it is still below 300 ms.

8.4.2 Comparison with Suffix Arrays

We implemented the AND query evaluation on suffix arrays as described in Section 5.2. Figure 8.11 shows the average query time over the AND queries of our Random query log. The inverted index performs well over the entire range of query lengths. In contrast, the compressed suffix arrays produce unsorted lists of occurrences for all query terms that have to be generated using complicated algorithms, which cause many cache faults. This takes the major part of the querying time. In comparison to that, the time required for mapping the positions to document IDs and merging the lists is negligible. The bottom-line difference is huge. On the average, our

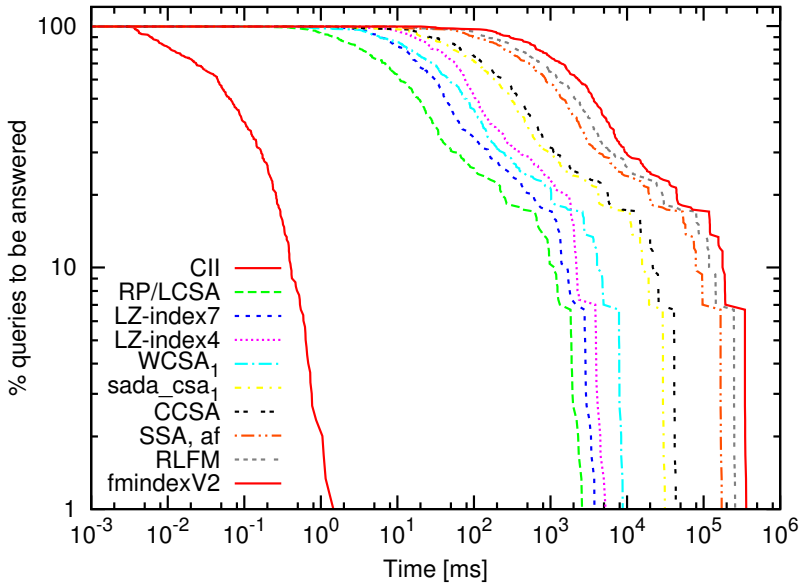


Figure 8.12: *AND queries with a length of two terms on WT2g**

data structure is more than 1 000 times faster than the fastest suffix array (RPSA) that needs more than 3.5 times the space.

In Figure 8.12 we take a closer look at the 'difficult' queries with a length of two terms. The figure shows what percentage of the queries take longer than a given time. In a double logarithmic plot, all curves have a quite clear cutoff. However, the differences are again huge. While the inverted index never took more than 1.6 ms, the fastest suffix array (RPSA) needs up to 3.9 s, almost 2 500 times longer (using more than 3.5 times the space). We can conclude that a search engine based on suffix arrays (including WCSA) would probably need an additional inverted index at least for Boolean queries.

8.5 Phrase Queries

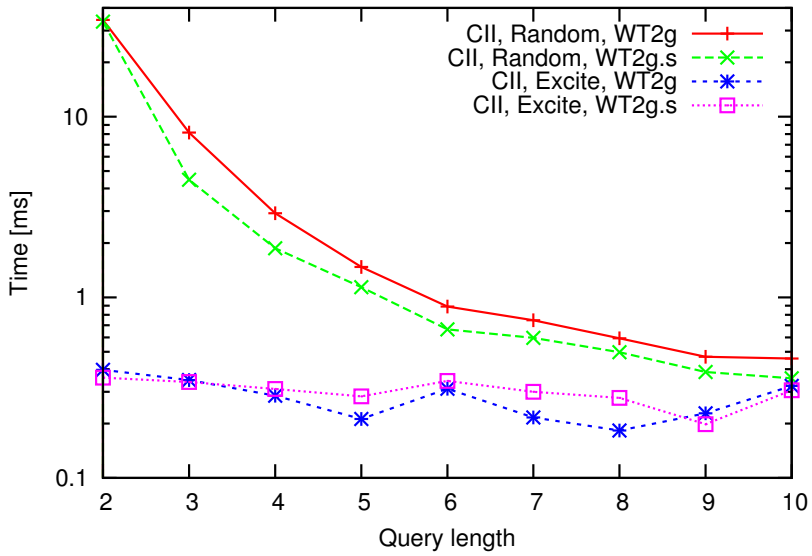
8.5.1 Positional Inverted Indexes

Figure 8.13(a) shows the average phrase querying time against the query length for the inverted index on WT2g and WT2g.s. As already observed in the experiment for AND queries, the Excite log can be processed much more quickly than the Random log, as it has less hits. Again, the CII benefits from an increasing number of query terms for larger queries. It makes it more likely that the query contains a rare term that also produces fewer results.

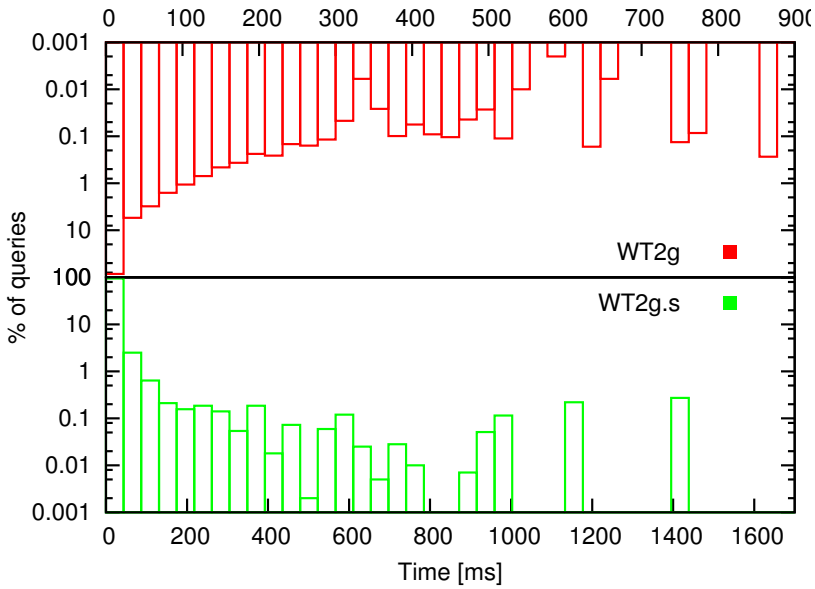
Phrase queries are faster on WT2g.s than on WT2g, because the position lists are shorter for small documents. In this case, the problem is moved from expensive merges of position lists into the highly optimized intersections of Lookup lists. However, this is bought dearly by a higher space consumption (see Section 8.2.1). But as Figure 8.13(b) shows, the worst-case querying time of the Random log is nevertheless nearly twice the number for WT2g.s. This is because the log contains very frequent (short) phrases that occur in most documents of WT2g.s (which has about 45 times more documents than WT2g).

8.5.2 A Home Match for Suffix Arrays?

We implemented the phrase query evaluation on suffix arrays as described in Section 6.5. In Figure 8.14(a) we see the average time required for the phrase queries of our Random query log. As indicated above, phrase queries are easy wins for suffix arrays. Therefore we compare the CII only with those that occupy more or less the same amount of space. For the most frequent practical case of two terms, the inverted index is nevertheless about 10 times faster than the best character-based suffix array (sada_csa$_2$). Pizza&Chili suffix arrays are sometimes slightly faster for rare and easy phrases with more terms, where almost all average running times are already below 10 ms. The WCSA has about the same (good) performance

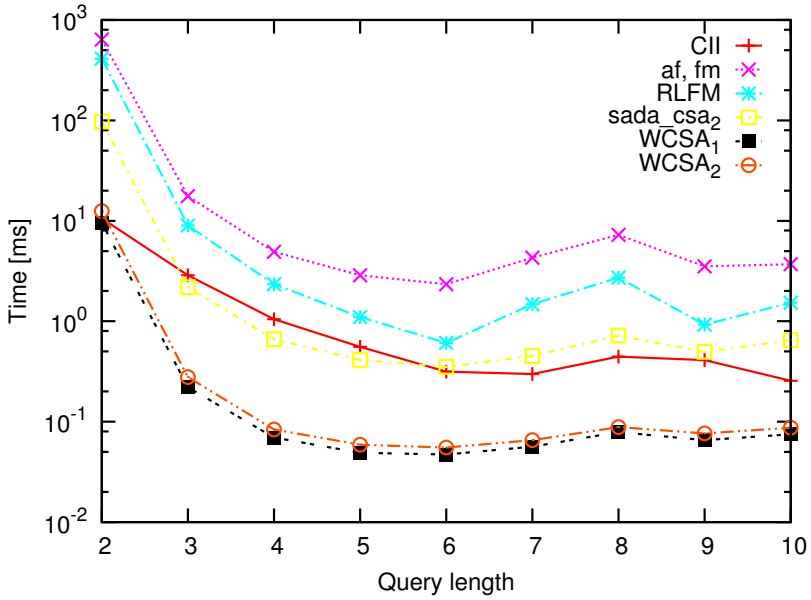


(a) Average phrase querying time of CII

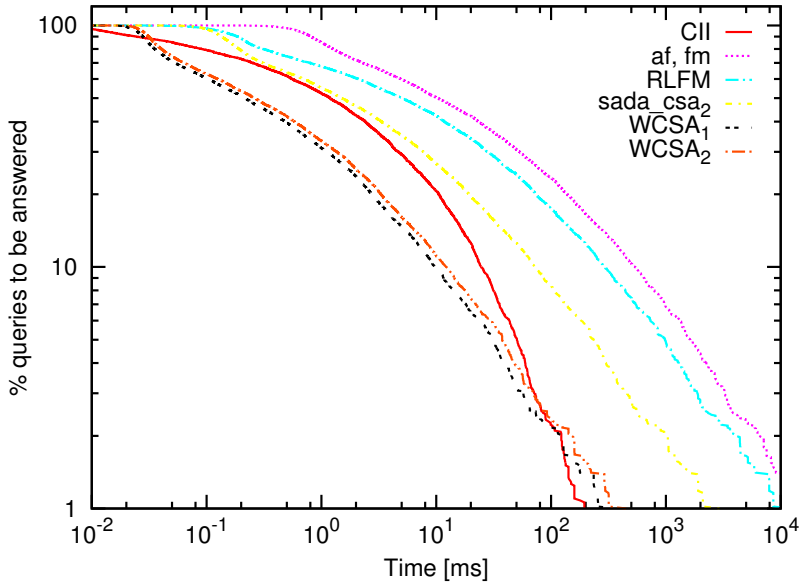


(b) Histogram of phrase query running times

Figure 8.13: *Phrase query performance of CII on WT2g and WT2g.s*

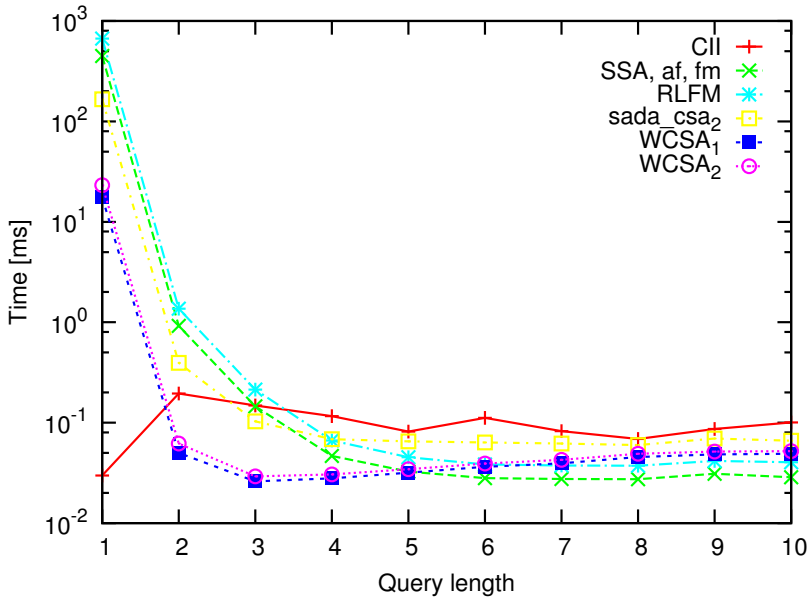


(a) Average times of the Random log

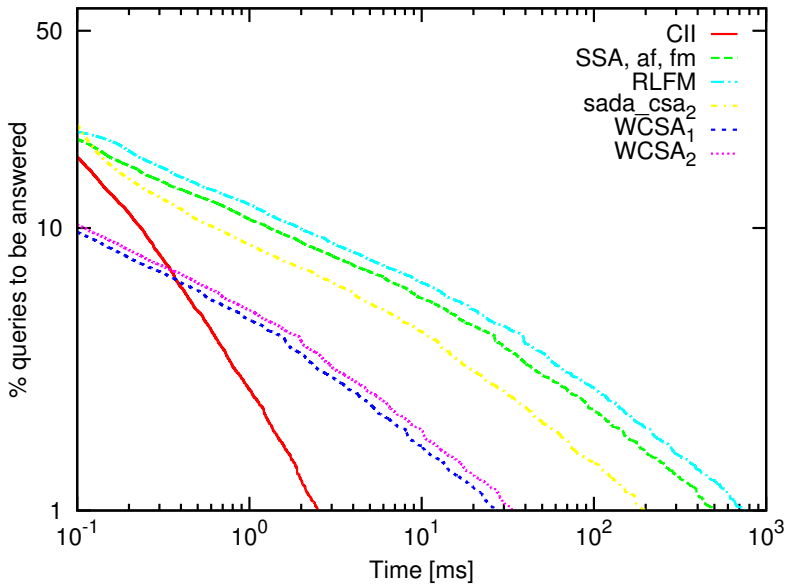


(b) Phrase queries with a length of two terms

Figure 8.14: *Random log phrase query performance on WT2g**



(a) Average times of the Excite log



(b) Excite log phrase query processing

Figure 8.15: *Excite log phrase query performance on WT2g**

as the CII in the two-term case, and is even faster than its character-based pendants for larger queries.

Again, Figure 8.14(b) shows what percentage of the difficult queries of the Random log with a length of two terms can be processed in a certain time. Even the fastest character-based suffix array (sada_csa$_2$) needs nearly 8 s for some of the phrases. This is more than a factor 30 longer than CII. The reason is that some phrases occur very frequently, and unpacking all of them can be very expensive. The WCSA, however, can compete with the good running times of CII for the difficult queries. Although its longest observed query time is a factor of 3.6 longer, the WCSA can process more of the two term queries within the range from around 0.1 ms to 75 ms.

Results for the Excite log are shown in Figures 8.15(a) and 8.15(b). The inverted index is clearly best for single-term queries. As the Excite log produces few results, all indexes are far below 1 ms for larger queries. However, worst-case times of the suffix arrays are factors longer than that of CII.

Experiments for WCSA on the complete WT2g showed similar results relative to CII. The difference between WCSA$_1$ and WCSA$_2$ is marginal. So, it seems also to be a good alternative to use WCSA$_2$ instead of the positional index and the bags of words of CII. This would also remove the major disadvantage of WCSA for single-term (phrase) queries.

8.5.3 Partial Phrase Indexes

Figure 8.16 and Table 8.6 show the results for our experiments with the simplest approach for speeding up phrase queries on inverted indexes, the partial phrase index of Section 6.2. As described in the introduction of this chapter, we split the query logs in two parts and used the first part for filling the phrase index and the second part for measuring.

We indexed the most frequently occurring queries of the training logs, varying the size of the phrase index between the values 10, 100, 1 000, 10 000, 20 000, 30 000, and 40 000. The table shows the space and time required for building the index. As to be expected, the difficult Random

Cached queries			10	100	1000	10000	20000	30000	40000
Time	[s]	Excite	0.08	0.30	1.31	8.81	16.69	22.25	26.89
		Rnd.	4.92	17.25	71.88	200.78	271.41	341.56	394.14
Size	[MB]	Excite	0.32	0.90	2.53	7.14	10.23	12.10	13.97
		Rnd.	11.29	31.43	73.06	115.05	131.16	150.12	166.05

Table 8.6: *Resources required by partial phrase indexes*

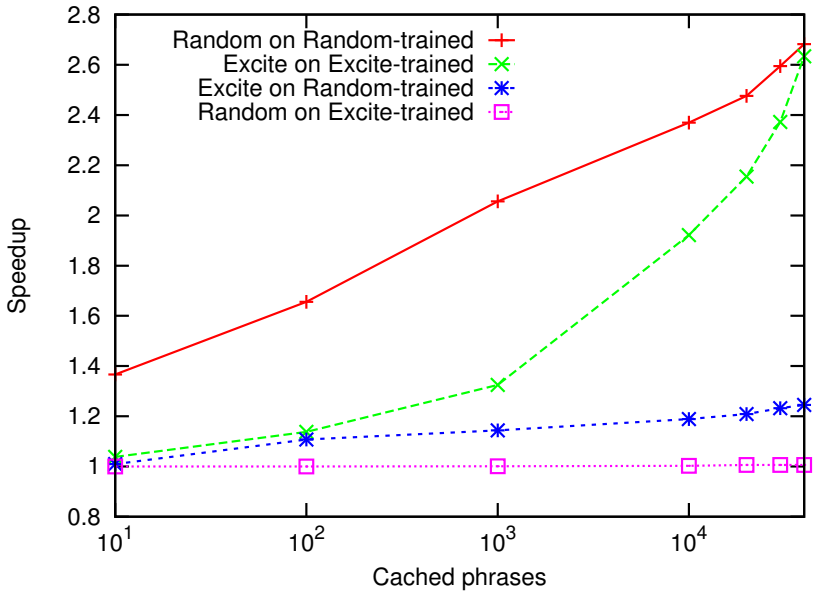


Figure 8.16: *Partial phrase indexes on WT2g*

queries are much more expensive. However, the first line in the plot shows that the high cost for building the phrase index pays off by considerable speedups of the Random log. As the second line in the plot shows, the Excite log can noticeably profit from a partial phrase index as well. Especially, an index size of 10 000 phrases seems to yield a good tradeoff between required space and obtained query performance.

This observation should be interpreted with care, since here the phrase index is being benchmarked on the Excite log for which it was optimized.

Indeed, these results are unrealistically good, as in most cases no suitable query logs are available or existing query logs are out of date. To simulate the more usual situation, we also measured both logs in each case using the other log for training. As the two lower lines of Figure 8.16 show, this transforms the outcomes. The speedup in both cases has become almost invisible. In particular, a Random-trained phrase index, which might be a tempting alternative to overcome the problem of missing query logs as training set, is not an attractive approach. Furthermore, our experiments showed that also the worst-case times cannot be significantly improved by partial phrase indexes.

8.5.4 Two-Term Phrase Indexes

Table 8.7 shows the time required to build the two-term phrase indexes of Section 6.3 using different cost models and both indexing strategies of Section 6.3.1. We varied the size of the phrase indexes by investing between 10 % and 50 % in addition to the memory required by the underlying inverted index (on WT2g). As we can see in the table, the two indexing methods have quite different construction times. The *re-process* phrase indexing is much faster than the *search-based* approach and it seems that it provides great scalability. However, the values given in the table are plain indexing times, and they do not contain the time required for preprocessing the text again, which would be necessary when using the re-process phrase indexing strategy. In fact, for our implementation, preprocessing is expensive and the search-based approach is actually faster for small phrase indexes requiring 10 % of additional space or less. Moreover, the search-based method allows us also to derive some characteristics of the different cost models. The nextword index (using the cost function C_0 from Section 6.3.2) has the longest search-based indexing times when the two-term phrase index occupies 10 % of additional space. But, investing more space, the minimum-based cost model (cost function $C_<$) requires clearly more time for construction than the others. Rather than seeing this as a disadvantage, we identify a quality feature of the minimum-based cost model

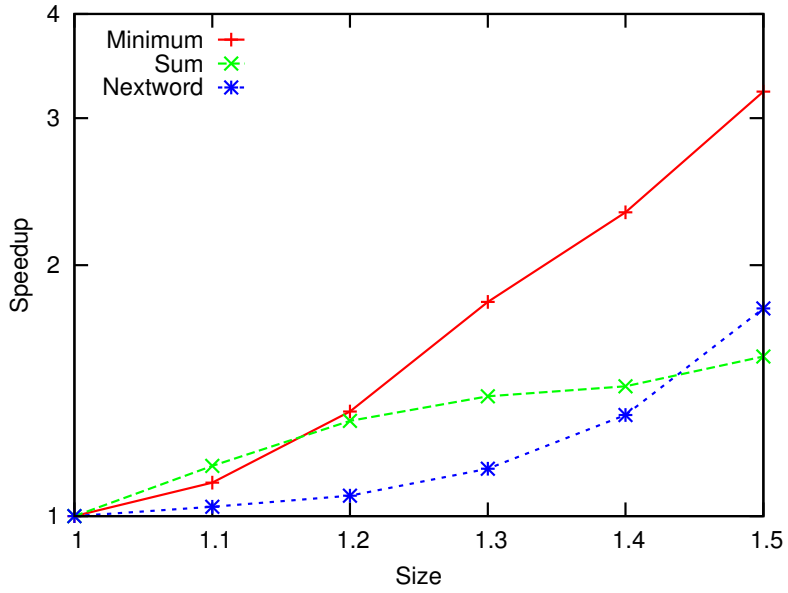
Add.	Re-process			Search-based		
size	C_+	$C_<$	C_0	C_+	$C_<$	C_0
10 %	3.12	3.16	3.29	5.92	6.53	8.46
20 %	3.96	3.99	4.04	15.79	21.92	15.58
30 %	5.14	5.10	5.00	28.68	47.74	27.62
40 %	6.24	6.43	6.11	39.94	82.51	45.12
50 %	7.67	7.97	7.39	60.83	123.98	72.70

Table 8.7: *Phrase index construction time in minutes for different amounts of space in fractions of the underlying inverted index*

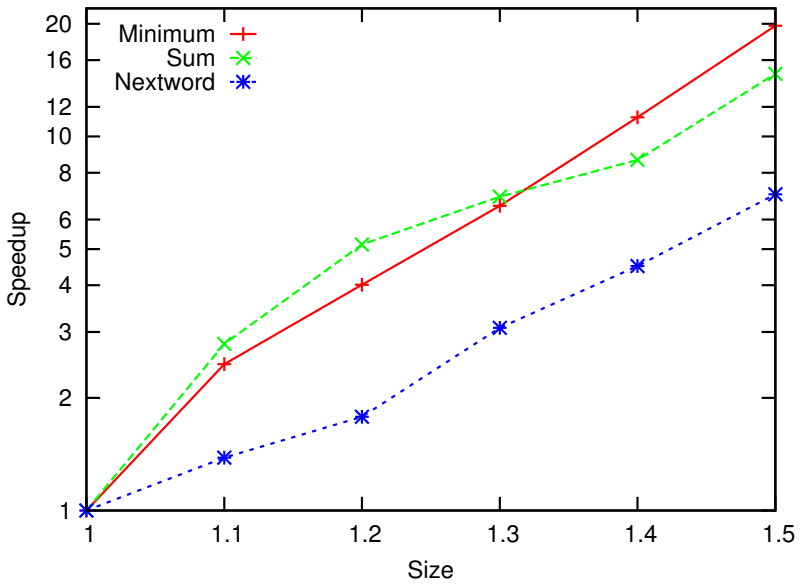
(on in-memory systems). Indeed, our goal while designing suitable cost functions was to select the most difficult phrases possible. Our idea behind this was to move as much work as possible from query time to construction time.

Figure 8.17(a) and 8.17(b) confirm the strength of $C_<$ for in-memory systems. The minimum-based cost model shows good performance on the Random log and on the Excite log. In particular, its scalability is excellent and it clearly wins over the other cost models when investing more than 30 % of additional memory. Using less memory, the performance of the minimum-based cost model is similar to that of the sum-based model. The nextword cost model, however, cannot compete with the others for either of the query logs.

Comparing the result of the query logs shows that the Random log can be considerably more effectively accelerated than the Excite log. This is because the logs have quite different characteristics. In particular, the Random log was generated by randomly selecting phrases of different lengths from the underlying WT2g corpus. Therefore, it is more likely that the log contains phrases that occur frequently (in WT2g) than phrases that occur rarely. On the other hand, our phrase index selects term pairs from the WT2g index that have — assuming a reasonable cost model — very large inverted lists. That is, the phrase index contains also some of the most frequently occurring (two-term) phrases of WT2g. Hence, we have similar circumstances as within the log. Even if this is an overoptimistic case, we



(a) Speedup of average times of the Excite log on WT2g



(b) Speedup of average times of the Random log on WT2g

Figure 8.17: *Query time speedup using different cost models*

can identify a good property of our two-term phrase index: difficult logs that match the data seem to profit more from the two-term phrase index than easy logs that are fast anyway.

Experiments on a subset of the GOV2 corpus using the Random (generated over the complete GOV2 corpus), the Excite, and the TREC query logs showed similar results. For some details see [115].

External Memory Systems

We also implemented the two-term phrase index as an extension of the disk-based search engine *Zettair* [125]. We report a query performance comparison between CII and Zettair in Appendix B. We downloaded Zettair version 0.9.3 and adapted it to get comparable results: we deactivated the ranking, as well as the expensive encoding of the result sets and the query cache. We modified Zettair to keep a second index in addition to its original one where we indexed the phrases. This seemed to be the most convenient way to implement a phrase index on top of Zettair. To avoid negative influences between the indexes, we put both indexes on different physical hard disks. As Zettair preprocesses documents very quickly, we used the re-process phrase indexing strategy that is also preferable for other external memory systems.

Throughout our experiments with Zettair, we switched off stemming and used the `--big-and-fast` option allowing the engine to consume larger amounts of memory. We used the GOV2 corpus as it has a larger size suitable for experiments with an external memory search engine. Consequently, we used also the 05 and 06 TREC query logs. We measured the query times in a batched fashion, but we cleared the file system caches using `drop_caches` whenever we changed the index configuration or the query log.

Table 8.8 shows the average query time speedup for all our query logs using different cost models. Again, we varied the size of the phrase indexes by investing between 10 % and 50 % in addition to the memory required by the underlying inverted index (on GOV2). The sum-based cost function

Add.	Random			Excite		
size	C_+	$C_<$	C_0	C_+	$C_<$	C_0
10 %	1.499	1.487	1.160	1.230	1.180	1.234
20 %	1.977	1.778	1.744	1.533	1.276	2.645
30 %	2.626	2.113	2.204	2.122	1.422	3.157
40 %	3.494	2.396	2.623	3.085	1.566	3.551
50 %	4.754	3.018	3.124	3.679	1.708	3.872
Add.	05eff			06eff		
size	C_+	$C_<$	C_0	C_+	$C_<$	C_0
10 %	1.142	1.116	1.216	1.416	1.363	1.295
20 %	1.445	1.261	3.107	1.981	1.664	3.674
30 %	2.381	1.405	3.806	2.712	1.931	4.741
40 %	3.566	1.428	4.573	4.505	2.146	5.625
50 %	4.422	1.574	5.374	5.871	2.524	6.596

Table 8.8: *Average query time speedup of the external memory phrase index on GOV2*

C_+ performs clearly best for the difficult Random query log, and therefore it fulfills our intention to speedup the most expensive queries. For the easier logs, the Nextword index (using cost function C_0) shows slightly better results. As expected, the minimum based cost model designed for in-memory systems cannot keep up with the others.

The worst-case times in Table 8.9 emphasize the strength of the sum-based cost model in terms of a certain performance robustness. In particular, for difficult queries it can significantly improve the worst-case query times. Moreover, it seems that C_+ can provide better scalability for increasing phrase index sizes than C_0 can do. The results coincide with previous estimations that we made in [115]. This suggests that even the query log based approach of [124] (which we explore for our in-memory system in Section 8.5.5) cannot compete with our two-term phrase index for difficult queries.

Add.	Random		Excite		05eff		06eff	
size	C_+	C_0	C_+	C_0	C_+	C_0	C_+	C_0
10 %	3.104	1.365	1.645	1.639	1.210	1.293	1.539	1.119
20 %	3.796	1.994	1.431	2.484	1.667	2.886	1.732	4.273
30 %	5.369	3.187	2.273	2.495	2.433	2.875	2.882	5.065
40 %	7.407	3.502	6.174	2.512	4.802	2.874	6.293	5.021
50 %	7.368	3.920	7.684	2.494	6.728	2.885	9.770	6.419

Table 8.9: *Speedup of worst-case times of the external memory phrase index on GOV2*

		Pos. index	Short-text index				
			Term lists + phrase index				Term lists
Size [KB]		1 460 683	×1.0	×0.9	×0.8	×0.7	×0.6
Avg. [ms]	05eff	0.614	0.186	0.260	0.386	0.592	0.987
	06eff	0.687	0.120	0.178	0.296	0.487	0.804
	Excite	1.193	0.298	0.475	0.765	1.179	2.200
	Random	23.985	0.943	1.249	2.097	5.062	63.118
W.-c. [ms]	05eff	134	126	126	124	126	424
	06eff	88	13	19	68	81	366
	Excite	212	18	22	46	108	697
	Random	1 918	110	110	110	178	4 277

Table 8.10: *Positional index vs. short-text index on WT2g.s*

Small Document Indexes

We also implemented the *short-text index* of Section 6.4 and used the two-term phrase index (in combination with the term lists of Section 4.1.2) as a powerful substitute for the positional index. For the following experiments, we used the WT2g.s test collection.

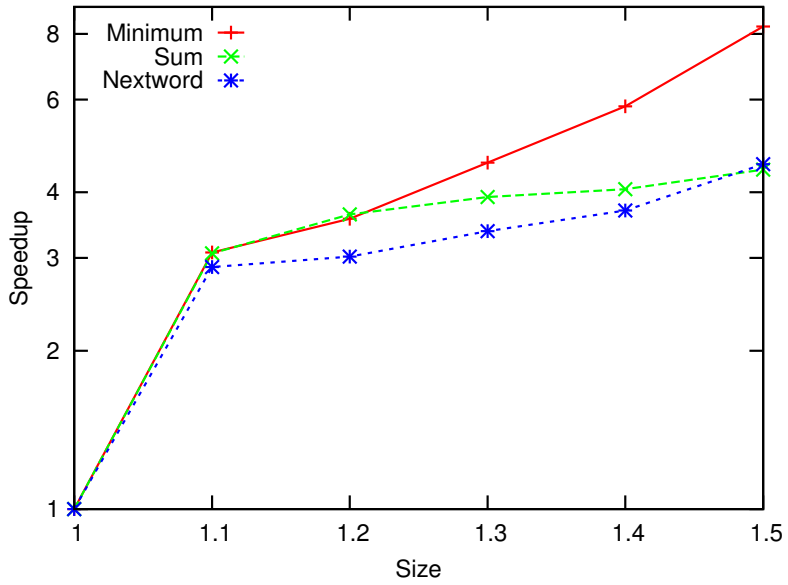
We compared the short-text index with the classical positional index. Table 8.10 shows the average query times as well as the worst-case query times for different query logs on both indexes. For the short-text index, the table shows the results of different configurations varying the available space (given as fractions of the size of the positional index). The reference value of the positional index contains the sizes of the *bags of words* of all

documents. Recall that a bag is the sorted list of the term IDs of a document (without duplicates). It is used for reconstructing the document content from the indexes (see Section 7.1), and for fast updates (see Chapter 9). Because the short-text index holds term ID sequences called *term lists* (see Sections 6.4 and 4.1.2), the bags become superfluous. So, the size values of the short-text index contain just the space needed for the term lists. The plain short-text index without any phrase needs about 60% of the size of the positional index. We added phrases and measured the query log running times at different thresholds. As Table 8.10 shows, if we use about 70% of the space of the positional index, the short-text index already performs better than the classical approach. Using the same amount of space, the short-text index is even up to 25 times faster.

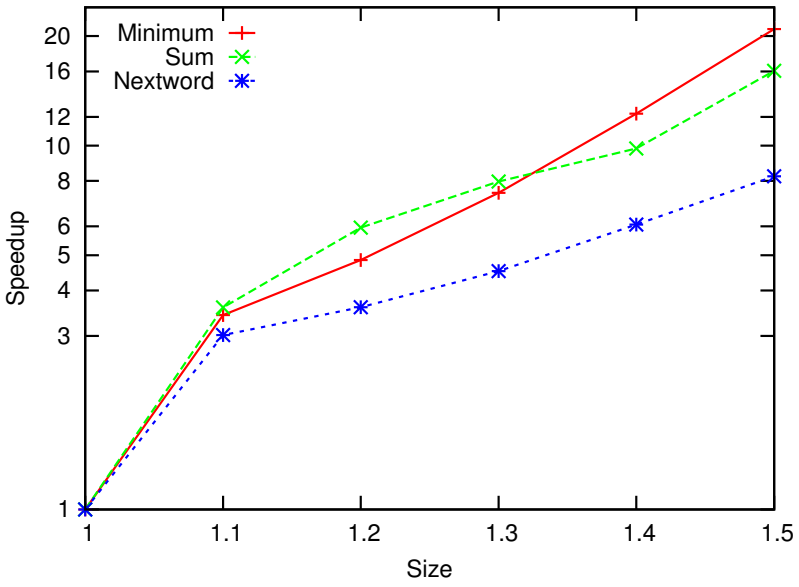
8.5.5 Combined Approaches

We also conducted experiments with combinations of partial phrase indexes and two-term phrase indexes using different cost models. For the partial phrase index, we used in each case the 10 000 most frequent queries of the training log. This number gave a good time-space tradeoff in our experiments above (coinciding with [124]). The previous experiments showed also that partial phrase indexes where the training log differs from the test log do not yield gains in query performance. Therefore, we show only the results with the optimistic configuration that uses the first half of each log for training and the second half of the same log for measuring. The combination that uses the nextword cost model corresponds exactly to the three-way index combination of [124].

Figure 8.18(a) shows again the average query time speedup of the combined approaches for the Excite log, and Figure 8.18(b) illustrates analogous results for the Random log. While for the Random log the combined approaches are not much better than a two-term phrase index alone (see Section 8.5.4), the partial phrase index and the two-term phrase index seem to complement one another for the Excite log. The minimum-based cost



(a) Speedup of average times of the Excite log on WT2g



(b) Speedup of average times of the Random log on WT2g

Figure 8.18: *Query time speedup using a combination of a two-term phrase index with different cost models and a partial phrase index with a size of 10 000 phrases*

model is again best on our main memory search engine. Its index combination can accelerate the Excite queries with a factor up to of eight.

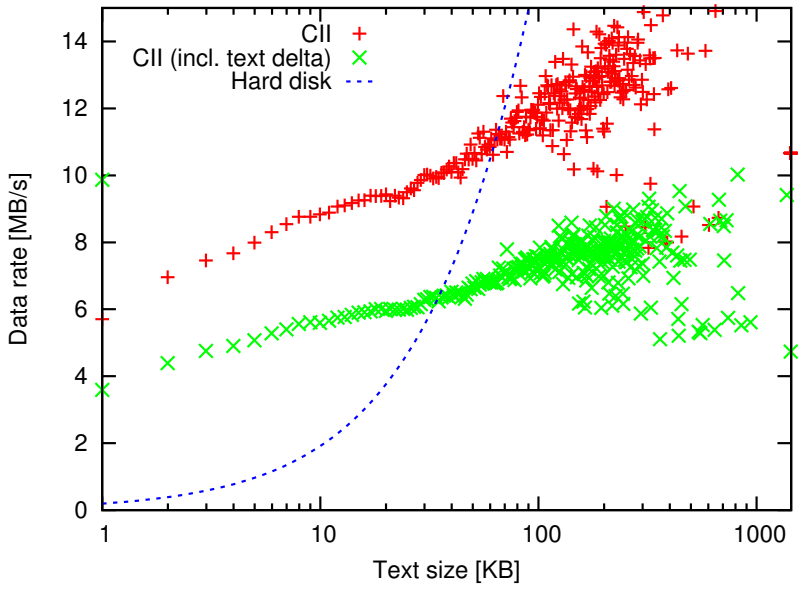
8.6 Document Reporting

8.6.1 Compressed Inverted Index

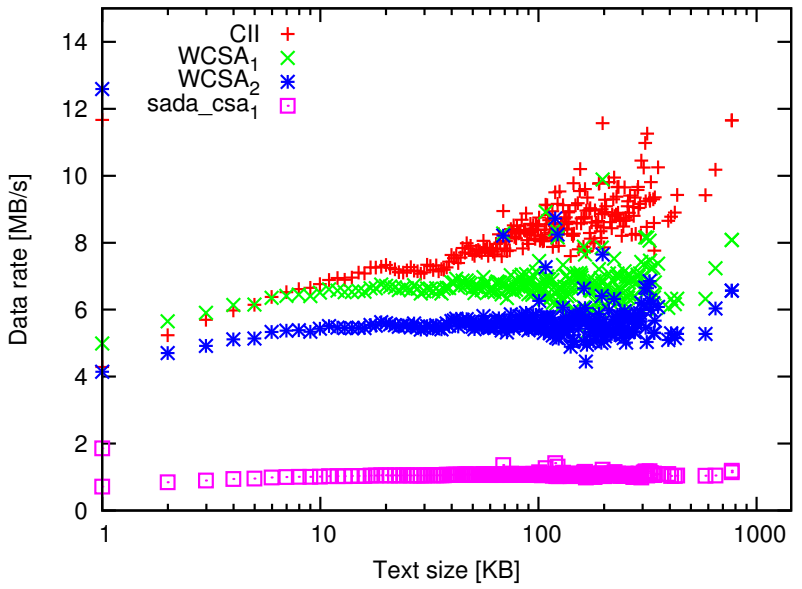
Figure 8.19(a) shows the average throughput for document reporting as a function of (KB-rounded) document size. We get noticeable differences between the data rate for reconstructing the normalized text and the data rate for reconstructing the original text (both without HTML). Our scheme allows roughly 5 MB per second of (original) text output (when using a text delta). Retrieving data from disk (assuming an access latency of 5 ms and a transfer rate of 100 MB/s) would be faster beginning at around 32 KB. This suggests a hybrid storage scheme keeping longer files on disk. But note that in many applications, texts are much shorter (see, for example, [42]). At first glance, it looks like parallel disks would be a way to mitigate disk access cost. However, with the advent of multicore processors, this option has become quite unattractive – one disk now costs more than a processing core so that even rack servers now tend to have less than one disk per processing core. Using blade servers, the ratio of cores to disks is even larger.

8.6.2 Comparison with Suffix Arrays

Figure 8.19(b) compares the four most competitive indexes when returning normalized text. The character-based suffix array sada_csa$_1$ is far behind the others. Its bandwidth is roughly five times smaller. WCSA$_1$ and CII are comparable, but CII can beat WCSA especially for documents larger than 10 KB. WCSA$_2$ shows again the performance for a combination of a document-grained inverted index and a word-based suffix array. Although it is behind WCSA$_1$ and CII, its bandwidth is still acceptable.



(a) WT2g



(b) WT2g*

Figure 8.19: *Average document reporting speed*

Chapter 9

Dynamization

All previous chapters assume a static scenario: the index is created from a collection of text documents once, and then the collection is never changed. However, on productive real-world systems this is often not admissible. Instead, new documents are inserted, old documents are deleted, or the contents of indexed documents are changed. Therefore, this chapter discusses how updates of documents can be implemented for our data structures.

There are different strategies for keeping (disk-based) inverted indexes up to date [74, 73, 59]. A simple approach is to rebuild an index from scratch when (possibly accumulated) updates arrive. However, this is in general very time-expensive, and pays off only if the number of documents that remain untouched is low. But in most cases only a small fraction of the documents in the collection is subject to update [76]. Therefore, it might be better to insert each update *in-place* into the single inverted lists. However, this requires frequent locks on some of the inverted lists resulting in a bad overall query performance. Thus, between the two extreme approaches, it has become popular to store updates in a smaller (fast updatable) *delta* index [126], and to merge them lazily [39] or in a batched fashion [28] into a larger (query-optimized) *main* index. Some approaches keep also several

indexes in parallel, and merge them (in some way at some time) repetitively to avoid the reorganization of a single huge main index [33, 72].

In delta index scenarios, search becomes a federated task requiring to merge the result sets of all indexes. On the other hand, a delta index can make updates instantly visible for users [70]. If there is any inconsistency between the results, the newer delta index overwrites the results of main index(es). Of course, the query performance during a merge is highly influenced by the merging strategy, the number of main and delta indexes, and their sizes [33].

The aspect of dynamization is certainly not the main focus of our work, but we believe that it is important to outline that our data structures could also be used in a dynamic environment. However, we have to keep in mind that our system is designed for a cluster of low-cost machines. If the search engine is subject to huge amounts of insertions, we suggest the use of additional cores.

For a single core of our main-memory search engine, we propose to use in addition to the main index (described in the previous chapters) a small but very fast updatable delta index. The delta index should be merged into the larger main index as soon as its size exceeds a certain threshold. During the merge, both indexes remain searchable as we store the updated parts separately, and activate them at one go. That reduces the down-time to a few seconds. Furthermore, we use a second delta index that maintains updates during the merge, and that becomes the new delta index after the merge. We briefly describe our delta index implementation and our merge algorithm in the following sections.

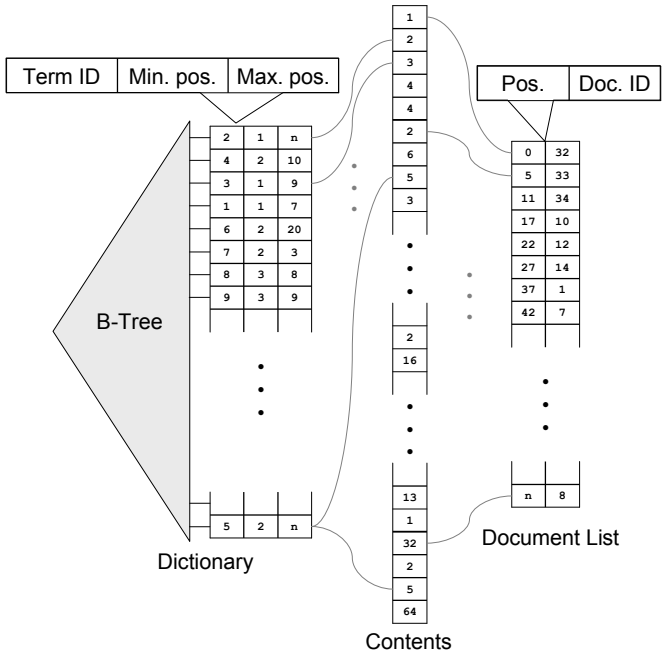


Figure 9.1: *Delta index scheme*

9.1 Fast Updatable Delta Indexes

Figure 9.1 shows the scheme of our fast-updatable delta index that is particularly designed for short documents. Instead of maintaining an inverted index, we store the term IDs of the documents in a large bit-compressed vector. That is, for updating a document we mainly have to append its sequence of term IDs to this *content vector*. Searching for some term ID requires a full-table scan over the vector. In our implementation this is highly optimized by loop-unrolling and hard-coded search functions for queries with a length of up to five terms. Alongside, we keep a list that stores all document offsets within the content vector together with the corresponding (global) document IDs. Positions found in the content vector have to be mapped to document IDs using this list. We keep a global bit-vector that has a flag for each document of the main and the delta index, indicating

whether an entry is valid or not. This guarantees that during a merge or when returning search results, the most recently committed document data is used.

The dictionary of our delta index has a B-tree structure, and therefore, it allows fast inserts. In addition, each leaf (corresponding to a term) stores the smallest and greatest index that a term can have in the content vector. The values can be used to truncate the full-table scan.

9.2 Merging Indexes

Our merging process begins with removing invalid entries from the document list of the delta index, and sorting it in increasing order of IDs. For each of the update documents, we use Algorithm 2 to fill temporary insertion and deletion lists that are similarly maintained as the inverted lists during the index construction (see Section 4.1.3). That is, each term t of the document index has an insertion list I_t^D that keeps the IDs of inserted documents containing t. Equally, each term t has a deletion list D_t^D where all deleted documents (formerly containing t) are listed. We implement the modification of documents as a combination of a deletion and an insertion. As we explain later, we also keep similar lists I_t^P and D_t^P for our positional index. In its second loop, Algorithm 2 exploits a particular advantage of our index scheme. As we store for each document a bag of words, we exactly know in which deletion lists its ID has to be added when the document is deleted (or updated). We therefore do not need to re-process the old version of the document (which probably no longer exists), nor do we need to rebuild the index. Furthermore, we do not need to use complicated techniques that try to reduce the update work by finding differences between the indexed and the update version of the document [77], as we can match insertion and deletion lists of a term before merging them into the index. Note that we can apply this for updating the inverted lists of the short-text index of Section 6.4 with minor modifications.

After the updates have been propagated from the delta index into the insertion and deletion lists, the lists have to be merged into the index. For

Algorithm 2 Pseudo-code for preparing updates of the main index

Require:
update document U,
document ID u,
current term set T_u,
document insertion lists I_t^D,
document deletion lists D_t^D,
positional insertion lists I_t^P,
positional deletion lists D_t^P,
inverted lists L_t^D

$S \leftarrow \emptyset$ {new bag of words}
$p \leftarrow 1$
while $p \leq \|U\|$ **do** {insert new terms}
  $t \leftarrow U[p]$ {term at position p of update document}
  if $d \notin S$ **then**
    $I_t^D \leftarrow I_t^D \cup \{u\}$ {prepare insertion into document index}
    $r \leftarrow \left\|I_t^P\right\| + i \,|\, L_t[i] = u$ {rank in insertion-merged inverted list}
    $l_r^t \leftarrow \{p\}$ {initialize list of positions}
    $I_t^P \leftarrow I_t^P \cup \{(r, l_r^t)\}$ {prepare insertion into positional index}
    $S \leftarrow S \cup \{t\}$ {append to updated bag of words}
  else
    $l_r^t \leftarrow l_r^t \cup \{p\}$ {append p to position list of last insertion}
  end if
  $p \leftarrow p + 1$
end while
$j \leftarrow 1$
while $j \leq \|T_u\|$ **do** {delete old terms}
  $t \leftarrow T_u[j]$
  $D_t^D \leftarrow D_t^D \cup \{u\}$ {prepare deletion from document index}
  $r \leftarrow i \,|\, L_t[i] = u$ {rank of t in current inverted list}
  $D_t^P \leftarrow D_t^P \cup \{r\}$ {prepare deletion from positional index}
  $j \leftarrow j + 1$
end while
return S {updated bag of words ready for replacing T_u}

our document-grained index, this is straightforward. We match the insertion and deletion lists, merge the results with the inverted lists of the index, and replace existing inverted lists by their updated versions. However, updating the positional index is more complex. Because the position list of a term t occurring in a document d is accessed using the rank of d in the document-grained inverted list of t (see Section 4.1.2), we need to store these ranks (rather than IDs) in the update lists I_t^P and D_t^P. An additional pitfall is that inserting an update somewhere in between an inverted list increments the ranks of all following entries. Therefore, ranks appended to the insertion lists I_t^P by Algorithm 2 are shifted by adding the number of insertions done so far (with increasing IDs or ranks). That is, we initially assume that there were insertions only and handle deletions (eventually also caused by changing documents) later. Consequently, Algorithm 3 that merges the position lists, carries an offset o that records the rank difference between the original and the updated version of the positions list. It is incremented whenever a new entry is appended to the result. Ranks in the deletion list correspond directly to the ranks in the original list. Hence they have to be shifted using offset o. They decide whether an old entry is inserted or not where the latter case is equivalent to a deletion. Finally, we replace also the updated positions lists with their old versions.

Summing up the process above, we use an *in-place* merge strategy which updates only the lists actually changed. Unfortunately, such strategies require a non-trivial memory management to avoid fragmentation, which consumes both space and time [34, 32]. However, we believe that reorganizing the space on main-memory systems is less expensive than on disk-based systems. Note that the document IDs of deleted documents still exist physically in the index and refer to empty documents (unless they were deleted from the end). But the IDs can be reused as soon as a new document is indexed.

Algorithm 3 Pseudo-code for our positional list merge algorithm

Require:
term ID t,
positional insertion list I_t^P,
positional deletion list D_t^P,
current position list L_t^P

$P \leftarrow \emptyset$ {output: new position list}
$i \leftarrow 0$ {control variable inserts}
$j \leftarrow 0$ {control variable deletions}
$o \leftarrow 0$ {rank offset between old and new list}
$s \leftarrow 0$ {rank}
while $s < \left\|L_t^P\right\| + \left\|I_t^P\right\|$ **do** {range of s assumes insertions only}
  if $i < \left\|U_t^P\right\|$ **then**
    $(r, l_r^t) \leftarrow I_t^P[i]$
    if $r = s$ **then** {check for an insertion}
      $P \leftarrow P \cup l_r^t$ {add new positions}
      $i \leftarrow i + 1$
      $o \leftarrow o + 1$ {increment offset}
      $s \leftarrow s + 1$
      continue while
    end if
  end if
  if $j < \left\|D_t^P\right\| \wedge D_t^P[j] = (s - o)$ **then**
    $j \leftarrow j + 1$ {delete old positions ('silent insertion')}
  else
    $P \leftarrow P \cup L_t^P[s - o]$ {insert old positions}
  end if
  $s \leftarrow s + 1$
end while
return P {return updated list}

9.3 Updating Phrase Indexes

The phrase index of Section 6.3 can be updated easily. Assuming that the threshold that decides whether a (two-term) phrase should be indexed or not is kept constant, only phrases that occur in updated documents can require a change within the phrase index. We therefore record all phrases of inserted and deleted documents, in a way like that described in Section 6.3.1. We get the phrases for deleted documents quickly by reconstructing their term sequences as described in Chapter 7. Finally, we loop through those phrases and check whether the inverted lists have to be deleted or replaced.

CHAPTER 10

Implementation Details

During our experiments, we have explored a huge number of index configurations and combinations. While providing convenient interchangeability between different approaches (with many parameters at all levels of abstraction), it was a big challenge to avoid negative influences on the performance. Even though our implementation is highly modular, there is highly optimized code behind well-defined APIs enabled by making extensive use of generic programming. But in addition, our code contains specialized versions for many tasks. So, it turned out that how we read or write the index data is crucial for the overall performance. This includes, for instance, using suitable memory allocation strategies, accessing the memory in a cache-friendly way, and being sparing in our use of functions like `memset` or `memcpy`. Giving all details of our implementation would go beyond the scope of this thesis, but here, we give some selected examples.

10.1 Bit-compressed Vectors

Bit-compressed vectors are among the key data structures in our search engine. This is not only because they show good performance (see Chapter 8), but also because they allow random access (see Section 3.2.1). Therefore, we show how we can implement them efficiently.

10.1.1 Encoding Integers

Built-in integer data types on modern computer systems are fixed-width words, where each type occupies a power-of-two number of whole bytes. Unfortunately, this means that we cannot address the main-memory with bit precision, as we may require for storing bit-compressed values of arbitrary width. Instead, we need to patch each bit-compressed value into an array of (typically) machine words by shifting it to the right position, masking out the target bits, and pasting it there using bit-wise OR operations. The target position and the required amount of shifts depend on the desired word width and index of the value in the list that we wish to encode: if i be the index, $w \leq W$ the width parameter of the bit-compression, and W the machine word width in bits, then the number of (left-)shifts is $s = W - w - (iw) \mod W$. This implies that the first position is $i = 0$ and that the encoding begins at a machine word bound. If $s < 0$ the value spans two target machine words. In this case, we shift the value $-s$ to the right for the first word and $W + s$ to the left for the second one. The indexes of these target machine words are $j = \lceil (iw)/W \rceil$ and $j + 1$, respectively (if the latter exists).

We have seen that bit-compression allows random access. But determining the target word(s) and the shift value is computationally far from negligible. Even if the integer division (or the modulo operation) by the width of a machine word W can be calculated using a fixed bit-shift (or a fixed bit-mask), we require at least one multiplication. This is unavoidable for writing a random single value, but when we encode many values linearly (either using a STL `vector`-like `push_back` or an iterator), we can

also calculate the shift and the machine word position incrementally. Given the current shift value s_k, the next value s_{k+1} is given by $s_{k+1} = s_k - w$ if $s > 0$ and $s_{k+1} = s_k + W$ if $s \leq 0$. Accordingly, we set $j = j + 1$ if $s \leq 0$. We can implement this using status variables that modern compilers might keep in CPU registers.

10.1.2 Decoding Integers

Generally, there is no big difference between encoding bit-compressed vectors and decoding them. For reading a single value or just a couple of values, we use the process as we described in the section above in reverse order. But often we are required to scan — we may even try to enforce this — a complete list linearly, as this is the most cache-friendly access pattern. While decoding a large number of bit-compressed values, we can observe that the way we shift the words recurs cyclically. At the latest, when we have consumed the least common multiple of W and w in bits, we arrive exactly again at a machine word bound. This is after W machine words in the worst case. Therefore, we can also read bit-compressed values in chunks of $m_w \leq W$ machine words: our implementation contains (for each $w \leq 64$) hard-coded sequences that decode m_w machine words in one go without calculating any shifts or masks. We pre-calculate those variables during compile time using generic meta-programming. To follow our well-defined API, this is encapsulated in a read-only iterator that internally holds a buffer (with the size of a chunk), whose dereference operator returns uncompressed values directly from this buffer.

10.2 Accessing Lookup Lists

In Section 3.4.1 we introduced Lookup lists. A Lookup list is a two-level inverted list data structure that logically splits its content values into buckets based on their most-significant bits. The actual data is stored in the bottom level, and the top level holds pointers to each of the buckets for allowing fast access to the values. We used Lookup lists for the larger inverted lists

in our search engine, and therefore, they are responsible for the time efficiency (see Section 8.2). Depending on the task, we use different methods for operating the lists.

10.2.1 Bucket Search

Algorithm Lookup of Section 3.4.1 intersects a Lookup list with an arbitrary smaller list by traversing the smaller list, and locating its values in the larger Lookup list. Because the values in the Lookup list are organized in buckets (based on their most-significant bits), we can jump almost exactly to the position where we expect a value that we are searching. In most cases, it suffices to read just a couple of values to decide whether we actually can find the element or not. Similarly quickly, we can determine the rank of a given value within these lists. We need ranks, for instance, for accessing the positional index (see Section 4.1.2). In these cases, we use bucket iterators that can be exactly positioned and that can decode the bucket value by value. In particular, we would not use the buffered iterator from the previous Section 10.1.2.

10.2.2 Unpacking Lookup Lists

When intersecting two inverted lists, algorithm Lookup traverses the smaller list linearly. Thus, if we have two Lookup lists as input, then we have to decode one of them completely from beginning to end. The simplest way to do this is to iterate over the buckets, and to unpack each of their values in an inner loop. However, our experiments showed that this is not very efficient. Therefore, we first unpack the top level before we iterate through the bottom. More precisely, we build a bit-vector that has a bit set at all the positions where a new bucket starts in the bottom. Note that this bit-vector uses at most as much memory as the coding of the bottom level. In addition, we hold an uncompressed vector that stores an offset for each of the set bits. The offset carries the most-significant bits of the values in the corresponding bucket, i.e. it is equal to the shifted bucket index. Thus, for unpacking a list, we iterate through the bottom and modify the

most significant bits of the current value whenever we hit a set bit. This approach has different advantages. First, we can decompress the normally bit-compressed top level in chunks (see Section 10.1.2). Second, because we do not need a top-level iterator, we can reduce the number of status variables that might replace each other from CPU registers. Third, we obtain a simpler program flow without too many branches. While iterating through the list, checking a single bit suffices to find out if we are currently entering a new bucket or not. This procedure is again encapsulated within a read-only iterator.

Chapter 11

Conclusion

Text search engines have to manage huge amounts of data. Therefore, classical search systems hold the major parts of their index structures on hard disks. But given growing amounts of RAM [93, 57] it becomes possible to store also larger indexes in main memory. In particular, in scenarios where fast response times are essential, using in-memory approaches is a great opportunity. In this thesis, we have studied data structures and algorithms for in-memory text search engines in a bottom-up fashion.

11.1 Contributions

We have designed, implemented, and extensively studied the modular core of a purely main-memory based full text search engine. We have explored different inverted list representations with respect to both compression and performance of pair-wise intersections. We have used three different encoding schemes covering a wide range of space-time tradeoffs. For Escaping (which provides an interesting tradeoff between the compactest and fastest encodings), we have given a new solution for how a space-optimal parameter can easily be determined (assuming a normal distribution of the differences). Inspired by the idea of randomly assigned document IDs, we have designed and analyzed a new inverted list data structure that uses auxiliary information for speeding up important operations which avoids (in contrast to others) to storage of redundant information.

We have seen that in practice roughly 90 % of the pair-wise intersections caused by AND queries have list length ratios $\frac{m}{n} < 0.1$. Hence, it is worth looking for an algorithm that is especially suitable for those small ratios. The classical algorithm Zipper is good for nearly equal lengths but at least for $\frac{m}{n} < 0.2$ it cannot compete with algorithms that use auxiliary information. Using a higher compression, this disadvantage of Zipper compared to the two-level algorithms is even larger. Algorithm Skipper, for instance, is noticeably better for smaller ratios. Asymptotically efficient algorithms like Baeza-Yates cannot realize their theoretical advantages in practice. For all $\frac{m}{n}$ there are other algorithms that are considerably faster. In addition, Baeza-Yates and its successors are comparatively complicated and get even more complicated when combined with compression. Rather than seeing this as another disadvantage, its supporters could argue that a clever implementation could considerably improve on our results.

Our new (not purely comparison based) intersection algorithm Lookup is at the same time simple and among the best algorithms over the entire spectrum of length ratios. For small $\frac{m}{n}$ it can claim considerable speedups over all other algorithms in our experiments. Randomization allows interesting performance guarantees on both execution time and space require-

ment. However, it seems less advantageous in practice to obliterate the dependencies present in real world inputs. Even Lookup itself profits from these dependencies.

Using these findings, we have shown that a carefully engineered in-memory search engine based on inverted indexes and positional indexes allows fast queries using considerably less memory than the input size (while being able to reconstruct the input).

We have compared our approach with recent implementations of character-based suffix arrays. Their space consumption is comparable to our inverted index. However, they need more time and space during construction. As inverted indexes are well designed to answer AND queries, suffix arrays cannot compete with them in this field. Their response times are orders of magnitude slower. But for phrase queries, suffix arrays perform quite well. Nevertheless, they have a major disadvantage when generating large results. We have seen that in practice more than 50 % of all phrase queries are of length two. Furthermore, our experiments have shown that these short queries are the most expensive ones as they produce large result sets. This aspect narrows the advantage of suffix arrays in some cases where the average querying time is already below 10 ms for all competitors. We believe that it is important to afford good worst-case times, as it is often difficult to explain to users of a search engine that they have to wait (for example) up to 8 s to getting their answers.

Word-based suffix arrays can dispel this worst-case behavior and can even improve the average phrase query execution time of their character-based pendants. But they are still slow for single term queries as well as for AND queries. In particular, they have poor worst-case behavior in common with character-based suffix arrays, when queries produce a large number of hits. Furthermore, they require more than twice the time for index construction than compressed inverted indexes. Therefore, a purely suffix array based text search engine is not yet a serious alternative to an inverted index based system. But note that there are other scenarios than (English) text search engines, where suffix arrays might be the better choice.

The main problem area for inverted indexes is that of phrase queries. In this area, the suffix arrays were most likely to be a serious competitor for inverted indexes. But, our experiments have shown that our new and flexible two-term phrase indexing scheme can be used to achieve significant query time speedup compensating this disadvantage. It is adaptable to the underlying storage system and can achieve significant speedups on both in-memory and external memory search systems. Also, it yields interesting theoretical robustness for difficult queries. We have explored different cost models for selecting suitable phrases to be indexed. Our minimum-based cost model outperforms others clearly, and, in particular, it provides high scalability. Partial phrase indexes are satisfactory only in the optimal situation where we use perfectly fitting training logs with easy phrases. In this case, the advantages of the partial phrase index and the two-term phrase index seem to complement one other.

We have used our phrase query acceleration technique to design a new index scheme suitable for small documents. It outperforms positional indexes in terms of compression and query performance at the same time. Furthermore it is tunable to obtain different space-time tradeoffs. For instance, our experiments have shown that using 70 % of the space of a positional index, our short-text index scheme already has better average and worst-case query times. Using the same amount of space, it is more than 25 times faster than the classical approach.

We have designed a new framework for reconstructing document contents using inverted indexes. It stores only the fraction of the information that is not yet contained in the index but that is essential for reporting the documents. This results in a compact representation of the text (using less space than the input) that allows fast querying.

We have outlined that our data structures can efficiently handle changes in the indexed document collection. Finally, we have explored some low-level details of our main memory search engine implementation.

11.2 Outlook

Although our search engine is implemented carefully and a lot of work has been put into details, we believe that we still have not yet reached the end of what can be achieved with our approach. For accessing our compressed data structures, a promising approach (especially for bit-compressed lists) seems to be to use low-level SIMD instructions (see, for example, [121]). Lookup lists seem also to provide a good basis for even more fine-grained parallelism. Due to their organization in power-of-two ranges, we could easily intersect corresponding ranges (probably with different sized buckets) of Lookup lists in parallel.

The text delta compression could be improved by maintaining a dictionary storing *building blocks* for sequences of separators occurring between terms. Perhaps we could combine this with a mechanism that learns recurring patterns of terms and separators during indexing, and that generates grammar-like rules (similar to [97]) using the dictionary terms and the building blocks as terminals.

Our dictionary is very simple, so for example it does not allow efficient pattern or fuzzy search. It is also not compressed. Here, existing work could be used or adapted (see, for example, [133, 10, 55]).

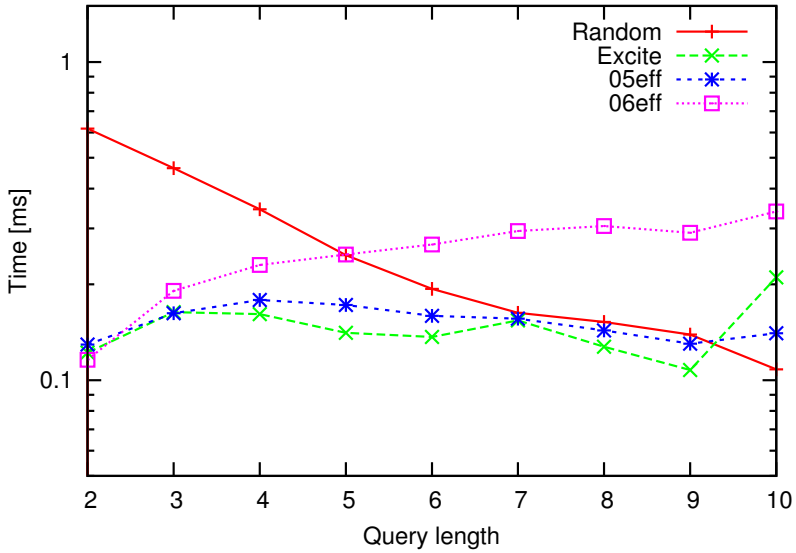
Although we have explored two very important types of queries, there are other query types that might also be fundamental. Our search engine core does not yet support ranking. It returns the results in the order of document IDs. Ranking is a major area of research, and there are approaches that could be applied to our engine (see, for example, [91, 134]).

In the introduction we mentioned that our text index should be seen as a part of a large distributed search system. A next step would be to embed our index into a multicore framework that can also support techniques for load balancing. Then, it would be interesting to compare the system with a distributed state-of-the-art disk-based search engine. In particular if *interactive response time* (see also [98]) is an essential claim, the main-memory based system could be a very competitive approach (see also [22]).

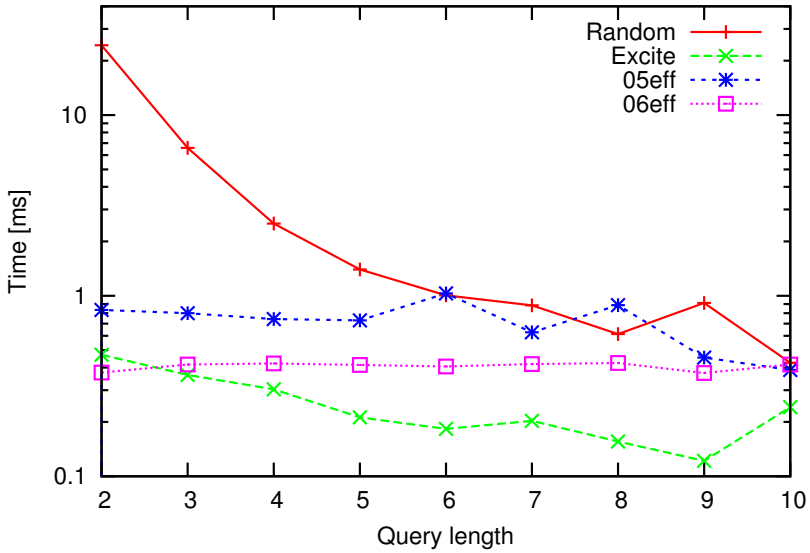
APPENDIX A

Results on GOV2

In this section, we give bottom-line results for some of our experiments for a randomly selected sixty-fourth subset of the GOV2 corpus containing roughly 400 000 documents. We used the query logs described in Section 8.1.4, where we generated the Random log using the complete GOV2 corpus and not only the selected subset.



(a) Average AND querying times



(b) Average phrase querying times

Figure A.1: *CII on a subset of GOV2*

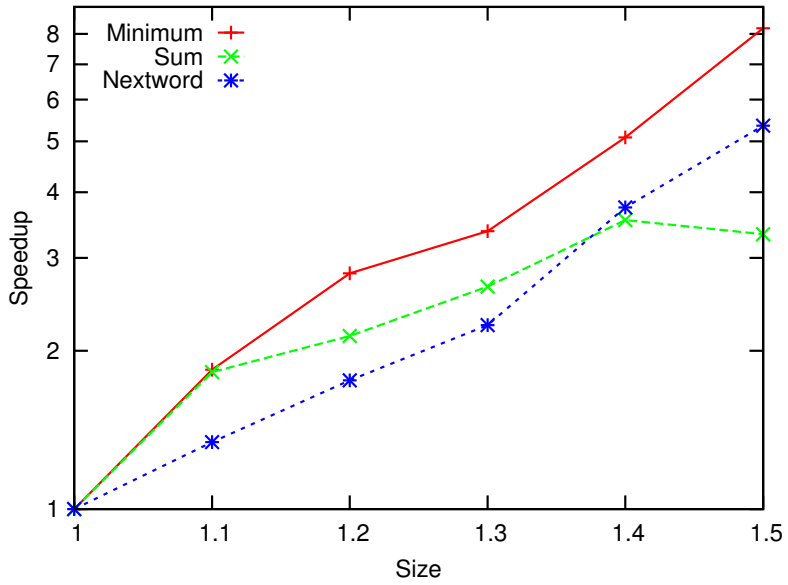


Figure A.2: *Random log query time speedup using different cost models*

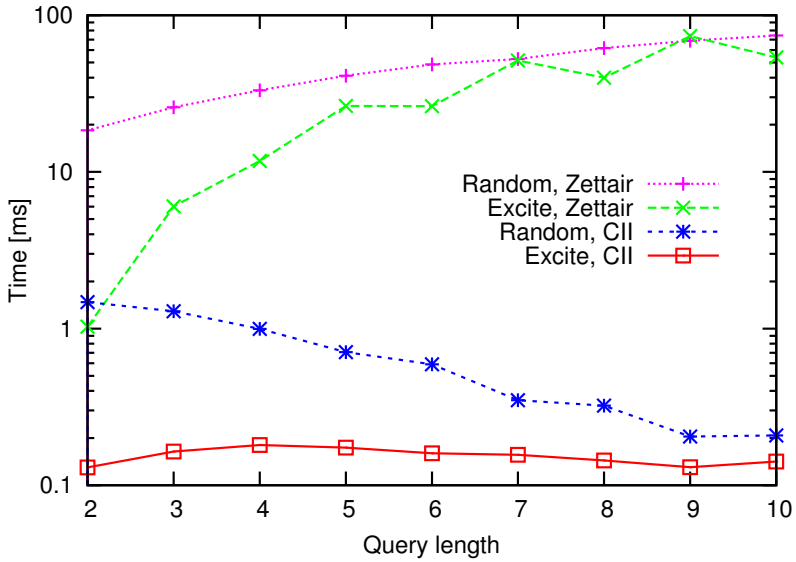
As Figure A.1(a) shows, the average AND querying times are very fast (< 1 ms) for all logs. The average phrase querying times shown in Figure A.1(b) are good as well. The Random phrase queries are somewhat more expensive in particular for small lengths. This coincides with the results for the smaller WT2g corpus (see Section 8.5). Figure A.2 shows that for the difficult Random log the minimum-based cost model again outperforms the others for our in-memory index (see Section 8.5.4).

Concluding the results, our search engine seems to provide reasonable scalability. It provided comparable times in our tests when the number of documents was significantly increased (relating to Chapter 8).

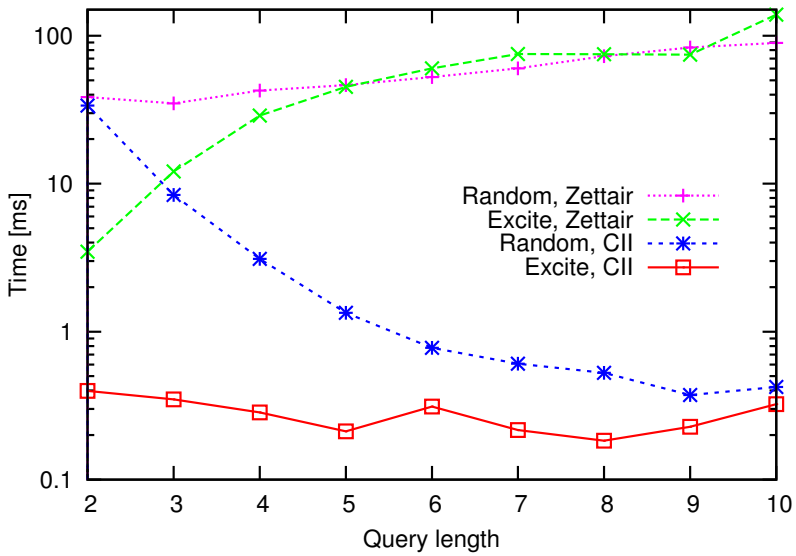
APPENDIX B

In-memory vs. Disk-based

The following plots show a comparison between our in-memory text search engine implementation as described in this thesis and the disk-based search engine Zettair [125]. We used the WT2g corpus introduced in [62] to run the queries of the Excite log and the Random query log (see Section 8.1.4). All plots show average query times against the query length, i.e. the number of words in the query.



(a) Average AND querying times



(b) Average phrase querying times

Figure B.1: *CII vs. Zettair on WT2g*

Bibliography

[1] ABUSUKHON, A., OAKES, M. P., TALIB, M., AND ABDALLA, A. M. Comparison between document-based, term-based and hybrid partitioning. In *First International Conference on the Applications of Digital Information and Web Technologies, 2008. ICADIWT 2008.* (Aug. 2008), IEEE Computer Society, pp. 90–95. (Cited on page 27.)

[2] ANH, V. N., AND MOFFAT, A. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st Annual International ACM Conference on Research and Development in Information Retrieval SIGIR 1998* (New York, NY, USA, 1998), ACM, pp. 290–297. (Cited on pages 7 and 20.)

[3] ANH, V. N., AND MOFFAT, A. Random access compressed inverted lists. In *Proceedings of the 9th Australasian Database Conference ADC 1998* (1998), Australian Computer Science Communications, Springer, pp. 1–12. (Cited on page 7.)

[4] ANH, V. N., AND MOFFAT, A. Index compression using fixed binary codewords. In *Proceedings of the 15th Australasian Database Conference ADC 2004* (Darlinghurst, Australia, 2004), Australian Computer Society, pp. 61–67. (Cited on page 6.)

[5] ANH, V. N., AND MOFFAT, A. Inverted index compression using word-aligned binary codes. *Information Retrieval 8*, 1 (2005), 151–166. (Cited on page 6.)

[6] ANH, V. N., AND MOFFAT, A. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering 18*, 6 (2006), 857–861. (Cited on page 6.)

[7] ANH, V. N., AND MOFFAT, A. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval SIGIR 2006* (New York, NY, USA, 2006), ACM, pp. 372–379. (Cited on page 9.)

[8] ANH, V. N., AND MOFFAT, A. Structured index organizations for high-throughput text querying. In *Proceedings of the 13th International Symposium on String Processing and Information Retrieval SPIRE 2006* (2006), vol. 4209 of *LNCS*, Springer, pp. 304–315. (Cited on page 32.)

[9] ARROYUELO, D., NAVARRO, G., AND SADAKANE, K. Reducing the space requirement of LZ-index. In *Proceedings of the 17th Symposium on Combinatorial Pattern Matching CPM 2006* (2006), LNCS, Springer, pp. 318–329. (Cited on page 74.)

[10] ASKITIS, N., AND SINHA, R. HAT-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the 30th Australasian Conference on Computer Science ACSC 2007* (Darlinghurst, Australia, 2007), Australian Computer Society, pp. 97–105. (Cited on page 115.)

[11] BAEZA-YATES, R. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Symposium on Combinatorial Pattern Matching CPM 2004* (2004), vol. 3109 of *LNCS*, Springer, pp. 400–408. (Cited on pages 8 and 21.)

[12] BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F. P., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. Design trade-offs for search engine caching. *ACM Transactions on the Web 2*, 4 (2008), 1–28. (Cited on page 10.)

[13] BAEZA-YATES, R. A., AND SALINGER, A. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval SPIRE 2005* (2005), vol. 3772 of *LNCS*, Springer, pp. 13–24. (Cited on pages 8, 9 and 21.)

[14] BAHLE, D., WILLIAMS, H. E., AND ZOBEL, J. Compaction techniques for nextword indexes. In *Proceedings of the 8th Symposium on String Processing and Information Retrieval SPIRE 2001* (2001), IEEE Computer Society. (Cited on page 11.)

[15] BAHLE, D., WILLIAMS, H. E., AND ZOBEL, J. Optimised phrase querying and browsing in text databases. In *Proceedings of the 24th Australasian Computer Science Conference* (2001), pp. 11–19. (Cited on pages 44 and 47.)

[16] BAHLE, D., WILLIAMS, H. E., AND ZOBEL, J. Efficient phrase querying with an auxiliary index. In *Proceedings of the 25th Annual International Conference on Research and Development in Information Retrieval SIGIR 2002* (2002), ACM. (Cited on pages 11, 44, 45, 46 and 47.)

[17] BARBAY, J., LÓPEZ-ORTIZ, A., AND LU, T. Faster adaptive set intersections for text searching. In *Proceedings of the Fifth International Workshop on Experimental Algorithms WEA 2006* (2006), C. Àlvarez and M. J. Serna, Eds., vol. 4007 of *LNCS*, Springer, pp. 146–157. (Cited on pages 8 and 40.)

[18] BARBAY, J., AND NAVARRO, G. Compressed representations of permutations, and applications. In *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)* (Dagstuhl, Germany, 2009). (Cited on page 8.)

[19] BARROSO, L. A., DEAN, J., AND HOLZLE, U. Web search for a planet: The google cluster architecture. *Micro, IEEE 23*, 2 (2003), 22–28. (Cited on pages 1 and 28.)

[20] BAST, H., MORTENSEN, C. W., AND WEBER, I. Output-sensitive autocompletion search. *Information Retrieval* (2008), 269–286. Special Issue on SPIRE 2006. (Cited on page 12.)

[21] BAST, H. ET AL. Seminar: Searching with suffix arrays. `http://search.mpi-inf.mpg.de/wiki/IrSeminarWs06`, 2007. (Cited on page 12.)

[22] BENDER, M., MICHEL, S., TRIANTAFILLOU, P., AND WEIKUM, G. Design alternatives for large-scale web search: Alexander was great, Aeneas a pioneer, and Anakin has the force. In *Workshop on Large Scale Distributed Systems for Information Retrieval* (2007), pp. 16–22. (Cited on page 115.)

[23] BENTLEY, J. L., AND YAO, A. C.-C. An almost optimal algorithm for unbounded searching. *Information Processing Letters 5*, 3 (1976), 82–87. (Cited on page 21.)

[24] BLANCO, R., AND BARREIRO, A. TSP and cluster-based solutions to the reassignment of document identifiers. *Information Retrieval 9*, 4 (Sept. 2006), 499–517. (Cited on page 6.)

[25] BLANDFORD, D. *Compact Data Structures with Fast Queries*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. (Cited on pages 6 and 28.)

[26] BLANDFORD, D., AND BLELLOCH, G. Index compression through document reordering. In *Proceedings of the Data Compression Conference DCC 2002* (Washington, DC, USA, 2002), p. 342. (Cited on page 6.)

[27] BRISABOA, N. R., FARIÑA, A., NAVARRO, G., PLACES, Á. S., AND RODRÍGUEZ, E. Self-indexing natural language. In *15th International Symposium on String Processing and Information Retrieval SPIRE 2008* (2008), A. Amir, A. Turpin, and A. Moffat, Eds., vol. 5280 of *LNCS*, Springer, pp. 121–132. (Cited on pages 13, 50, 51, 53 and 75.)

[28] BROWN, E. W., CALLAN, J. P., AND CROFT, W. B. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Data Bases VLDB 1994* (San Francisco, CA, USA, 1994), Morgan Kaufmann, pp. 192–202. (Cited on page 97.)

[29] BROWN, E. W., CALLAN, J. P., CROFT, W. B., AND MOSS, J. E. B. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the Fourth International Conference on Extending Database Technology EDBT 1994* (1994), vol. 779 of *LNCS*, Springer, pp. 365–378. (Cited on pages 9 and 10.)

[30] BÜTTCHER, S., CLARKE, C., AND SOBOROFF, I. TREC 2006 efficiency topics. `http://trec.nist.gov/data/terabyte/06/06.efficiency_topics.tar.gz`, 2006. (Cited on page 59.)

[31] BÜTTCHER, S., CLARKE, C., AND SOBOROFF, I. The TREC 2006 terabyte track. In *Proceedings of the 15th Text Retrieval Conference (TREC 2006)* (2006). (Cited on page 59.)

[32] BÜTTCHER, S., AND CLARKE, C. L. Hybrid index maintenance for contiguous inverted lists. *Information Retrieval 11*, 3 (2008), 175–207. (Cited on page 102.)

[33] BÜTTCHER, S., AND CLARKE, C. L. A. Indexing time vs. query time: Trade-offs in dynamic information retrieval systems. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management CIKM 2005* (New York, NY, USA, 2005), ACM, pp. 317–318. (Cited on page 98.)

[34] BÜTTCHER, S., CLARKE, C. L. A., AND LUSHMAN, B. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval SIGIR 2006* (New York, NY, USA, 2006), ACM, pp. 356–363. (Cited on page 102.)

[35] CACHEDA, F., CARNEIRO, V., PLACHOURAS, V., AND OUNIS, I. Performance comparison of clustered and replicated information retrieval systems. In *Proceedings of the 29th European Conference on Information Retrieval ECIR 2007* (2007), vol. 4425 of *LNCS*, Springer, pp. 124–135. (Cited on page 28.)

[36] CAMBAZOGLU, B. B., CATAL, A., AND AYKANAT, C. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *ISCIS* (2006), pp. 717–725. (Cited on page 27.)

[37] CARMEL, D., COHEN, D., FAGIN, R., FARCHI, E., HERSCOVICI, M., MAAREK, Y. S., AND SOFFER, A. Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM Conference on Research and Development in Information Retrieval SIGIR 2001* (2001), pp. 43–50. (Cited on page 6.)

[38] CHANG, M., AND POON, C. K. Efficient phrase querying with common phrase index. In *Proceedings of the 28th European Conference on Information Retrieval ECIR 2006* (2006), vol. 3936 of *LNCS*, Springer. (Cited on page 11.)

[39] CHIUEH, T., AND HUANG, L. Efficient real-time index updates in text retrieval systems. Tech. rep., Experimental Computer Systems Lab, Department of Computer Science, State University of New York, 1999. (Cited on page 97.)

[40] CLARKE, C., SCHOLER, F., AND SOBOROFF, I. TREC 2005 efficiency topics. `http://trec.nist.gov/data/terabyte/05/05.efficiency_topics.gz`, 2005. (Cited on page 59.)

[41] CLARKE, C., SCHOLER, F., AND SOBOROFF, I. The TREC 2005 terabyte track. In *Proceedings of the 14th Text Retrieval Conference (TREC 2005)* (2005). (Cited on page 59.)

[42] CLARKE, C., SOBOROFF, I., AND CRASWELL, N. GOV2 test collection. `http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`, 2004. (Cited on pages 55 and 94.)

[43] CLAUDE, F., FARIÑA, A., AND NAVARRO, G. Re-pair compression of inverted lists. Tech. Rep. TR/DCC-2008-16, University of Chile, Department of Computer Science, 2008. (Cited on page 9.)

[44] CULPEPPER, J. S. *Efficient Data Representations for Information Retrieval*. PhD thesis, Melbourne University, 2007. (Cited on page 16.)

[45] CULPEPPER, J. S., AND MOFFAT, A. Compact set representation for information retrieval. In *14th International Symposium on String Processing and Information Retrieval SPIRE 2007* (2007), vol. 4726 of *LNCS*, Springer, pp. 137–148. (Cited on pages 9, 16, 20 and 40.)

[46] CUTTING, D., AND PEDERSEN, J. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th Annual International Conference on Research and Development in Information Retrieval SIGIR 1990* (New York, NY, USA, 1990), ACM Press, pp. 405–411. (Cited on page 9.)

[47] DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms SODA 2000* (Philadelphia, PA, USA, 2000), SIAM, pp. 743–752. (Cited on page 8.)

[48] DEMAINE, E. D., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the Third Workshop on Algorithm Engineering and Experimentation ALENEX 2001* (2001), vol. 2153 of *LNCS*, Springer, pp. 91–104. (Cited on pages 8 and 40.)

[49] DEMENTIEV, R., SANDERS, P., SCHULTES, D., AND SIBEYN, J. Engineering an external memory minimum spanning tree algorithm. In *Proceedings of the Third IFIP International Conference on Theoretical Computer Science IFIP TCS 2004* (Toulouse, 2004). (Cited on page 27.)

[50] FAGAN, J. Automatic phrase indexing for document retrieval. In *Proceedings of the 10th Annual International Conference on Research and Development in Information Retrieval SIGIR 1987* (1987), ACM. (Cited on page 10.)

[51] FERRAGINA, P., AND FISCHER, J. Suffix arrays on words. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM 2007* (2007), vol. 4580 of *LNCS*, Springer, pp. 328–339. (Cited on page 12.)

[52] FERRAGINA, P., AND MANZINI, G. Indexing compressed text. *Journal of the ACM 52*, 4 (2005), 552–581. (Cited on page 74.)

[53] FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms 3*, 2 (2007), 20. (Cited on page 74.)

[54] FERRAGINA, P., AND NAVARRO, G. The pizza&chili website. `http://pizzachili.dcc.uchile.cl`, 2005. (Cited on pages 12, 50 and 53.)

[55] FERRAGINA, P., AND VENTURINI, R. Compressed permuterm index. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval SIGIR 2007* (New York, NY, USA, 2007), ACM, pp. 535–542. (Cited on pages 34 and 115.)

[56] GAILLY, J., AND ADLER, M. Zlib compression library. `http://www.zlib.net/`. (Cited on page 71.)

[57] GELSINGER, P. Moore's law: "We See No End In Sight". `http://websphere.sys-con.com/node/557154`. (Cited on page 111.)

[58] GONZÁLEZ, R., AND NAVARRO, G. Compressed text indexes with fast locate. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM 2007* (2007), LNCS, Springer, pp. 216–227. (Cited on pages 12 and 74.)

[59] GUO, R., CHENG, X., XU, H., AND WANG, B. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management CIKM 2007* (New York, NY, USA, 2007), ACM, pp. 751–760. (Cited on page 97.)

[60] GUTWIN, C., PAYNTER, G., WITTEN, I., NEVILL-MANNING, C., AND FRANK, E. Improving browsing in digital libraries with keyphrase indexes. *Decision Support Systems 27*, 1-2 (1999). (Cited on pages 10 and 43.)

[61] HAWKING, D., CRASWELL, N., AND THISTLEWAITE, P. Overview of the TREC-7 very large collection track. In *Proceedings of the Seventh Text Retrieval Conference (TREC-7)* (1999). (Cited on page 54.)

[62] HAWKING, D., VOORHEES, E., CRASWELL, N., AND BAILEY, P. Overview of the TREC-8 web track. In *Proceedings of the Eighth Text Retrieval Conference (TREC-8)* (2000). `http://ir.dcs.gla.ac.uk/test_collections/access_to_data.html`. (Cited on pages 54 and 121.)

[63] HSIEH, P. Superfasthash. `http://www.azillionmonkeys.com/qed/hash.html`, 2004. (Cited on page 33.)

[64] HWANG, F. K., AND LIN, S. Optimal merging of 2 elements with n elements. *Acta Informatica 1* (1971), 145–158. (Cited on pages 8 and 21.)

[65] HWANG, F. K., AND LIN, S. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM Journal on Computing 1* (1972), 31–39. (Cited on page 8.)

[66] JANSEN, B. J., SPINK, A., BATEMAN, J., AND SARACEVIC, T. Real life information retrieval: a study of user queries on the web. *SIGIR Forum 32*, 1 (1998), 5–17. (Cited on pages 44 and 56.)

[67] JÓNSSON, B. T., FRANKLIN, M. J., AND SRIVASTAVA, D. Interaction of query evaluation and buffer management for information retrieval. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data SIGMOD 1998* (New York, NY, USA, 1998), ACM, pp. 118–129. (Cited on page 10.)

[68] KNUTH, D. E., MORRIS, J. J. H., AND PRATT, V. R. Fast pattern matching in strings. *SIAM Journal on Computing 6*, 2 (1977). (Cited on page 48.)

[69] KUHN, M. `http://www.cl.cam.ac.uk/~mgk25/download/transtab.tar.gz`, 2000. (Cited on page 54.)

[70] LEMPEL, R., MASS, Y., OFEK-KOIFMAN, S., SHEINWALD, D., PETRUSCHKA, Y., AND SIVAN, R. Just in time indexing for up to the second search. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management CIKM 2007* (New York, NY, USA, 2007), ACM, pp. 97–106. (Cited on page 98.)

[71] LEMPEL, R., AND MORAN, S. Competitive caching of query results in search engines. *Theoretical Computer Science 324*, 2 (2004), 253–271. (Cited on page 10.)

[72] LESTER, N., MOFFAT, A., AND ZOBEL, J. Efficient online index construction for text databases. *ACM Transactions on Database Systems 33*, 3 (2008), 1–33. (Cited on page 98.)

[73] LESTER, N., ZOBEL, J., AND WILLIAMS, H. Efficient online index maintenance for contiguous inverted lists. *Information Processing and Management 42*, 4 (2006), 916–933. (Cited on page 97.)

[74] LESTER, N., ZOBEL, J., AND WILLIAMS, H. E. In-place versus rebuild versus re-merge: Index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian Conference on Computer Science ACSC 2004* (Darlinghurst, Australia, 2004), Australian Computer Society, pp. 15–23. (Cited on page 97.)

[75] LIBXML2, 1999. `http://xmlsoft.org`. (Cited on page 54.)

[76] LIM, L., WANG, M., PADMANABHAN, S., VITTER, J. S., AND AGARWAL, R. Characterizing web document change. In *Proceedings of the Second International Conference on Web-Age Information Management WAIM 2001* (2001), Springer, pp. 133–144. (Cited on page 97.)

[77] LIM, L., WANG, M., PADMANABHAN, S., VITTER, J. S., AND AGARWAL, R. Dynamic maintenance of web indexes using landmarks. In *Proceedings of the 12th International Conference on World Wide Web WWW 2003* (New York, NY, USA, 2003), ACM, pp. 102–111. (Cited on page 100.)

[78] LONG, X., AND SUEL, T. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th International Conference on World Wide Web WWW 2005* (New York, NY, USA, 2005), ACM, pp. 257–266. (Cited on page 10.)

[79] LUBY, M., AND RACKOFF, C. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing 17*, 2 (1988), 373–386. (Cited on page 27.)

[80] LUK, R. W. P., AND LAM, W. Efficient in-memory extensible inverted file. *Information Systems 32*, 5 (2007), 733–754. (Cited on page 9.)

[81] MÄKINEN, V., AND NAVARRO, G. Compressed compact suffix arrays. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching CPM 2004* (2004), vol. 3109 of *LNCS*, Springer, pp. 420–433. (Cited on page 74.)

[82] MÄKINEN, V., AND NAVARRO, G. Run-length FM-index. In *Proceedings of DIMACS Workshop on "The Burrows-Wheeler Transform": Ten Years Later* (2004), pp. 17–19. August 19–20. Informal Proceedings. (Cited on page 74.)

[83] MÄKINEN, V., AND NAVARRO, G. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12*, 1 (2005), 40–66. (Cited on page 74.)

[84] MANNING, C. D., RAGHAVAN, P., AND SCHÜTZE, H. *Introduction to Information Retrieval*. Cambridge University Press, 2008. `http://informationretrieval.org/`. (Cited on page 6.)

[85] MARKATOS, E. P. On caching search engine query results. 137–143. (Cited on page 10.)

[86] MOFFAT, A., AND CULPEPPER, J. S. Hybrid bitvector index compression. In *Proceedings of the 12th Australasian Document Computing Symposium ADCS 2007* (Melbourne, Australia, 2007), RMIT University, pp. 25–31. (Cited on pages 9, 30 and 40.)

[87] MOFFAT, A., AND STUIVER, L. Exploiting clustering in inverted file compression. In *Proceedings of the Sixth Conference on Data Compression DCC 1996* (Washington, DC, USA, 1996), IEEE Computer Society, p. 82. (Cited on pages 6 and 27.)

[88] MOFFAT, A., AND STUIVER, L. Binary interpolative coding for effective index compression. *Information Retrieval 3*, 1 (2000), 25–47. (Cited on page 6.)

[89] MOFFAT, A., AND TURPIN, A. *Compression and Coding Algorithms*. Kluwer Academic, Norwell, MA, USA, 2002. (Cited on page 19.)

[90] MOFFAT, A., AND ZOBEL, J. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems 14*, 4 (1996), 349–379. (Cited on pages 7, 8 and 22.)

[91] MOFFAT, A., ZOBEL, J., AND SACKS-DAVIS, R. Memory efficient ranking. *Information Processing and Management 30*, 6 (1994), 733–744. (Cited on page 115.)

[92] MOFFAT, A., ZOBEL, W., AND KLEIN, S. T. Improved inverted file processing for large text databases. In *Proceedings of the 6th Australasian Database Conference ADC 1995* (1995), vol. 17 of *Australian Computer Science Communications*, Springer, pp. 162–171. (Cited on page 7.)

[93] MOORE, G. E. Cramming more components onto integrated circuits. *Proceedings of the IEEE 86*, 1 (Jan. 1998), 82–85. (Cited on page 111.)

[94] MUCCI, P. E. A. Performance API. `http://icl.cs.utk.edu/papi/`, 2005. (Cited on page 54.)

[95] NAVARRO, G. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms 2*, 1 (2004), 87–114. (Cited on page 74.)

[96] NAVARRO, G., AND MÄKINEN, V. Compressed full-text indexes. *ACM Computing Surveys 39*, 1 (2007). (Cited on page 12.)

[97] NEVILL-MANNING, C. G., AND WITTEN, I. H. Identifying hierarchical structure in sequences: a linear-time algorithm. *Journal of Artificial Intelligence Research 7* (1997), 67–82. (Cited on page 115.)

[98] NIELSEN, J. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA, USA, 1993. (Cited on page 115.)

[99] OKANOHARA, D., AND SADAKANE, K. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments ALENEX 2007* (2007), SIAM, pp. 60–70. (Cited on page 16.)

[100] PUGLISI, S. J., SMYTH, W. F., AND TURPIN, A. Inverted files versus suffix arrays for locating patterns in primary memory. In *SPIRE* (2006), vol. 4209 of *LNCS*, Springer, pp. 122–133. (Cited on page 12.)

[101] R DEVELOPMENT CORE TEAM. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. `http://www.r-project.org`. (Cited on page 65.)

[102] RAAB, M., AND STEGER, A. "Balls into Bins" - A simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science RANDOM 1998* (London, UK, 1998), vol. 1518 of *LNCS*, Springer, pp. 159–170. (Cited on page 27.)

[103] SADAKANE, K. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms 48*, 2 (2003), 294–313. (Cited on page 74.)

[104] SALOMON, D. *Data Compression: the Complete Reference*. Springer, New York, NY, USA, 2006. (Cited on pages 6 and 19.)

[105] SALTON, G., YANG, C. S., AND YU, C. T. A theory of term importance in automatic text analysis. *Journal of the American Society for Information Science 26* (1975). (Cited on page 10.)

[106] SANDERS, P., AND TRANSIER, F. Intersection in integer inverted indices. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments ALENEX 2007* (2007), SIAM, pp. 71–83. (Cited on pages 9, 15, 20, 55 and 71.)

[107] SARAIVA, P. C., MOURA, E. S. D., ZIVIANI, N., MEIRA, W., FONSECA, R., AND RIBERIO-NETO, B. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval SIGIR 2001* (New York, NY, USA, 2001), ACM, pp. 51–58. (Cited on page 10.)

[108] SCHOLER, F., E., W. H., YIANNIS, J., AND ZOBEL, J. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM Conference on Research and Development in Information Retrieval SIGIR 2002* (2002), ACM, pp. 222–229. (Cited on page 7.)

[109] SILVERS, C. E. A. Sparsehash. `http://code.google.com/p/google-sparsehash/`, 2009. (Cited on page 33.)

[110] SILVESTRI, F. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on Information Retrieval ECIR 2007* (2007), vol. 4425 of *LNCS*, Springer, pp. 101–112. (Cited on page 6.)

[111] SILVESTRI, F., ORLANDO, S., AND PEREGO, R. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proceedings of the 27th Annual International ACM Conference on Research and Development in Information Retrieval SIGIR 2005* (New York, NY, USA, 2004), ACM, pp. 305–312. (Cited on page 6.)

[112] SPINK, A., WOLFRAM, D., JANSEN, B., AND SARACEVIC, T. Searching the web: The public and their queries. *Journal of the American Society for Information Science 52*, 3 (2001). (Cited on pages 16, 43 and 59.)

[113] STROHMAN, T., AND CROFT, W. B. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval SIGIR 2007* (New York, NY, USA, 2007), ACM Press, pp. 175–182. (Cited on page 9.)

[114] TRANSIER, F., AND SANDERS, P. Compressed inverted indexes for in-memory search engines. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments ALENEX 2008* (2008), SIAM, pp. 3–12. (Cited on pages 29, 30, 44 and 56.)

[115] TRANSIER, F., AND SANDERS, P. Out of the box phrase indexing. In *Proceedings of the 15th Symposium on String Processing and Information Retrieval SPIRE 2008* (2008), vol. 5280 of *LNCS*, Springer, pp. 200–211. (Cited on pages 41, 89 and 90.)

[116] TRANSIER, F., AND SANDERS, P. Compressed storage of documents using inverted indexes, 2009. United States Patent Application 2009-0089256, April 2, 2009. (Cited on page 49.)

[117] TROTMAN, A. Compressing inverted files. *Information Retrieval 6*, 1 (2003), 5–19. (Cited on pages 6 and 17.)

[118] TSEGAY, Y., TURPIN, A., AND ZOBEL, J. Dynamic index pruning for effective caching. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management CIKM 2007* (New York, NY, USA, 2007), ACM, pp. 987–990. (Cited on page 10.)

[119] VENABLES, W. N., AND RIPLEY, B. D. *Modern Applied Statistics with S*, fourth ed. Springer, New York, 2002. ISBN 0-387-95457-0. (Cited on page 65.)

[120] WIKIPEDIA. Truncated binary encoding. `http://en.wikipedia.org/wiki/Truncated_binary_encoding`. (Cited on page 19.)

[121] WILLHALM, T., POPOVICI, N., BOSHMAF, Y., PLATTNER, H., ZEIER, A., AND SCHAFFNER, J. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *The Proceedings of the VLDB Endowment 2*, 1 (2009), 385–394. (Cited on page 115.)

[122] WILLIAMS, H. E., AND ZOBEL, J. Compressing integers for fast file access. *The Computer Journal 42*, 3 (1999), 193–201. (Cited on pages 17 and 18.)

[123] WILLIAMS, H. E., ZOBEL, J., AND ANDERSON, P. What's next? - index structures for efficient phrase querying. In *Proceedings of the 10th Australasian Database Conference ADC 1999* (1999), vol. 21 of *Australian Computer Science Communications*, Springer, pp. 141–152. (Cited on pages 11, 43 and 45.)

[124] WILLIAMS, H. E., ZOBEL, J., AND BAHLE, D. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems 22*, 4 (2004), 573–594. (Cited on pages 10, 11, 43, 44, 59, 90 and 92.)

[125] WILLIAMS, H. E. E. A. The Zettair search engine. `http://www.seg.rmit.edu.au/zettair`, 2004. (Cited on pages 89 and 121.)

[126] WITTEN, I. H., MOFFAT, A., AND BELL, T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999. (Cited on pages 2, 6, 7, 12, 19 and 97.)

[127] XIE, Y., AND O'HALLARON, D. Locality in search engine queries and its implications for caching. In *IEEE Infocom 2002* (2002), pp. 1238–1247. (Cited on page 10.)

[128] YAN, H., DING, S., AND SUEL, T. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web WWW 2009* (New York, NY, USA, 2009), ACM, pp. 401–410. (Cited on page 7.)

[129] YOUNG, R. M. Euler's constant. *The Mathematical Gazette 75* (1991), 187–190. (Cited on pages 31 and 32.)

[130] ZHANG, J., LONG, X., AND SUEL, T. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web WWW 2008* (New York, NY, USA, 2008), ACM, pp. 387–396. (Cited on pages 6, 7 and 10.)

[131] ZIPF, G. K. *Selected Studies of the Principle of Relative Frequency in Language*. Harvard University Press, Cambridge, Massachusetts, 1932. (Cited on page 30.)

[132] ZIPF, G. K. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, Massachusetts, 1949. (Cited on page 30.)

[133] ZOBEL, J., HEINZ, S., AND WILLIAMS, H. E. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters 80*, 6 (2001), 271–277. (Cited on page 115.)

[134] ZOBEL, J., AND MOFFAT, A. Exploring the similarity space. *SIGIR Forum 32*, 1 (1998), 18–34. (Cited on page 115.)

[135] ZOBEL, J., AND MOFFAT, A. Inverted files for text search engines. *ACM Computing Surveys 38*, 2 (2006). (Cited on page 2.)

[136] ZOBEL, J., MOFFAT, A., AND SACKS-DAVIS, R. Storage management for files of dynamic records. In *Proceedings of the 4th Australasian Database Conference* (Brisbane, Australia, 1993), World Scientific, pp. 26–38. (Cited on page 9.)

[137] ZUKOWSKI, M., HEMAN, S., NES, N., AND BONCZ, P. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering ICDE 2006* (Washington, DC, USA, 2006), IEEE Computer Society, p. 59. (Cited on page 6.)