

Index Compression and Redundancy Elimination
in Large Textual Collections

Hao Yan

October 15, 2010

Contents

Vita	iv
Acknowledgements	vi
Abstract	vii
1 Introduction	1
2 Background and Related Work	7
2.1 Inverted Index	7
2.2 Inverted Index Compression	9
2.2.1 General Process	9
2.2.2 Layouts of Inverted Lists	11
2.2.3 State-of-the-Art Index Compression Techniques	11
2.2.4 Evaluation of Compression Techniques	15
2.2.5 Document Reordering and Related Ideas	16
2.3 Query Processing in Search Engines	19
2.3.1 Query Processing without Positions	20
2.3.2 Query Processing with Positions	22
2.4 File Synchronization	23
3 DocID and Frequency Compression with Optimized Document	

Ordering	26
3.1 DocID Compression	27
3.1.1 Distributions of DocIDs	27
3.1.2 Optimizing PForDelta Compression	28
3.1.3 Optimizing Other Methods	31
3.2 Frequency Compression	32
3.2.1 Effect of Reordering on Frequencies	32
3.2.2 New Algorithms	33
3.2.3 Experimental Results on Compression	36
3.3 Query Processing Performance	42
3.4 Mixed-Compression Indexes	46
4 Position Compression	50
4.1 Adaptive Index Compression Methods	51
4.2 Position Gap Distribution	53
4.3 Our Algorithms	55
4.3.1 Page-Adaptive Rice Coding	55
4.3.2 Adaptation with Smoothing	56
4.3.3 Multidimensional Adaptation	57
4.4 Google’s Index Layout for Positions	59
4.5 Experimental Results	61
4.6 Query Processing	67
4.6.1 Position Look-Up Structure	68
4.6.2 Proximity-Aware Scoring	69
5 Algorithms for Low-Latency Remote File Synchronization	71
5.1 Background and Related Work	73
5.1.1 Single Round and Multi-Round Protocols	73

5.1.2	The <i>rsync</i> Algorithm	75
5.1.3	The Set Reconciliation Problem	77
5.1.4	Content-Dependent File Partitioning	79
5.2	An Algorithm Using Set Reconciliation	83
5.2.1	Details on Encoding and Decoding	85
5.2.2	A Variant Based on Golomb Coding	85
5.2.3	Comparison to Previous Work	86
5.2.4	Communication Complexity	87
5.2.5	Preliminary Experimental Results	88
5.2.6	Effect of Similarity on Performance	91
5.3	Integrating Sampling into the Algorithm	93
5.3.1	A More Practical Algorithm	93
5.3.2	A Hybrid Method with FBS	97
5.3.3	Estimating a Good Block Size	98
5.4	A Two-Level Approach	99
6	Conclusions and Future Work	103
6.1	Concluding Remarks	103
6.2	Future Work	105

List of Figures

1.1	A general search engine architecture.	2
2.1	An inverted list for the term <i>soccer</i>	8
2.2	An complete inverted list composed of three separate lists of docIDs, frequencies and positions	10
3.1	Histograms of d-gaps for inverted lists corresponding to 1000 random queries on the TREV GOV2 data set, under three different orderings: <i>original</i> , <i>sorted</i> and <i>random</i> . The x-axis is the number of bits required to represent d-gaps in binary, and the y-axis is the percentage of such d-gaps. (Thus, the first point is for 1-gaps, the second for 2-gaps, the third for 3-gaps plus 4-gaps, and so on. . .	27
3.2	The implementation of the original PFD	28
3.3	The implementation of NewPFD	30
3.4	An example of the look-up table for MLN with $Q = 16$	35

3.5	Compressed size in MB/query versus decompression speed in million integers per second for docIDs, using PFD, NewPFD, and OptPFD under the sorted ordering. The points from left to right for PFD and NewPFD correspond to the following percentages of exceptions: 5%, 8%, 10%, 20%, and 30%. For OptPFD, the points correspond to different target speeds for the optimization and their corresponding sizes.	37
3.6	Compressed size in MB/query for docIDs using twelve methods, under the original, sorted, and random orderings.	38
3.7	Compressed size in MB/query for frequencies using twelve methods, under the original, sorted, and random orderings.	40
3.8	Compressed size in MB/query versus decompression speed in million integers per second for frequencies, using PFD, NewPFD and OptPFD, under sorted ordering.	40
3.9	Total index size in MB versus processing speed per query in milliseconds, for a hybrid index involving OptPFD and IPC. The leftmost point is for pure IPC and the rightmost for pure OptPFD (without MLN or MTF).	49
4.1	Distribution of p-gaps for four words on the TREC GOV2 data set. On the x-axis is the number of bits required to represent the p-gaps in binary, and on the y-axis is the percentage of p-gaps that fall in this range.	53
4.2	Distribution of p-gaps for first, second, and further occurrences, for the four words from Figure 4.1	54
4.3	An example of word locations in a document.	56
4.4	Compressed size per query for various baseline methods on the <i>DS1</i> data set.	62

4.5	Compressed size in MB for various baseline methods on the <i>DS2</i> data set.	63
4.6	Compressed size per query for RPA-RC-S and RPA-BASC-S on the <i>DS1</i> data set.	63
4.7	Compressed size per query for RPA-RC-S and RPA-BASC-S on the <i>DS2</i> data set.	65
4.8	Compressed size per query for RPA-BASC-S, Optb-4D, Optb-5D, Optb-6D, Huff-4D and LLRUN-4D.	66
5.1	Example of the file partitioning technique in [85]. We are hashing windows of size $c = 3$, and then selecting block boundaries based on these hashes for $w = 2$	80
5.2	Comparison of different partitioning techniques on various block sizes for <i>gcc</i> . <i>On the x-axis we show the average block size of the resulting blocks.</i>	82
5.3	Comparison of algorithms on the <i>gcc</i> data set. The graphs from top (worst) to bottom are rsync, FBS, Golomb, reconciliation, and the optimal combination of the last two.	89
5.4	Comparison of algorithms on the <i>html</i> data set. The graphs from top (worst) to bottom are rsync, FBS, Golomb, reconciliation, and the optimal combination of the last two.	89
5.5	Costs of the different data structures transmitted in the reconciliation-based protocol, for <i>gcc</i>	90
5.6	Comparison of Golomb and reconciliation methods for data with varying degrees of similarity. A value of k on the x-axis means that about a $1/2^k$ fraction of the content of each file has been changed (where each changed region has an expected length of 5 characters).	92

5.7	Comparison of different algorithms on <i>emacs</i> data. The graphs from top (worst) to bottom are for rsync, reconciliation, Golomb, the optimal combination, and FBS.	92
5.8	Performance of sampling-based methods on <i>gcc</i>	95
5.9	Costs of various parts of the three algorithms on <i>gcc</i> , for w values of 100, 210, and 300, with resulting average block sizes of 199, 405, and 589, respectively. (Colors in the charts are in the same order.)	96
5.10	A hybrid algorithms with FBS on <i>emacs</i> data.	97
5.11	Using two different block sizes on the <i>gcc</i> data set, without sampling and reconciliation. On the x -axis we show the average size of the larger blocks, and on the y -axis the total communication cost. We look at three different settings of w for determining the smaller blocks, 40, 60, and 80, with resulting average block sizes of 82, 121, and 160, respectively.	100
5.12	Comparison of the Golomb-only method, the sampling-based method, and an idealized method for $w = 60$. Even the idealized method cannot do better than about 450 KB, while the best sampling-based method achieves a cost of about 510 KB.	100
5.13	Using two different block sizes on the <i>emacs</i> data set.	102

List of Tables

3.1	Compressed size in MB/query for docIDs using a basic list-wise IPC (without optimizations), a list-wise version with all other optimizations enabled, and its block-wise version, under the original, sorted, and random orderings.	37
3.2	Compressed size in MB/query for frequencies, under the original, sorted, and random orderings, using IPC, IPC with MTF, and IPC with MLN, for list- and block-oriented methods.	41
3.3	Compressed size in MB/query for frequencies under sorted document ordering.	41
3.4	Decoding speeds in millions of integers decoded per second, for docIDs, frequencies, and frequencies with MLN transformation.	43
3.5	Compressed index size in MB for the entire TREC GOV2 data set, for IPC, NewPFD with 10% threshold on exceptions, and OptPFD optimized for minimal index size. For the sorted case, we provide numbers for frequencies and total index sizes with and without MLN.	44
3.6	Running time and number of decoded docIDs and frequencies for OptPFD on the GOV2 data set.	45
3.7	Running times in ms per query for IPC (with MLN), NewPFD, and OptPFD.	46
4.1	Compressed sizes (MB) for the <i>DS2</i> data set using selected compression methods under Google’s layout.	64
4.2	Compressed sizes (MB/q) for selected methods.	67

4.3	Space overhead and performance of the lookup structure for $K = 100$.	69
4.4	Percent of queries that achieve the same top- m results as an exhaustive evaluation.	70
4.5	Percent of correct top- m results returned.	70

Chapter 1

Introduction

Web search engines are designed to search for information, which may consist of web pages, images, videos and other types of files, on the World Wide Web. Web users type a few words into the query box of a search engine, for example, Google, and the search engine returns a list of URLs, usually in a sorted order in terms of their relevance to the queries users submit. During the past almost twenty years, the dramatic development of web search engines has made them one of the most popular web tools and has changed the way people share and search information. In the United States, the most popular search engines, Google, Yahoo!, and Bing, are being used by millions of users every day.

Due to the rapid growth in the size of the web and the number of web users, search engines are faced with significant performance challenges. Current commercial search engines already have to process thousands of queries per second on billions of documents, and the total number of queries issued is still increasing every year. In addition, users expect higher and higher result quality even in the presence of spam and other manipulation, requiring constant tuning of the system. There are many known techniques to improve search engine performance. This thesis will focus on some of them.

Before we describe these techniques, we first need to understand the basic

architecture of a web search engine, that is, the necessary components of a search engine and how those components are put together to obtain an effective search tool. The general search engine architecture is shown in Figure 1.1.

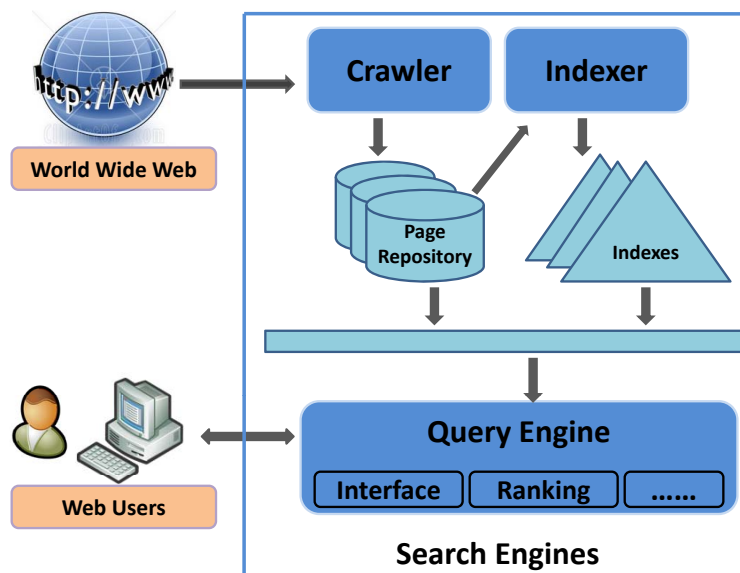


Figure 1.1: A general search engine architecture.

From Figure 1.1, we can see that the main functions of a search engine are as follows: the search engine crawls web pages from the World Wide Web and constructs an index structure on the crawled pages. Once web users enter queries in web browsers, the search engine processes the queries based on the indexes, and finally returns the query results to the users. In order to achieve these functions, a search engine has to implement at least the following three key modules: crawler, indexer, and query engine. Please note that Figure 1.1 only shows a minimum set of necessary components for a search engine. However, the techniques studied in this thesis, which are underlying such an architecture, can be easily applied to a much more realistic search engine architecture.

Thus, the crawler downloads web pages and stores them in a large page repository that is then indexed later for efficient query processing. It traverses the

web by automatically following hyperlinks in web pages to reach other pages in a breath-first manner. Normally, a crawler is first given a starting set of URLs, through which the corresponding pages can be directly retrieved from the web. It then extracts the URLs appearing in the retrieved pages to compose a larger set of URLs, from which more pages can be retrieved in a similar manner. The above process is performed recursively until a satisfactory amount of pages has been crawled. Please note that in reality crawlers may also use other policies to traverse web pages, for example, crawling more important pages first. A highly efficient crawler is crucial to search engine performance, since millions of new web pages are created daily and need to be downloaded by the engines to provide web users with up-to-date information. In addition, after pages are initially crawled, they have to be periodically recrawled and checked for updates. The most attractive pages are often crawled or refreshed with higher priorities than other pages. Therefore, a crawler not only needs to download a particular set of pages as fast as possible, but it also must be able to decide which page to download next intelligently, such that the most useful pages are obtained and kept up-to-date.

The indexer module of a search engine extracts the words (terms) from each web page in the page repository (document collection) and builds an index structure, called an inverted index. The inverted index is a set of inverted lists, each of which stores for a particular term information about all locations where the term occurs in the collection. The inverted index can be seen as a huge look-up table for the later query processing.

The query engine processes user queries based on the index and returns to users ranked query results. In particular, it first parses a query into terms and loads the inverted lists of these terms from disk to memory. It then traverses these lists to find the documents that contain all of these terms and at the same time calculates a document score for each of the document. Finally, the query

engine will rank these documents according to their scores and return to the users a list of result snippets in descending order of their rankings. Please note that the above score calculation is often very complex and depends on a variety of features derived from many other information sources, e.g., web data statistics or query logs. However, we will not describe these data mining or machine learning techniques since they are largely orthogonal to the techniques we study in this thesis.

Another important feature, not shown in Figure 1.1, is that a search engine is essentially a large scale distributed computing system. Therefore, each of the above modules is implemented on clusters of hundreds to thousands of machines. This distributed system architecture can fundamentally improve the query processing speed and robustness of the search engine.

There have been significant efforts to improve the performance of each module of the search engine under the architecture in Figure 1.1. For example, for the crawler, techniques to detect near duplicate pages are used to avoid repeatedly downloading the same pages. For the indexer, index compression techniques can greatly reduce the index size. More importantly they can reduce disk traffic significantly and thus speed up query processing. For the query engine, caching techniques [9–11, 37, 52, 54, 91, 94] can speed up query evaluation in the following two different ways: First, we can cache the query results for the most frequently used queries, such that results can be returned quickly without executing the same queries repeatedly. Additionally, we can cache the inverted lists of the most frequently used query terms. Thus, even if there is a cache miss for the entire query, we do not need to load inverted lists for all query terms from disk to memory. This can reduce disk or network transfers significantly. Query engines also often use early termination techniques [3, 7, 36, 53, 67, 73, 93, 96] to save query processing time, since such techniques can avoid processing all documents in the

relevant inverted lists, instead evaluating only a small subset of them.

In this thesis, we mainly focus on inverted index compression techniques. In particular, we study techniques to compress docIDs, frequencies and positions. As discussed above, one of the most important advantages of improved index compression techniques is to reduce I/O traffic and thus speed up query processing.

We also study another related topic in this thesis, file synchronization. The file synchronization problem is to update the old version of a file, located on one machine, with the new version of it, located on another machine, at a minimum communication cost over a network. The goal in file synchronization techniques is to ideally only transfer the difference between versions instead of the whole new version. In the case where two versions are very similar, a large amount of network traffic can be avoided. File synchronization techniques can be used in many applications, for example, synchronization of user files between different machines, distributed file systems, remote backups, mirroring of large web and ftp sites, content distribution networks.

File synchronization techniques are also very useful for a search engine. For both efficiency and safety issues, the data of a search engine is often stored and replicated across multiple data centers that may be located in different cities or even in different countries. Periodically, the search engine recrawls new versions of millions of web pages, and then needs to synchronize the old versions located on other machines. In such cases, using file synchronization techniques can save a huge amount of data transfer over networks, especially when the outdated versions of pages are similar to the ones that are lately recrawled.

In this thesis, we propose a new file synchronization algorithm based on *set reconciliation* techniques [57], and show that the communication cost of our algorithm is often significantly smaller than that of *rsync*, which is a widely used open source tool for file synchronization.

In summary, search engines often need to transfer huge amounts of data from I/O devices or over networks, which may significantly affect their efficiency. In order to overcome such problems, in this thesis we study inverted index compression techniques and file synchronization techniques. Our goal is to exploit the similarities within web pages, and also between different web pages, in order to reduce the data redundancy and thus improve search engine efficiency.

The remainder of the thesis is organized as follows. In the next chapter, we provide some technical background and discuss related work. In Chapter 3, we study techniques for docID and frequency compression, while in Chapter 4 we study those for position compression. We then discuss file synchronization techniques in Chapter 5, and finally provide concluding remarks in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we first describe technical background and related work for inverted index compression techniques. We then provide background on file synchronization techniques, with more details discussed in Chapter 5.

2.1 Inverted Index

Web search engines as well as many other IR systems are based on an *inverted index*, which is a simple and efficient data structure that allows us to find all documents that contain a particular term. An *inverted index* I for the collection consists of a set of *inverted lists* $I_{w_0}, \dots, I_{w_{m-1}}$ where list I_w contains a sequence of postings for each document containing the word w . Each posting contains the ID of the document where the word occurs (docID), the number of occurrences in this document (frequency), and the positions of the occurrences within the document expressed as the number of words preceding the occurrence (positions). In particular, a posting for a word w is of the form $(d_i, f_i, p_{i,0}, \dots, p_{i,f-1})$ [90], indicating w appears f_i times in document d_i and $p_{i,j} = k$ iff w is the k -th word in document d_i . The postings in each inverted list are usually sorted by docID and stored in highly compressed form on disk. An inverted list for the term *soccer* is

shown in Figure 2.1.

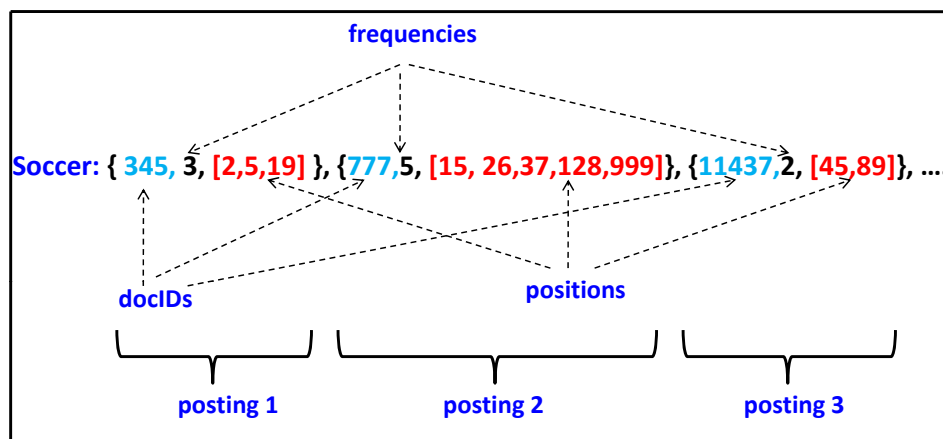


Figure 2.1: An inverted list for the term *soccer*

DocIDs, frequencies, and positions have their own unique properties and are normally compressed separately using different methods (more details will be discussed later). For the simplicity of our later discussion of various compression techniques, we can treat an inverted list as three separate lists of docIDs, frequencies, and positions. That is, the sequence of postings $(d_i, f_i, p_{i,0}, \dots, p_{i,f_i-1})$ can be considered as three sequences of docIDs, $(d_1, d_2, d_3, \dots, d_i, \dots, d_n)$, frequencies, $(f_1, f_2, f_3, \dots, f_i, \dots, f_n)$, and positions, $([p_{1,0}, p_{1,1}, \dots, p_{1,f_1}], [p_{2,0}, p_{2,1}, \dots, p_{2,f_2}], \dots, [p_{i,0}, p_{i,1}, \dots, p_{i,f_i}])$. Each of the lists is sorted in the same order, that is, the i th item on the docID list, d_i , corresponds to the i th item f_i on the frequency list and also to the i th item on the position list, $[p_{i,0}, p_{i,1}, \dots, p_{i,f_i}]$. They can be easily combined to comprise the i th posting of $(d_i, f_i, [p_{i,0}, p_{i,1}, \dots, p_{i,f_i}])$. Please note that this is just for the simplicity of our later discussion of various compression techniques. The real layouts of inverted lists can be of various forms and will be discussed later.

2.2 Inverted Index Compression

2.2.1 General Process

Many inverted index compression techniques have been proposed in the literature [90]. In most techniques, the lists of docIDs are often first preprocessed by taking the differences (called d-gaps) between two consecutive docIDs, while the list of positions are preprocessed by taking the differences between the values of any two consecutive positions (called p-gaps) in the same posting. The resulting d-gaps and p-gaps are then encoded together with the frequencies using some integer compression algorithm. We can easily achieve d-gaps since they are in sorted order in a list, while we can achieve p-gaps only for the positions within the same document (posting), since they are in sorted order only within each document but not across documents (postings) on a list. More precisely, we replace each docID d_i with $i > 0$ by $d_i - d_{i-1} - 1$, each f_i by $f_i - 1$ (since no posting can have a frequency of 0), and each $p_{i,j}$ with $j > 0$ by $p_{i,j} - p_{i,j-1} - 1$.

Using d-gaps and p-gaps instead of original docIDs and positions decreases the average value that needs to be compressed, resulting in better compression. Of course, these values have to be summed up again during decompression, but this can usually be done very efficiently. Unlike docIDs and positions, for frequencies, we cannot use gaps since they are usually not in sorted order.

One slight problem with this approach (compressing gaps instead of original values) is that in order to compress a particular posting, we would have to decompress all preceding postings and add up their values. To avoid this problem, an inverted list is often partitioned into blocks say, $m=128$ postings before it is compressed. For each block, we store one uncompressed docID and one additional pointer, allowing the query processor to independently decompress any block. The docID specifies the maximum value of all docIDs in the current block, and the

pointer points to the beginning of the next block. During query processing, we can use these uncompressed docIDs to decide if a particular docID is possibly in the current block. If it is not so, we can use the pointer to skip the entire block without decompressing any information in it, resulting in faster query processing.

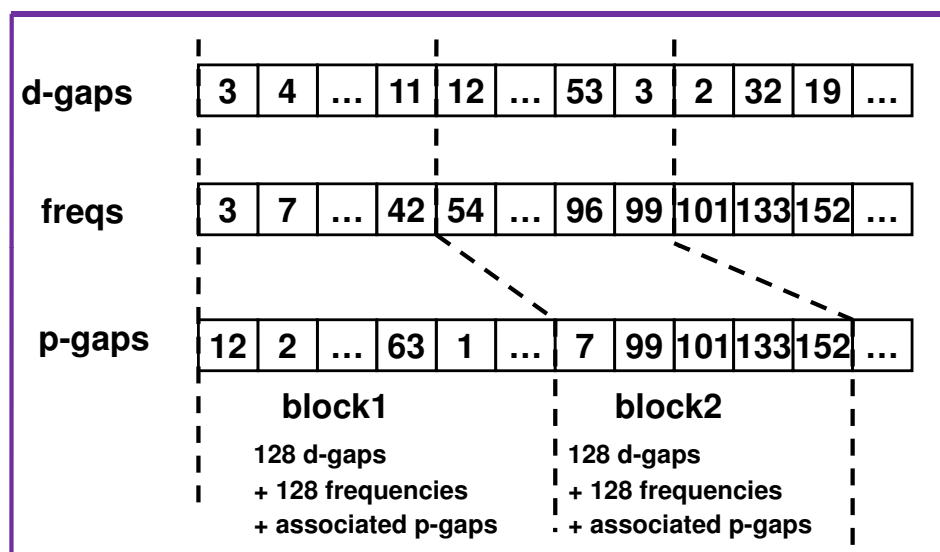


Figure 2.2: An complete inverted list composed of three separate lists of docIDs, frequencies and positions

After the above preprocessing step, we get for each term a list of d-gaps, a list of frequencies, and a list of p-gaps, shown in Figure 2.2, each of which is a sequence of blocks of non-negative integers. Thus, inverted index compression techniques are concerned with compressing sequences of non-negative integers.

Interestingly, the integers in the sequence are usually not independent of each other. In particular, occurrences of words (terms) in text tend to be clustered (or locally homogeneous [59]) rather than spread out uniformly. This clustering property can be exploited to improve the position compression. Thus, we have the problem of compressing sequences of integer values in certain contexts. The resulting compression ratio depend on the exact properties of these sequences, which may depend on the way in which docIDs are assigned to documents and the

way the articles (web pages) are written. As a result, index compression methods can be roughly divided into the following two categories: context-free methods, which compress each number in the sequence independently, and context-sensitive methods, which compress the sequence by additionally taking into account context information, such as the value of the previous docID (or frequency or position), or the document size.

2.2.2 Layouts of Inverted Lists

There are several possible layouts of inverted lists. Two kinds of them have been mentioned above, the first layout in Figure 2.1 and the second one in Figure 2.2 treating an inverted list as three separate lists of docIDs, frequencies and positions. Another possible layout is to concatenate the three separate lists together sequentially, that is, a sequence of docIDs followed by a sequence of frequencies followed by a sequence of positions. Or we can first partition each list (in the second layout) into blocks, and then concatenate the blocks in a way such that the inverted list of a term is a sequence of blocks, each of which contains 128 docIDs followed by 128 frequencies followed by their corresponding positions.

In this thesis, we separately discuss compression techniques for docIDs, frequencies, and positions. Therefore we focus on the second layout with blocks. That is, we store for each term three separate lists of docIDs, frequencies, and their corresponding positions. Each list is partitioned into blocks with size 128. However, we still refer to a posting as $(d_i, f_i, p_{i,0}, \dots, p_{i,f-1})$, although unlike in the first layout in Figure 2.1, d_i , f_i and $p_{i,0}, \dots, p_{i,f-1}$ are stored separately.

2.2.3 State-of-the-Art Index Compression Techniques

There are many different inverted index compression techniques. For context-free approaches [90], this includes gamma and delta coding [34], Golomb and Rice

coding [90,97], and variable-byte coding (var-byte) [75,89]. For context-sensitive methods, there are Simple9 (S9) [5] and its variants [2, 4, 6, 94], LLRUN [38] and its variants [62], PForDelta [98] and its variants [92], compression techniques using 64-bit words [8], HMM (Hidden Markov Model)-based methods [20, 21], interpolative coding [60,61], and BASC and RBUC [59] and many others. We provide brief outlines of some of the above methods; for more details, please see the cited literature.

Gamma Coding: Gamma coding [34] represents a value $n \geq 0$ by a unary code for $1 + \lfloor \log(n + 1) \rfloor$ followed by a binary code for the lower $\lfloor \log(n + 1) \rfloor$ bits of n . Gamma coding is good for compressing small numbers but relatively inefficient for large numbers.

Golomb Coding and Rice Coding: In Golomb coding [40,90] an integer n is encoded in two parts: a quotient q stored as a unary code, and a remainder r in binary form. To encode a set of integers, we first choose a parameter B ; a good choice is $B = 0.69 \times ave$, where *ave* is the average of the values to be coded. Then for each number n we compute $q = \lfloor n/B \rfloor$ and $r = n \bmod B$. If B is a power of two, then $\log(B)$ bits are used to store the remainder r ; otherwise, either $\lfloor \log(B) \rfloor$ or $\lceil \log(B) \rceil$ bits are used depending on r . Rice coding is the case where B is always chosen as a power of two. This allows for a more efficient implementation through use of bit shifts and masks, while the difference in size is usually small.

Var-Byte Coding: Variable-byte (var-byte) coding [75,89] represents an integer n as a sequence of bytes. In each byte, we use the lower 7 bits to store a part of the binary representation of n , and the highest bit as a flag to indicate if the next byte is still part of the current number. Thus, $142 = 1 \cdot 2^7 + 16$ is represented as 10000001 0001000, while 2 is represented as 00000010. Variable-byte coding is simple to implement and known to be significantly faster than traditional bit-

wise methods such as Golomb, Rice, Gamma, and Delta coding [97]. However, it usually does not achieve the same reduction in index size as bit-wise methods.

S9 and S16: Simple9 (S9) coding [5] is not byte-aligned, but can be seen as combining word alignment and bit alignment. The basic idea is to try to pack as many integers as possible into one 32-bit word. Simple9 divides each word into 4 status bits and 28 data bits, where the data bits can be divided up in 9 different ways and the status bits are used to indicate which of the 9 ways was used. For example, if the next 7 values are all less than 16, then we can store them as 7 4-bit values. Or if the next 3 values are less than 512, we can store them as 3 9-bit values (leaving one data bit unused). Simple16 (S16) is a variation of Simple9 that uses 16 instead of 9 cases and thus achieves a slightly better use of each 32-bit word [94]. The Carryover-12 [5] and the Slide methods [6] are two other extensions of S9 to save its wasted bits.

PForDelta: PForDelta is a recent technique proposed in [43,98] for compression in database and IR systems. The basic idea is to split a list into chunks of some fixed size, and then select a value b such that most of the values in the current chunk (say, 90%) are less than 2^b and thus fit into a fixed bit field of b bits each. The remaining integers, called exceptions, are coded separately. Variants of PForDelta have been shown to outperform variable-byte coding in terms of both speed and compression. For best results, chunks should contain a number of values that is a multiple of 32; this guarantees that the bit fields of each chunk align with word boundaries.

LLRUN: LLRUN [38] uses Huffman coding [90] instead of unary coding for the unary parts of the Gamma code. The Huffman code is derived from statistics over the entire collection. Thus, LLRUN splits the space of integer values into intervals or buckets such that each number can be represented by its bucket number k and its offset o within the bucket. We note that this idea is also adopted in the widely

used zlib library [39], where the binary part is referred to as “extra bits”. LLRUN was improved in [62] by using a separate Huffman table for each inverted list.

Interpolative Coding: Interpolative coding (IPC) is a coding technique proposed in [61] that is ideal for the types of clustered or bursty term occurrences that exist in real large texts (such as books). The basic idea of interpolative coding [61] is to first encode the docID in the middle of the list, represented by the gap from the start of the list, and then recursively compress the left and right halves of the list. This might seem counterintuitive at first glance, but if occurrences are heavily clustered, then this divide-and-conquer approach will eventually focus on fairly small regions of the collection that contain many occurrences of a term and can thus be encoded very succinctly. In fact, IPC was shown to perform very well for clustered data in [61].

IPC differs from the other methods in an important way: It directly compresses docIDs, and not docID gaps. Given a set of docIDs $d_i < d_{i+1} < \dots < d_j$ where $l < d_i$ and $d_j < r$ for some bounding values l and r known to the decoder, we first encode d_m where $m = (i + j)/2$, then recursively compress the docIDs d_i, \dots, d_{m-1} using l and d_m as bounding values, and then recursively compress d_{m+1}, \dots, d_j using d_m and r as bounding values. Thus, we compress the docID in the center, and then recursively the left and right half of the sequence. To encode d_m , observe that $d_m > l + m - i$ (since there are $m - i$ values d_i, \dots, d_{m-1} between it and l) and $d_m < r - (j - m)$ (since there are $j - m$ values d_{m+1}, \dots, d_j between it and r). Thus, it suffices to encode an integer in the range $[0, x]$, where $x = r - l - j + i - 2$, that is then added to $l + m - i + 1$ during decoding; this can be done trivially in at most $\lceil \log_2(x + 1) \rceil$ bits, since the decoder knows the value of x .

In areas of an inverted list where there are many documents that contain the term, the value x will be much smaller than $r - l$. As a special case, if we have to

encode k docIDs larger than l and less than r where $k = r - l - 1$, then nothing needs to be stored at all as we know that all docIDs properly between l and r contain the term. This also means that IPC can use less than one bit per value for very dense term occurrences.

RBUC and BASC: Moffat and Anh proposed two binary codes [59], RBUC and BASC, for compressing locally homogenous sequences. RBUC encodes the next s numbers into s b -bit binary codes, where the shared b , called selector, is the number of bits of the binary code for the maximum value of those s numbers. RBUC can be applied recursively to the resulting sequence of selectors, and s can be reduced at each recursive level by using different escalation functions. For example, the s in the next level can be computed as $f(s) = 2 * s$ or $f(s) = s * s$. BASC is an on-line method that predicts the number of bits b_i used to encode the next number x_i by using the value of b_{i-1} . In particular, it first uses one bit to indicate if $b_i < b_{i-1}$, and then encodes x_i as a b_{i-1} -bit binary code if that is true, and otherwise as a $(b_i - b_{i-1})$ -bit unary code followed by $(b_i - 1)$ -bit binary code. A variant of BASC is BASC-smooth, which predicts b_i by exploiting the average b -value used for k previous numbers.

2.2.4 Evaluation of Compression Techniques

Index compression techniques are usually evaluated in terms of the following measures: (1) The compression ratio, which determines the amount of main memory needed for a memory-based index or the amount of disk traffic for a disk-based index. State-of-the-art systems typically achieve compression ratios of about 3 to 10 versus the naive 32-bit representation, while allowing extremely fast decompression during inverted list traversals. (2) The decompression speed, typically hundreds of millions of integers per second, which is crucial for query throughput. In contrast, compression speed is somewhat less critical, since each inverted list is

compressed only once during index building, and then decompressed many times during query processing.

We note that there are two different ways to evaluate the compression ratio. We can consider the total size of the index; this models the amount of space needed on disk, and also the amount of main memory needed if the index is held entirely in main memory during query processing. Alternatively, we can measure the compressed size of the inverted lists associated with an average query under some query load; this models the amount of data that has to be transferred from disk for each query if the index is entirely on disk (and also the amount of data that has to be moved from main memory to CPU as this can become a bottleneck in highly optimized systems). In reality, many systems cache some part of the index in memory, making a proper evaluation more complicated. We consider both cases in our experiments, but find that the relative ordering of the algorithms usually stays the same.

2.2.5 Document Reordering and Related Ideas

As we mentioned before, inverted index compression techniques compress sequences of integers. The compression ratio depends on the exact properties of these sequences, which depend on the way in which docIDs are assigned to documents. This has motivated several authors [16–18, 32, 77–80] to study how to assign docIDs in a way that optimizes compression. The basic idea here is to assign docIDs such that many similar documents (i.e., documents that share a lot of content or at least many terms) are close to each other in the docID assignment. In this case the resulting sequence of d-gaps will become more clustered, with large clusters of many small values interrupted by a few larger values (We will see in later chapters that the compression of frequencies can also be improved through reordering techniques). These techniques can be roughly divided into two

categories, the top-down and bottom-up methods.

The top-down approaches break down the collection into increasingly smaller clusters, within each of which documents are similar to each other, and then docIDs are sequentially assigned within each cluster. The algorithm in [18] has been proven to achieve better performance than other top-down methods and than many other approaches in the bottom-up class [17, 80]. However, it can only deal with very small data sets and is therefore not suitable for large search engines.

The bottom-up approaches create a dense graph, where each vertex represents a document while each edge is associated with a weight that indicates the similarity between the two documents connected by the edge. An example of such a weight is the number of shared terms between two documents [17, 77]. The approaches then try to maximize the total path similarity by traversing the edges. In particular, to achieve this goal, several approaches are studied in [77], based on the algorithms of the Maximum Spanning Tree (MST) and Traveling Salesman Problem (TSP) respectively. Although the methods in [17] further reduce the computational effort through SVD for dimensionality reduction, the time complexity is still quadratic in the number of documents. Very recently, a new framework is proposed in [32] and is based on performing a TSP-like computation on a reduced sparse graph obtained through Locality Sensitive Hashing. Although TSP-based methods can achieve better performance than other bottom-up approaches, most of them are also limited to a data set size with at most a few hundred thousand documents (except [32]).

In contrast to the above two categories of methods, a method in [78] looked at a much simpler approach, simply sorting all documents in the collection alphabetically according to their URLs and then assigning docIDs sequentially. It was shown in [78] that this method can reduce the inverted index size greatly and basically match the performance of previous techniques based on text clustering.

More importantly, compared to the above top-down and bottom-up methods, this method is much more scalable and therefore suitable for even large-scale search engines. Note that such an alphabetical ordering places all documents from the same site, and same subdirectory within a site, next to each other. This results in improved compression as such documents tend to have the same topic and use the same writing style. However, the sorting-by-URL method may not be applicable to certain types of collections, where URLs are not available or where URL similarity does not necessarily indicate document similarity, for example, web pages in Wikipedia.

In this thesis, we follow the document reordering approach studied in [16–18, 77–80]. However, while previous work has focused on finding the best ordering of documents in a collection, we focus on the next step, how to optimize actual index compression and query processing given some suitable document ordering obtained from previous work. For example, previous work considered only a few standard techniques for docID compression, and did not consider frequency compression or query processing efficiency.

Another related problem is the compression of inverted indexes for archival collections, i.e., collections that contain different versions of documents over a period of time, with often only minor changes between versions. This problem has recently received some attention in the research community [14, 25, 42, 44, 95], and the basic idea is also to exploit similarity between documents (or their versions). The techniques used are different, and more geared towards getting very large benefits for collections with multiple very similar versions, as opposed to the reordering approach here which tries to exploit more moderate levels of similarity.

Feasibility of Document Reordering: In practice, IR systems may assign docIDs to documents in a number of ways, e.g., at random, in the order they are crawled or indexed, or sometimes based on global measures of page quality

(such as Pagerank [22]) that can enable faster query processing through early termination. The document reordering approach in this thesis and the previous work in [16–18, 77–80] assumes that we can modify this assignment of docIDs to optimize compression. While this is a reasonable assumption for some systems, there are other cases where this is difficult or infeasible. We now discuss two cases, distributed index structures, and tiering and early termination techniques.

Large-scale search engines typically partition their document collection over hundreds of nodes and then build a separate index on each node. If the assignment of documents to nodes is done at random, then a local reordering of documents within a node might not give much benefit. On the other hand, if pages are assigned to nodes based on a host-level assignment or alphabetical range-partitioning, then we would expect significant benefits. However, this might require changes in the architecture and could impact issues such as load balancing.

Document ordering is also complicated by the presence of tiering and other early termination mechanisms, which are widely used in current engines. In a nutshell, these are techniques that avoid a full traversal of the inverted lists for most queries through careful index layout, which often involves some reordering of the documents. In some approaches, such as a document-based tiering approach [72], or a partitioning of inverted lists into a small number of chunks [53, 67], reordering for better compression can be applied within each tier or chunk. Other approaches may assign docIDs based on Pagerank [22] or other global document scores mined from the collection [71], or use a different ordering for each list [36]; in these cases our approach may not apply.

2.3 Query Processing in Search Engines

Query processing in state-of-the-art systems involves a number of phases such as query parsing, query rewriting, and the computation of complex, often machine-

learned, ranking functions that may use hundred of features. However, at the lower layer, all such systems rely on extremely fast access to an inverted index to achieve the required query throughput. In particular, for each query the engine typically needs to traverse the inverted lists corresponding to the query terms in order to identify a limited set of promising documents that can then be more fully scored in a subsequent phase. The challenge in this initial filtering phase is that for large collections, the inverted lists for many commonly queried terms can get very long. For example, for the TREC GOV2 collection of 25.2 million web pages used in this thesis, on average each query involves lists with several million postings. In this section, we first discuss the general query process without considering the position information, and then discuss the case when positions are taken into account.

2.3.1 Query Processing without Positions

Current systems typically use a style of query processing called *document-at-a-time* (DAAT) query processing [24, 49], where all inverted lists associated with a query are opened for reading and then traversed in parallel. We typically sort all associated lists of a query by their lengths in ascending order and always first access the shortest list and then try to find matching docIDs in other longer lists. In order to look up the matching docIDs in an efficient way, we keep one pointer p_i for each of the inverted lists L_i related to the query, where i is from 1 to n and n is the number of terms in the query, and move these pointers forward in a synchronized manner as follows: (Initially all pointers point to the first docIDs of their lists.)

- (1) We first check if the pointer p_1 of the shortest list L_1 has reached the end (that is, is past the last docID) of L_1 . If it is, then the program exits; otherwise we set $i = 2$ and go to Step (2).
- (2) We advance the pointer p_i of the next shortest list L_i from left to right until

we find a docID $docID_{p_i}$ that is equal to or greater than the docID $docID_{p_1}$ pointed by p_1 . If p_i reaches the end (that is, such a docID is not found), the DAAT is finished and the program exits.

- (a) If $docID_{p_i} > docID_{p_1}$, we record the value of i as $K = i$ and then go to Step (3).
- (b) Otherwise ($docID_{p_1} == docID_{p_2}$), if $i < n$, we set $i = i + 1$ and go back to Step (2) for the next shortest list; otherwise ($i == n$) we find a matching docID for all of the query terms and we calculate its score. In the latter case, we then move p_1 one step further and go to Step (1).
- (3) We advance the pointer p_1 of the shortest list until $docID_{p_1} \geq K$ or p_1 reaches the end of L_1 . We then go back to Step (1).

This DAAT approach has several advantages: (a) it performs extremely well on the AND and WAND [24] style queries common in search engines, (b) it enables a very simple and efficient interface between query processing and the lower-level access and index decompression mechanisms, and (c) it allows for additional performance gains through forward skips in the inverted lists, assuming that the postings in each list are organized into blocks of some small size that can be independently decompressed.

In our experiments, we use an optimized DAAT query processor developed in our group, and we organize each inverted list into blocks with a fixed number of postings. We choose 128 postings as our default block size (shown to perform well, e.g., in [94]), and keep for each inverted list two separate arrays containing the last docID and size of each block in words in (almost) uncompressed form. This allows skipping of blocks during query processing by searching in the array of last docIDs. All decompression is performed in terms of blocks; to add another compression method to our query processor, it suffices to supply one method for

uncompressing the docIDs of a block, and one to uncompress the frequencies. (A block consists of all 128 docIDs followed by all 128 frequency values.) This design is highly useful in the later experiments, where we use several different compression techniques within the same index.

2.3.2 Query Processing with Positions

We now discuss how search engines access and use the position data stored in the index. There are two main uses of position data. First, positions are used for queries containing phrases, either specified by the user or created by the search engine through query analysis and reordering. For example, the engine might recognize that a query contains a person name, and rewrite the query into a new query that requires the first and last name to be in close proximity in the text. Thus, positions are used as a filter. Second, positions can be used in ranking functions to improve result quality. The idea here is that a document containing the search terms in close proximity is more likely to be relevant to the user than a document where the terms occur in completely different parts of the document. Several researchers [26, 55, 58, 73, 84] have recently proposed ranking functions that use proximity to improve result quality on TREC collections and tasks. In practice, web search engines tend to use machine learning to find good ranking functions, and term positions or proximity are important features among the hundreds used.

We focus in this thesis on the second use, where positions are considered by the ranking function; the first use typically only applies to very specific subset of the queries. There appears to be little published work on how to optimize query processing with position data. One challenge is that the position data in the index is usually several times (3 to 5 times) larger than the docID and frequency data, and thus a naive use of positions could significantly decrease system throughput.

To avoid this, query processing can be performed in two stages. First, a simple ranking function requiring docIDs and frequencies only (e.g., BM25 or similar) is applied. In the second stage, position data is fetched only for a small subset of documents (a few hundred or thousand) that scored very high on the simple ranking function. This changes the trade-off between compressed size and decompression speed somewhat compared to the case of docIDs, as we only decompress a limited number of positions. In fact, as we show later, the CPU cost of this second phase can be much smaller than that of the first phase, even with fairly slow position decompression methods. On the other hand, the compressed size of the position data has a very significant impact on system cost. In the case of a memory-resident index, smaller compressed size means less memory is needed. In the case of a primarily disk-resident index, disk transfers are reduced significantly due to reduced list sizes and higher cache hit rates, since a larger percentage of the total index data can be cached in main memory [94]. (Note that even if only some of the position data has to be fetched from each list, disk access costs will usually be equal to that of fetching the complete lists, as random lookups are prohibitively expensive on current disks.)

In summary, access to position data may be performed in a second stage after traversing the lists of docIDs, only a limited amount of position data is retrieved, and a small compressed size may be more important than extremely fast access to positions.

2.4 File Synchronization

In this section, we only briefly describe the file synchronization problem and state-of-the-art file synchronization techniques. More details are provided in Chapter 5.

Remote file synchronization can be informally described as follows: Assume

that we have two versions of each file, an outdated file f_{old} held by one machine, the *client*, and a current file f_{new} held by another machine, the *server*. Further, assume that neither machine knows the exact content of the file held by the other machine. Periodically, the client may initiate a synchronization operation that updates its file f_{old} to the current version f_{new} . Or alternatively, a synchronization operation is performed whenever the client accesses f_{old} . If the amount of data is large, or the network fairly slow, then it is desirable to perform this synchronization with a minimum amount of communication.

As we discussed in Chapter 1, file synchronization arises in a number of applications, such as synchronization of user files between different machines, distributed file systems, remote backups, mirroring of large web and ftp sites, content distribution networks, or distributed crawlers of search engines. In many cases, updated files differ only slightly from their previous version; for example, updated web pages often change only in a few places. In such cases, it would be desirable to update files without transmitting the redundant data.

Rsync is an open-source tool for synchronization of file systems across machines [86]. The basic idea in *rsync* is to split a file into blocks and use hash functions to compute hashes or “fingerprints” of the blocks. These hashes are sent to the other machine, where the recipient attempts to find matching blocks in its own file, and then asks for the sender to send it only the unmatched blocks. From the process we can see that *rsync* is essentially a distributed delta compressor, where one machine compresses its target file f_{new} against the reference file f_{old} on the other machine as a delta (or patch) file, i.e., the difference between f_{old} and f_{new} , and then transfers the delta to the machine on the other side.

From the angle of network protocols, *rsync* is a single round protocol (strictly speaking, it involves one and a half rounds), and a fairly simple protocol with low computing and I/O overhead. In contrast, many multi-round protocols [28, 29, 35,

51, 64, 66, 76, 83] have been shown to provide significant bandwidth savings over the single-round case. However, they often suffer from protocol complexity and high computing and I/O overheads.

This motivates the search for single-round protocols that transmit significantly less data than *rsync* while preserving its main advantages. In Chapter 5, we propose a new single-round algorithm for file synchronization based on the use of *set reconciliation* [57] and sampling techniques. The communication cost of our algorithm is often significantly smaller than that of *rsync*, especially for very similar files.

Chapter 3

DocID and Frequency Compression with Optimized Document Ordering

In this chapter, we only focus on docID and frequency compression and will discuss position compression in the next chapter. As mentioned in the last section, we follow the document reordering approach studied in [16, 18, 77–79]. However, while previous work has focused on finding the best ordering of documents in a collection, we focus on the next step, how to optimize actual index compression and query processing given some suitable document ordering obtained from previous work. In particular, we extensively study and optimize state-of-the-art compression techniques for docIDs, and propose new algorithms for compressing frequencies, under such optimized orderings. Frequency values tend to be small compared to docID gaps (on average when a word occurs in a web page it occurs only 3 to 4 times), and thus different techniques are needed to improve their compression. We further study the impact of docID reordering on query throughput, and propose and study a new index optimization problem motivated by the trade-off between speed and compression ratio of the various methods. Overall, our experimental results show very significant improvements in both overall index size and query processing speed in realistic settings.

3.1 DocID Compression

In this section, we perform a detailed study of compression techniques for docIDs. In particular, we first study distributions of docIDs on TREC GOV2 data set (the data set will be explained later in the experimental results), and then discuss state-of-the-art compression methods and propose our new algorithms.

3.1.1 Distributions of DocIDs

The performance of a compression method depends on the data distribution it is applied to. For inverted index compression, compression is best when there are many small numbers. The optimized assignment of docIDs is intended to increase the number of small numbers and thus improve compression performance.

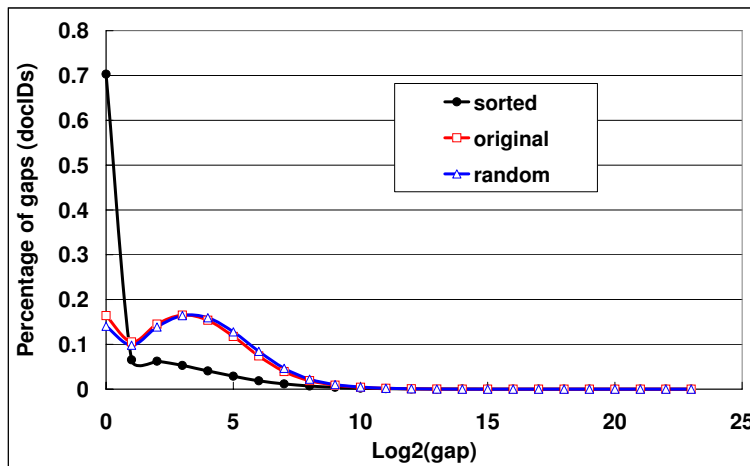


Figure 3.1: Histograms of d-gaps for inverted lists corresponding to 1000 random queries on the TREC GOV2 data set, under three different orderings: *original*, *sorted* and *random*. The x-axis is the number of bits required to represent d-gaps in binary, and the y-axis is the percentage of such d-gaps. (Thus, the first point is for 1-gaps, the second for 2-gaps, the third for 3-gaps plus 4-gaps, and so on.

In Figure 3.1, we show a histograms of d-gaps for the TREC GOV2 data set under three different orderings of documents: *Original*, which we get from the given ordering of documents in TREC GOV2 data set; *sorted*, where docIDs are

re-assigned by us after we sort their URLs, as in [78]; and *random*, where docIDs are assigned at random.

From Figure 3.1 we can see that the *sorted* ordering results in more small gaps than the other two kinds of indexes, suggesting a higher compression ratio. In addition, the d-gaps for the *original* ordering have a similar histogram as those for the *random* ordering, suggesting that the compression methods will very likely have a similar performance. Furthermore, we analyzed individual inverted lists and found that such a reordering results in more clusters (not shown in Figure 3.1), i.e., sequences of consecutive small d-gaps.

3.1.2 Optimizing PForDelta Compression

We now describe two modifications to PFD that achieve significant improvements over the versions in [43, 94, 98].

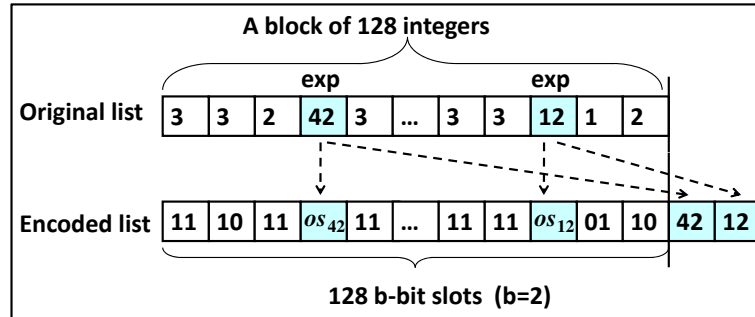


Figure 3.2: The implementation of the original PFD

The implementations of PFD in previous work, shown in Figure 3.2, encode a block of 128 value by first allocating 128 b -bit slots, and then for those 90% of values less than 2^b directly storing them in their corresponding slots. For each value larger than 2^b , called a *exception* (*exp*), we store an offset (*os*) value in the exception's corresponding slot indicating the distance from the current exception to the next one, and the actual value of the exception in some additional space after

the 128 b -bit slots. One disadvantage of such a code structure is that when two consecutive exceptions have a distance of more than 2^b , we have to use more than one offset to represent the distance, by forcing additional exceptions in between these two exceptions. We cannot solve this problem by simply increasing b since this would waste lots of bits on 90% of values; but if we decrease b more exceptions will be produced. This means in particular that this version of PFD cannot profitably use any values of b less than $b \leq 3$, but this case is very important in the reordered case since for such small values, most exceptions would be forced additional exceptions.

To overcome this problem, we present a new code structure for PFD, shown in Figure 3.3, by storing the offset values and parts of the exceptions in two separate arrays (while we still maintain 128 b -bit slots). In particular, for an exception, we store its lower b bits (*lo*), instead of the offset to the next exception, in its corresponding slot, while we store the higher *overflow* bits (*hi*) and the offset (*ofs*) in two separate arrays. Please note that unlike *os* in the original PFD, the value of *ofs* is not limited by the value of b . These two arrays can be further compressed by any compression method, and we find that S16 is particularly suitable for this. We call this approach NewPFD.

Our second improvement is in the selection of the b value for each block. As it turns out, selecting a constant threshold for the number of exceptions does not give the best tradeoff between size and speed. Instead, we model the selection of the b for each block as an optimization problem as follows: Given a limit t on the average time for processing a list, the NewPFD method, and a set of different values of b for NewPFD, we select for each block in the list a b such that the overall index size is minimized, while satisfying the time limit t .

The problem is obviously NP-Complete due to its relationship to Bin Packing, but we would expect a very good approximation via simple greedy approaches in

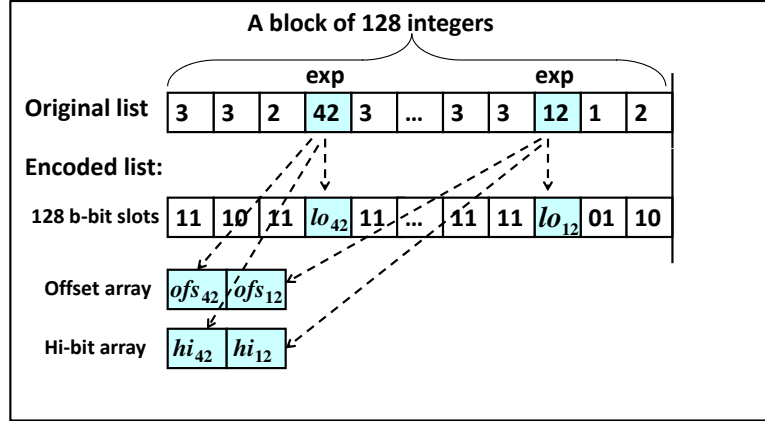


Figure 3.3: The implementation of NewPFD

this case. In particular, we take the following approach, which we call OptPFD:

- (a) Select a sufficiently large set of inverted lists. Build a look-up table where for each possible value of b and each possible number of exceptions n_{exp} of a block (which is achieved by compressing the block using NewPFD with the b), we can directly fetch the decoding speed in seconds per integer. Then for each inverted list, we do the following:
 - (i) For each (the i th) block in the list, and each b , measure the following: (i) $s_b(i)$, the compressed size of block under the b , and (ii) $t_b(i)$, the total amount of time spent decompressing the block using NewPFD using b , which is estimated from the table we built above based on the number of exceptions of the block using b .
- (c) Initially, assign to each block the b that gives the smallest size.
- (d) Now repeatedly greedily select a block and change values of b such that a faster but less space-efficient NewPFD can be achieved, until the time constraint is satisfied. In particular, in each step, choose the block that minimizes $(s_{b'}(i) - s_b(i))/(t_b(i) - t_{b'}(i))$ over all i and all $b' \neq b$, where b is

the current b used to compress the block.

In other words, OptPFD initially assigns the b with the smallest compressed size to each block, and then increases speed as desired by selecting a block that gives us the most time savings per increase in size, and changes the b of that block.

We note that for a given target speed, we can easily derive simple global rules about the choice of b , instead of running the iterative optimization above. Thus this version can be very efficiently implemented even on very large collections.

3.1.3 Optimizing Other Methods

We now present a few minor optimizations of some other methods that we used in our experimental evaluation.

GammaDiff: This is a variation of Gamma coding that, for a given integer x , encodes the unary part of the Gamma code (that is, $1 + \lfloor \log x \rfloor$) as the difference between $1 + \lfloor \log x \rfloor$ and the number of bits required to represent the average of all gaps in the list. The motivation for GammaDiff is that when docIDs are clustered, the differences between d-gaps and their expected average gap may be much smaller than the gaps themselves.

S16-128: As S9 and S16 only have 9 or 16 possible cases for encoding numbers, sometimes they have to choose a wasteful case when a better one might exist. Now suppose we have a sequence of numbers consisting mainly of small values. In this case, a version of S16 called S16-128 can do slightly better by providing more cases for small numbers and fewer for larger numbers.

Optimized IPC: Recall that the key step of interpolative coding (IPC) is to encode a number x in the range $\langle lo, hi \rangle$, where lo and hi are respectively the lowest and highest possible values of x . The original IPC encodes the offset $o = x - lo$ using a b -bit number, where $b = \lceil r \rceil$ and $r = hi - lo + 1$ is the number of possible values of the offset. This wastes bits if r is not a power of 2. We can do

better by using a trick from Golomb coding to encode o as follows: If $o < 2^b - r$, use $b - 1$ bits to represent o , otherwise use b bits to represent $o + 2^b - r$. (This technique was already described for IPC in [90].) In addition, before we apply the above optimization, we transform the range of values in such a way that the shorter codes are applied to values in the middle of the range, since such values are more likely even in a highly clustered list.

Also, while IPC is usually considered as a list-oriented method, meaning it starts by encoding the median of the entire list, we apply it to blocks of a certain size. As it turns out, this also improves compression if we choose a good block size. In particular, block sizes of the form $2^b - 1$ appear to work best, since such block sizes allow a more efficient implementation of the recursive process of IPC. For example, the for loop using use block size $2^b - 1$ can be simplified than using other block sizes. In our implementation we choose blocks of 127 values.

3.2 Frequency Compression

Frequency values tend to be quite small, and unlike docIDs, they are not in sorted order. In this section, we first discuss the effect of docID reordering on frequencies, and then propose more effective compression algorithms. In particular, we show that reordered frequencies can be transformed in such a way that their entropy is lowered significantly, leading to better compression.

3.2.1 Effect of Reordering on Frequencies

Frequency values by themselves are not changed at all by reordering, and thus reassigning docID by sorting URLs does not affect the distribution of frequencies. However, such an ordering results in more local clusters of similar values. This can be shown through the later experimental results by comparing the compressed size of context-sensitive and context-free methods. The context-sensitive methods,

which include IPC, S9, S16, and OptPFD, exploit the neighbor information to encode a number, while the context-free methods, such as gamma or delta coding, encode each number independently, resulting in no change in compression after reordering.

However, none of the existing methods takes full advantage of the local clusters created by the sorted ordering to further reduce compressed size. In the following, we show that under such an ordering, the context information of frequencies can be further exploited to reduce frequency values and thus significantly improve compression.

3.2.2 New Algorithms

The basic idea is that we exploit the context information of frequencies to transform them into even smaller values, using one of the following two techniques: a version of Move-To-Front coding (MTF) [13], and a method we call Most-Likely-Next (MLN). More precisely, we propose to perform a transformation on the frequency values before compressing them with other compressors.

Move-To-Front (MTF): The MTF [13] transform is used as an important part of Burrows-Wheeler transform-based compression [90]. Its basic idea is that, as long as a number has been seen lately, it will be represented by an index that is likely to be smaller than its own value, in a separate index array whose first element is always the number we just saw. For example, given a list of numbers [5, 5, 5, 5, 3, 2, 2], and assuming that all numbers are in the range [1,5], we keep a separate index array which is initialized as $\langle 1, 2, 3, 4, 5 \rangle$. We first encode the first number 5 as its index in the index array, which is the same as own value 5, and then move 5 to the front of the index array such that next time when we meet 5 again we will encode it as the index in the index array, which is 1, instead of the real value 5. From then on, whenever we meet a value, we encode it as its

index in the index array and move it to the front of the index array. Therefore, the original list could be encoded as $\langle 5, 1, 1, 1, 4, 4, 1 \rangle$. From the example we can see that MTF works well especially when there is a cluster of numbers of the same value.

We experimented with several MTF-based mechanisms for preprocessing frequency values. While the basic MTF version achieved some benefits, we found that other variants that do not directly teleport the last used element to the first slot in the array actually performed better. In the end, methods that move the last used value from its current position i to a position such as $i/2$ or $2i/3$ achieved overall best performance in our experiments. We also note that MTF may slow down the speed of decompression, especially when the range of values is large, since we have to do exactly the same move-to-front operations for all numbers to be decoded.

Most-Likely-Next (MLN): An alternative called MLN is also used to transform numbers to smaller values, but can overcome some problems of MTF. In a nutshell, MLN uses a table that stores for each value (within some limited range $[1 \dots Q]$) which values are most likely to follow. Thus, for $Q = 16$, MLN would rely on a 16×16 array as a look-up table, precomputed for each list. Each row of the array is for a particular value i within the range $[1 \dots Q]$, and contains a sequence of Q values, where the j th value is the j th most likely value to follow the value of i in the list to be compressed. For example, suppose that we are compressing a sequence of one million numbers and $Q = 16$, then the resulting 16×16 array may look like the one shown in Figure 3.4.

From Figure 3.4, we can see that for value $i = 2$ (the second row), the number that most frequently follows it is 6, while the number that follows it second most frequently is 2 itself, and the number that follows it least frequently is 16. We note that the table does not store any information about the values $\geq Q$, which

1:	1, 3, 4, 2, 6, 16, 5, 7, 8, 10, 9, 11, 13, 12, 15, 14
2:	6, 2, 4, 3, 5, 1, 8, 11, 12, 13, 7, 15, 9, 10, 14, 16

16:	16, 15, 13, 14, 12, 9, 11, 8, 7, 3, 4, 5, 2, 1, 6, 10

Figure 3.4: An example of the look-up table for MLN with $Q = 16$

are not changed in the transformation.

With such a look-up table, we compress the list by replacing each value with its rank in the array indexed by the value of its predecessor. For example, suppose a segment of the list to be compressed is (2, 6, 2, 6, 16, 6). In order to compress the k th number, we need to check the $k - 1$ th number, and use k 's rank in the i th row of the look-up table to represent its original value k (the very first number of the list will be encoded just as itself since it has no predecessors). For instance, the second number, 6, is encoded as 1 since it is the most frequent number to follow 2 (that is, the first number after the colon on the second row shown in Figure 3.4). However, the last 6 will be encoded as 15 based on its rank on the 16th row in the table and. Thus, the resulting codes of the above list will be (2, 1, 1, 1, 4, 15).

From above, we can see that MLN needs to store a small extra array for each list. However, in our experiments, MLN outperformed the best version of MTF in terms of both size and decompression speed (since during decompression, MLN can directly fetch the values from the in-memory look-up tables while MTF has to do more expensive move-to-front operations). Both MTF and MLN result in significant runs of 1 values in the transformed set of frequencies, since many frequency values under the ordered list are followed by more occurrences of the same value.

3.2.3 Experimental Results on Compression

Experiment Setup: For our experiments, we use the TREC GOV2 data set of 25.2 million web pages from the `gov` domain that is distributed by the US National Institute of Standards and Technology (NIST) and used in the annual TREC competitions. This data is widely used for research in the IR community, thus allowing others to replicate our results. It is based on a 2004 crawl of the `gov` domain, and is also accompanied by a set of 100000 queries (the 2006 Efficiency Task Topics) that we use in our evaluation.

While the data set does not represent a complete snapshot of the `gov` domain at the time of the crawl, it nonetheless contains a fairly significant subset of it. This is important since our techniques perform best on “dense” data sets such as GOV2 that are based on a fairly deep crawl of a subset of domains. In contrast, a “sparse” set of 25.2 million pages crawled at random from the many billions of pages on the web would not benefit as much.

We then selected 1000 random queries from the supplied query logs; these queries contain 2171 unique terms. All experiments were performed on a single core of a 2.66GHz Intel(R) Core(TM)2 Duo CPU with 8GB of memory.

Experimental Results on docID Compression: In Table 3.1, we compare the original IPC, which is list-wise, with its improved version with various optimizations and its block-wise version with our optimizations, on the GOV2 data set under the original, sorted, and random orderings. From Table 3.1, we can observe the following: First, all IPC algorithms work significantly better on the d-gaps under the *sorted* ordering than under the other two orderings; second, both list-wise and block-wise IPC with optimizations are much better than the original IPC, but block-wise IPC with optimizations achieves the best compression (the block size is 127).

Compared to IPC, the main advantage of PFD is that decoding is very fast. In

	sorted	original	random
list-IPC w/o opt	0.95	2.70	2.83
list-IPC	0.88	2.46	2.57
block-IPC	0.85	2.40	2.51

Table 3.1: Compressed size in MB/query for docIDs using a basic list-wise IPC (without optimizations), a list-wise version with all other optimizations enabled, and its block-wise version, under the original, sorted, and random orderings.

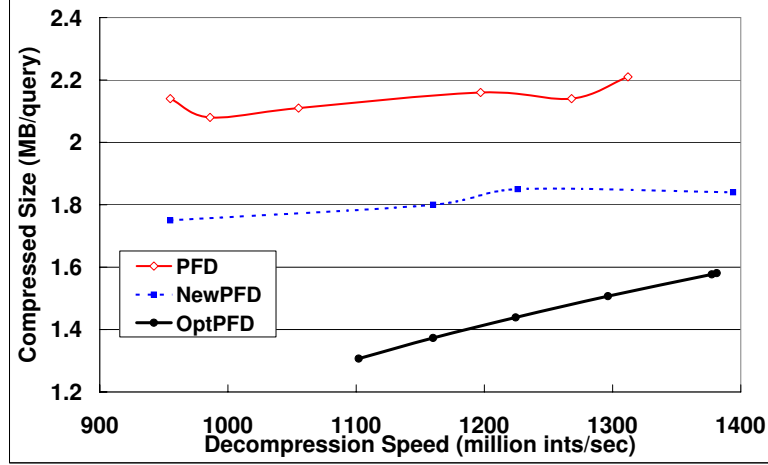


Figure 3.5: Compressed size in MB/query versus decompression speed in million integers per second for docIDs, using PFD, NewPFD, and OptPFD under the sorted ordering. The points from left to right for PFD and NewPFD correspond to the following percentages of exceptions: 5%, 8%, 10%, 20%, and 30%. For OptPFD, the points correspond to different target speeds for the optimization and their corresponding sizes.

Figure 3.5, we show the trade-offs between decompression speed and compressed size for PFD (as used in [94]), NewPFD, and OptPFD as introduced above. From Figure 3.5, we see that OptPFD can always achieve a much smaller compressed size for a given decoding speed than the other PForDelta-based methods. Thus, choosing b not based on a global threshold on exceptions, but based on a global target speed, achieves a much better trade-off than the naive global threshold used in PFD and NewPFD. While OptPFD is still worse than IPC in terms of compressed size, decompression is much faster than for any version of IPC (as we will show later). We also ran experiments under the original document ordering, and observed slightly smaller but still significant gains for OptPFD over PFD and

NewPFD, while PFD and newPFD were overall similar in performance.

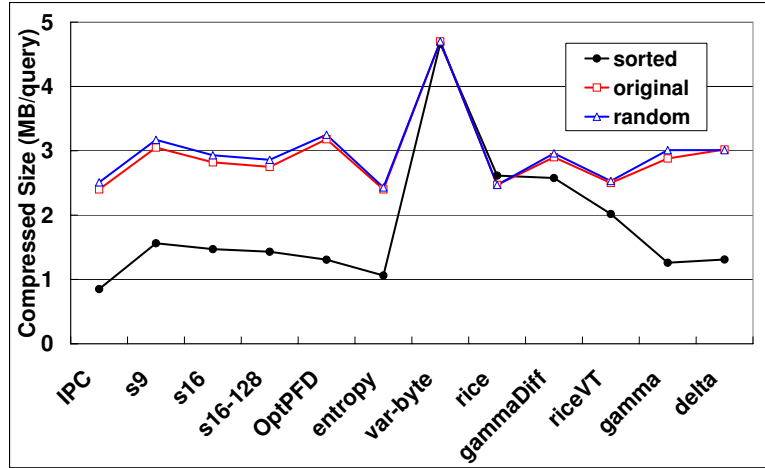


Figure 3.6: Compressed size in MB/query for docIDs using twelve methods, under the original, sorted, and random orderings.

In Figure 3.6, we compare the average compressed size per query of the docIDs for many state-of-the-art inverted index compression methods on the TREC GOV2 data set under the original, sorted, and random orderings. For each data set, we show results of twelve compression methods: var-byte, S9, S16, S16-128, OptPFD, Delta coding, Gamma coding, GammaDiff, Rice coding, a variant of Rice coding called RiceVT described in [90] which essentially promotes the implicit probabilities of small gaps, the block-wise interpolative coding with our above optimizations, and entropy, which uses the global frequency distribution of the compressed integers. For OptPFD, we chose a setting that minimizes the compressed size (i.e., the leftmost point in Figure 3.5).

From Figure 3.6, we make the following observations: First, just as Figure 3.1 suggested, many compression methods achieve a much better compression ratio on the d-gaps under the *sorted* ordering than under the other two orderings; second, all compression methods on d-gaps under the *original* ordering achieve similar performances with those under the *random* orderings; third, IPC achieves

the best compression performance among all methods; fourth, OptPFD is quite competitive with all other methods (even with IPC, although it is slightly worse than IPC in terms of size). One disadvantage of IPC is that its decompression is slow, as we will show later. In contrast, all other methods to the left of the *entropy* method are fairly fast, and much faster than those further to the right.

Experimental Results on Frequency Compression: We first show in Figure 3.7 the impact of reordering on frequency compression. In Figure 3.7, we display the compressed size of the frequency data under state-of-the-art compression methods on the TREC GOV2 data set, using original, sorted, and random orderings. From Figure 3.7, we see exactly what we would expect: The context-sensitive methods (all methods to the left of entropy) get better compression results under the sorted ordering than under the other orderings, while the other methods get the same results under all three orderings. We also notice that for the context-sensitive methods, compression under the original ordering has very similar performance with that under the random ordering. As before, IPC (with MLN) achieves the best compression performance, which is 0.92MB. (The compressed size using IPC without MLN is 1.92MB, which will be shown in the table below).

We also compare the performance of our PForDelta variants, PFD, NewPFD, and OptPFD, on frequency values under sorted document ordering. The results are shown in Figure 3.8, where we see that again OptPFD significantly outperforms the other two versions in terms of the trade-off between decoding speed and size.

In Table 3.2, we compare the average compressed sizes of the frequency data per query on the TREC GOV2 data set, under the original, sorted, and random orderings. We use three different versions each for list-oriented and block-oriented IPC: The best version from before, a version that uses MTF, and one that uses

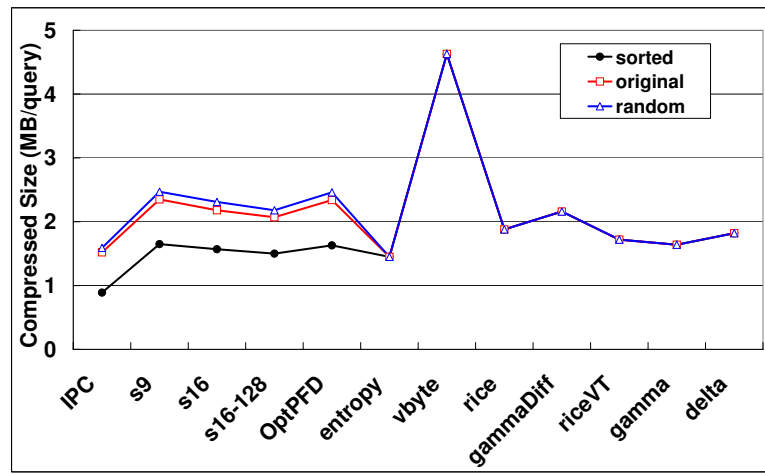


Figure 3.7: Compressed size in MB/query for frequencies using twelve methods, under the original, sorted, and random orderings.

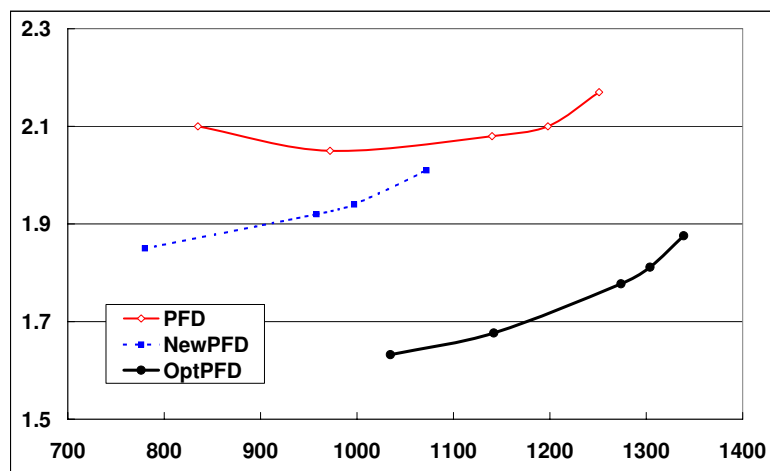


Figure 3.8: Compressed size in MB/query versus decompression speed in million integers per second for frequencies, using PFD, NewPFD and OptPFD, under sorted ordering.

MLN.

	list			block		
	sorted	orig	rand	sorted	orig	rand
IPC	1.26	1.65	1.71	1.21	1.59	1.65
IPC-MTF	0.93	1.65	1.75	0.89	1.59	1.69
IPC-MLN	0.92	1.58	1.65	0.89	1.52	1.59

Table 3.2: Compressed size in MB/query for frequencies, under the original, sorted, and random orderings, using IPC, IPC with MTF, and IPC with MLN, for list- and block-oriented methods.

From Table 3.2 we make the following observations: First, as with docIDs, IPC performs much better under sorted ordering than under the original and random orderings, and the block-wise versions always perform better than their list-wise counterparts; second, for frequencies under the sorted ordering, the versions with MTF and MLN are much better than the one without them; third, IPC with MLN slightly outperforms IPC with MTF.

	basic	MTF	MLN
block IPC	1.21	0.89	0.89
s9	1.65	1.53	1.52
s16	1.57	1.44	1.43
s16-128	1.50	1.38	1.37
NewPFD	1.88	1.73	1.72
OptPFD	1.63	1.43	1.31
entropy	1.45	1.13	1.14
var-byte	4.63	4.63	4.63
rice	1.88	1.70	1.69
gammaDiff	2.16	1.80	1.79
riceVT	1.72	1.44	1.43
gamma	1.64	1.52	1.28

Table 3.3: Compressed size in MB/query for frequencies under sorted document ordering.

Both MTF and MLN can also be applied to the other algorithms to get better compression ratios. From Table 3.3, we observe the following: First, the entropy is greatly reduced by either MTF or MLN; second, all methods except var-byte improve over their basic versions, no matter whether they use MTF or MLN; third, MLN is usually better and never much worse than MTF. We also tried MTF and MLN transformations of d-gaps for docIDs, but there was no benefit.

3.3 Query Processing Performance

One interesting result of our experiments is that reordering of documents, in addition to improving compression, also speeds up index traversal in a DAAT query processor. In particular, our query processor (with no changes in the software, and independent of compression method) performs more and larger forward skips during index access in the reordered case, and as a result decompresses less than half as many blocks per query as in the unordered case (as we shown further below). Note that this is related to, but different from, recent work in [19,27] that shows how to choose an optimal set of forward pointers (basically, how to choose variable block boundaries) for each list based on an analysis of the query load. Thus, we reorder documents while keeping block sizes constant, while [19,27] modify block sizes while keeping the ordering constant; it would be interesting to see how the approaches work in combination, and whether the reordering could be improved by considering query loads.

In previous sections, we studied the compression ratios of various techniques on typical queries, but did not consider decompression speed, total index size, and query processing performance. In this section, we study these issues in detail.

We start out with decompression speed. In the experiments, we used the optimized decompression methods from [94] for var-byte, Rice coding, S9, and S16, and S16-128, NewPFD with fixed threshold 10% for exceptions, OptPFD with minimum compressed size, and the best block-wise version of IPC. (We did not try to implement optimized decompressors for gammaDiff, riceVT, gamma, and delta coding, as these methods are known to be relatively slow.) In Table 3.4 we give for each method the decoding speed in millions of integers decoded per second for three cases: Decompression of docIDs, and decompression of frequencies with and without MLN transformation.

	docID	freq	freq-MLN
var-byte	637	729	273
s9	748	846	269
s16	691	898	267
s16-128	498	550	245
NewPFD	1055	1120	298
OptPFD	1102	1034	212
rice	489	404	199
IPC	55	51	52

Table 3.4: Decoding speeds in millions of integers decoded per second, for docIDs, frequencies, and frequencies with MLN transformation.

The results in Table 3.4 are overall not surprising. NewPFD and OptPFD are the fastest techniques, though S9, S16, S16-128, and var-byte are also quite efficient. In contrast, IPC is much slower. Adding MLN slows down the faster methods significantly, but does not impact slow methods such as IPC much. We note that additional increases in speed can be obtained for OptPFD by trading off size versus speed.

Next, we look at total index size. For this, we built block-wise compressed indexes for three methods that we believe provide the most interesting trade-offs between decompression speed and compressed size: IPC, NewPFD, and OptPFD. We compare ordered and unordered indexes, and for ordered indexes we provide numbers both with and without MLN. The results are shown in Table 3.5. We see very significant improvements in index size through document reordering.

The best compression is obtained with IPC, using MLN for frequencies, which results in a total index size of around 3.45 GB. This compares to an index size of about 3.88 GB for the smallest size under OptPFD, using sorted docID ordering and MLN for frequencies. In fact, even without MLN (which as shown earlier slows down OPTPFD significantly) we can obtain an index size only slightly larger than 4 GB. In contrast, NewPFD results in much larger index sizes, of 5.5 GB and more, showing the benefit of OptPFD over NewPFD. We note that many other sizes between 4 GB and 5.5 GB can be obtained by trading off size versus

speed in OptPFD (though even the smallest size results in fairly fast decoding). However, note that even NewPFD is much better than the best unordered results, and that all the ordered indexes can be completely held in main memory given realistic memory sizes of 4 to 6 GB.

	sorted			original		
	IPC	New	Opt	IPC	New	Opt
docID	2617	3746	2853	5365	6122	5903
freq	1142	2027	1255	1363	2307	1653
total	3759	5773	4108	6728	8429	7556
f+MLN	834	1844	1023	–	–	–
total	3451	5590	3876	–	–	–

Table 3.5: Compressed index size in MB for the entire TREC GOV2 data set, for IPC, NewPFD with 10% threshold on exceptions, and OptPFD optimized for minimal index size. For the sorted case, we provide numbers for frequencies and total index sizes with and without MLN.

Another interesting observation is that the ratio of frequency data to docID data is much smaller than in our previous experiments. The reason is that when looking at total index size, we include a large amount of data in shorter (but not very short) lists, while our query-based measurements are skewed towards longer lists. In shorter lists, d-gaps are larger while frequency values tend to be smaller, compared to longer lists. The benefits of OptPFD over NewPFD for compressed size also tend to be larger on these lists, particularly for frequencies.

Next, we look at query processing speed for intersection-based queries using BM25 ranking. Table 3.6 shows query performance for an index compressed with OptPFD (but no MLN for frequencies) using ordered and unordered docID assignments, under the assumption that all index data is in main memory. Somewhat surprisingly, the ordered index is about twice as fast as the unordered one! Note that this is not due to savings in disk accesses, as all the data is in main memory, and also not due to changes in decompression speed, as the ordering has only a moderate impact on the speed of OptPFD. Instead, as shown in Table 3.6, this is mainly due to the ordered index decoding significantly fewer blocks of data than the unordered one.

	sorted	original
running time (ms/query)	6.15	12.08
num of docIDs decoded (million/query)	0.71	1.53
num of freqs decoded (million/query)	0.53	1.04

Table 3.6: Running time and number of decoded docIDs and frequencies for OptPFD on the GOV2 data set.

In fact, this increase in speed can be explained in a simple and intuitive way. Consider the shortest list in a query. Under DAAT query processing, almost all of the shortest list will be decompressed, and most docIDs in this list will generate a lookup into the next longer list. If the docIDs in the shortest list are clustered, then more of these lookups will hit the same block of the next longer list, while other blocks do not get hit at all and do not have to be decompressed. (Informally, if we throw enough balls uniformly at random into n bins we will hit almost all bins, but if our throws are clustered in certain areas, then many more balls are needed to hit most bins.) A formal analysis of this phenomenon is complicated by dependencies between terms and by queries with more than two terms, and we leave this for future work.

Finally, we also give query processing speeds for other compression methods, in particular IPC and NewPFD, with and without docID reordering. Note that the number of decompressed blocks per query does not change, as all methods use the same blocks of 128 postings. As we see in Table 3.7, we also get significant improvements in query processing speed for the other methods by using ordered indexes. However, the method achieving the best compression, IPC, is much slower than the faster methods. NewPFD is even faster than OptPFD, but as shown in Table 3.5, the index size is much larger. However, the same speed at lower index size could be obtained by trading size for speed within OptPFD (not shown here).

	sorted	original
IPC	29.44	59.18
NewPFD	4.98	9.74
OptPFD	6.15	12.08

Table 3.7: Running times in ms per query for IPC (with MLN), NewPFD, and OptPFD.

3.4 Mixed-Compression Indexes

In previous sections, we have seen that using reordered index structures results in significant improvements in index size and query processing speed. However, the best method in terms of size, IPC, which outperforms all other methods by a significant margin, is fairly slow and can decompress only about 50 million integers per second. The fastest methods, PForDelta and its variants, i.e., PFD, NewPFD and OptPFD, are around 20 times faster, but produce a larger index (though the index size for PForDelta under reordered docIDs is still better than for the best method without reordering). Thus, there is a trade-off between size and speed among the different methods.

This motivates the question of whether we can get a better trade-off by combining different compression methods in the same index. Our index setup can easily accommodate different compressors within the same index (or even the same list), as all compression is performed in a block-wise fashion. Moreover, from studies on inverted index caching, we know that different parts of the index have very different access frequencies; e.g., in [94] more than 90% of all index accesses can be served from a cache of 30% of the index size. Thus, we could exploit this highly skewed access pattern, by compressing frequently accessed inverted lists using a very fast method, and other lists using a slower method that gives better compression. Our goal is for the resulting index to have both size and speed close to the best achieved by any method.

More formally, we are interested in the following problems:

Problem 1: Given a limit t on the average time for processing a query, and a set of available compression methods, select for each inverted list a compression method such that the overall index size is minimized, while satisfying the time limit t .

Problem 1': Given a limit t on the average time for processing a query, a limit b on the amount of I/O bandwidth (in MB/s) that is available, a caching policy P that uses some main memory to cache index data, and a set of available compression methods, select for each inverted list a compression method such that the total amount of main memory that has to be available for caching is minimized, while satisfying the limits on t and b .

In the first problem, we are looking at a main-memory resident index, and our goal is to minimize the amount of memory we need to provide, given a (feasible) time constraint. Our hope is that by relaxing the time constraint very slightly versus the minimum, we can very substantially decrease the memory requirement. The second problem looks at an index that is partially cached in memory (a very common setup in practice), and the goal is to minimize the amount of memory that needs to be provided for caching to assure that the available I/O-bandwidth does not become the main bottleneck of the system. Note that the first problem is the special case of the second where $b = 0$, i.e., no disk access is allowed. Also, there are obviously many other ways to set up these optimization problems, including duals of the above, or setups that model the search architecture in more detail.

We focus on Problem 1. The problem is obviously NP-Complete due to its relationship to Bin Packing, but we would expect a very good approximation via simple greedy approaches in this case. In particular, we take the following approach:

- (a) Select a sufficiently large query trace. For each available compression method,

build an index and issue the queries against this index.

- (b) For each inverted list I_w for a term w , and each compression method c , measure the following: (i) $s_c(w)$, the compressed size of list I_w under method c , and (ii) $t_c(w)$, the total amount of time spent decompressing the list using method c over the given query log.
- (c) Initially, assign to each inverted list the compression method that gives the smallest size.
- (d) Now repeatedly greedily select a list I_w and change its compression method to a faster but less space-efficient method, until the time constraint is satisfied. In particular, in each step, choose the list I_w that minimizes $(s_{c'}(w) - s_c(w))/(t_c(w) - t_{c'})$ over all w and all methods $c' \neq c$ where c is the compression method currently used for I_w . In other words, choose the list and compression method that gives you the smallest increase in index size per time saved.¹

We note that query processing time in our setup consists of the time for decompression and the times for other tasks such as intersection and score computation, and that the latter are independent of the compression methods used (since all methods use the same block size for decompression). Thus, we can easily check if the time constraint is satisfied in (d) without reexecuting the query trace. Also, for best results it is useful to treat the frequencies and docIDs of a list separately, as most queries decompress fewer frequency data than docID data.

We implemented the above method, and ran it on 99000 of the 100000 queries on the TREC GOV2 data set, leaving the other 1000 for testing the performance of the resulting configuration. In Figure 3.9, we show results for a hybrid index combining IPC (with MLN) and OptPFD (without MLN or MTF). As shown,

¹We assume here that both numerator and denominator are strictly positive.

while IPC requires about 29 ms per query, we can get less than 12 ms with almost the same size by using a hybrid index. We also note the version of OptPFD that we used only minimizes compressed size, and that a better overall tradeoff than the one in the figure could be achieved by selecting different settings for OptPFD. (In fact, this hybrid index optimization problem motivated the optimization problem underlying the size/speed tradeoff for OptPFD in Figure 3.5.)

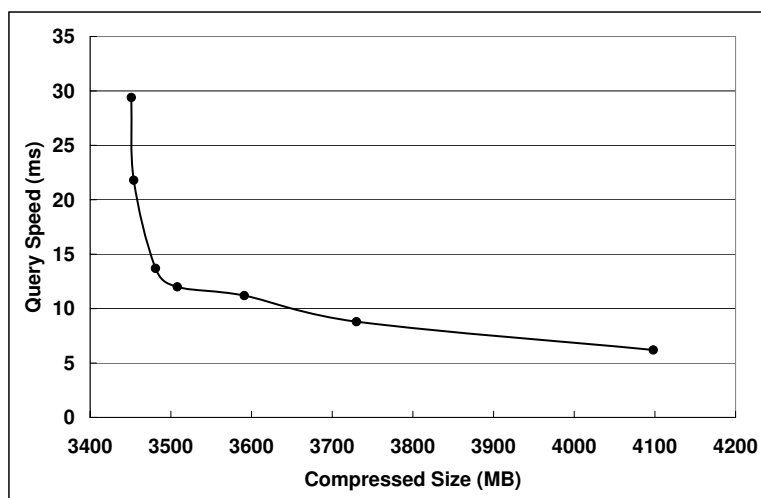


Figure 3.9: Total index size in MB versus processing speed per query in milliseconds, for a hybrid index involving OptPFD and IPC. The leftmost point is for pure IPC and the rightmost for pure OptPFD (without MLN or MTF).

Chapter 4

Position Compression

As discussed before, there is a significant amount of previous work on inverted index compression in web search and IR areas. However, most previous work focuses on the compression of docID and frequency data, or on the compression of positions within longer linear texts such as books. In contrast, in this chapter, we focus on position data for web indexes, where each page typically consists of only a few hundred words. This problem is important for two reasons. First, the size of the position data is typically several times larger than the docID and frequency data, thus having a significant impact on query processing efficiency. Second, positions are becoming increasingly important in scoring functions, as recently studied, e.g., in [26, 55, 58, 73, 84].

An important consideration in the compression of position data is the tendency of term occurrences to cluster, i.e., if a word occurs in a particular sentence on a page, then it is more likely to occur again soon thereafter, in one of the next sentences. It is important to exploit this clustering property, and suitable techniques can achieve significantly better compression on such clustered occurrences than in the uniform case.

However, compression of position information in web indexes differs from the traditionally studied case of positions in longer texts in that each page in a web

index is a separate document that may or may not be similar to the previous page (depending on the page ordering used, but also on the properties of the collection). In contrast, when compressing positions in a book, there are usually significant similarities between consecutive pages in the book, or consecutive sections and subsections. Thus, in web indexes it is difficult to identify any clustering effects beyond page boundaries, and the focus is on exploiting what clustering exists within each page itself. Compression of position data in web indexes also differs from the case of docID and frequency compression in that additional information such as page size and term frequency is available.

In this chapter, we focus on techniques for compressing position information in web indexes. We describe several new techniques, and perform a detailed experimental evaluation of existing and new techniques. We also show how to efficiently use compressed position data in ranking functions that take position information into account.

4.1 Adaptive Index Compression Methods

Recall that in Chapter 2 we have mentioned that index compression methods can be roughly divided into context-free methods, such as gamma and delta coding [34], Golomb and Rice coding [90, 97], and variable-byte coding [75, 89], which compress each number in the sequence independently, and context-sensitive methods, which compress the sequence by taking into account context information. In fact, in the experiments in that last chapter, we have shown that many context-sensitive methods, for example, S9 [5] and its variants [2, 4, 6, 94] and PForDelta [98], can achieve more benefit from reordering techniques than context-free methods, since reordering makes the docIDs on the inverted lists more clustered. Intuitively, they may also apply to position compression since, as explained earlier, occurrences of terms in text tend to be clustered (or lo-

cally homogeneous [59]) rather than spread out uniformly. Several previous papers [20,21,59–61] have proposed compression methods that exploit this property.

However, these methods are usually only able to exploit clustered sequences of gaps that are long enough, say gaps between word occurrences in books or other collections that contain sufficiently long stretches of linearly ordered text. In contrast, the average web page contains a few hundred words, and pages are ordered in the index via a docID that may be assigned based on criteria such as global page quality or crawl order. Thus, while clustering does occur within a document, there are typically only a few occurrences of the word, and the next document in the ordering is not related to the previous document. In addition, previous work did not exploit page-related information to compress positions, but treated all positions as one uniform sequence of numbers.

Intuitively, the more adaptive methods can take better advantage of the context information to achieve better compression than less adaptive methods. For example, LLRUN [38] and its variants [62], HMM (Hidden Markov Model)-based methods [20,21], Interpolative Coding (IPC) [60,61], and BASC and RBUC [59].

We now formalize this notion. We say that a method is *oblivious* if it compresses each value on its own, without using any collection- or list-specific statistics such as the average document size in the collection, or the length of a list. Examples are Gamma, Delta, and variable-byte coding. A method is *list-adaptive* if it compresses an inverted list of positions by using only statistics about the entire list or collection; examples are Golomb and Rice coding which use the average value in the list and the length of the list. A method is *page-adaptive* if it compresses the positions for a particular posting using document or posting features such as the document length or the frequency of the term in the document. An example would be the simple document-oriented version of Rice coding described later, which chooses a different parameter B for the positions in each posting based on

these features. Finally, fully adaptive methods may compress a position by also taking into account other position values in the same posting that have already been encoded; an example would be interpolative coding when applied within a single posting. In general, to properly exploit clustering of positions within documents, it is necessary to use page- or fully adaptive techniques. (Note that not all methods can be clearly categorized according to this taxonomy.)

4.2 Position Gap Distribution

From Chapter 2 we have seen that to some degree, d-gaps on the TREC GOV2 data set conform to a monotonically decreasing distribution, that is, most of the d-gaps have small values while much fewer of them have large values. However, in this section, we will find that this is not true for p-gaps through the following discussion on the p-gap distribution of the TREC GOV2 data set.

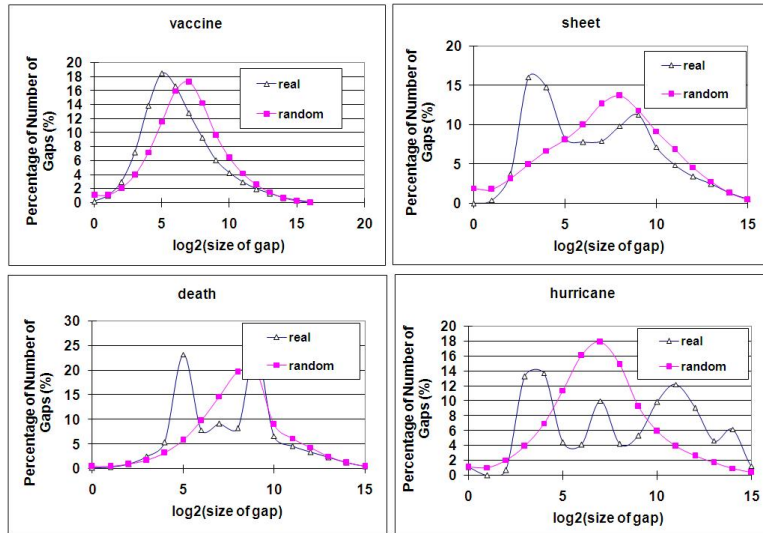


Figure 4.1: Distribution of p-gaps for four words on the TREC GOV2 data set. On the x-axis is the number of bits required to represent the p-gaps in binary, and on the y-axis is the percentage of p-gaps that fall in this range.

As examples, we select four terms and draw their corresponding p-gap distri-

butions in Figure 4.1. We show two graphs for each term, the distribution of real p-gaps and the distribution one would observe if all words were randomly arranged in the page. We expect that in the presence of clustering, these two graphs would behave very differently. From Figure 4.1 we can see that the real distributions are very different from the random (i.e., geometric) distributions. The distributions of real gaps for “sheet“, “death“, and “hurricane“ are very different from the random gaps, while the two distributions for “vaccine“ are more similar.

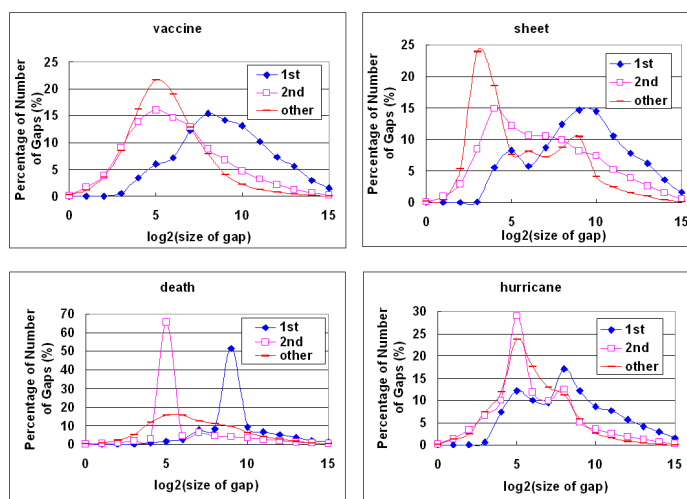


Figure 4.2: Distribution of p-gaps for first, second, and further occurrences, for the four words from Figure 4.1

A term may occur several times in a particular document, and different occurrences may behave very differently. In Figure 4.2, we plot the distributions of gaps for first occurrences, second occurrences, and further occurrences within a document, for the same four words as above. From Figure 4.2, we can see that for “sheet”, “death”, and “hurricane”, the distributions on gaps of its first occurrences, second occurrences and other occurrences are quite different from each other and show bursts at fairly distinct sizes of gaps. In fact, besides the index

of the occurrence, there are many other factors that may affect the distributions, e.g., document size and in-document frequency. Thus, it seems hard to capture the true probability distribution of all gaps with a single model.

4.3 Our Algorithms

In this section, we first propose a very simple but effective algorithm, *Remaining Page-Adaptive Rice Coding* (RPA-RC), and present its advanced version with smoothing based on a regression model (RPA-RC-S). We then propose another algorithm, *Remaining Page-Adaptive BASC with smoothing* (RPA-BASC-S), and perform an experimental comparison with a number of baseline algorithms from the literature.

4.3.1 Page-Adaptive Rice Coding

Standard Rice coding determines its parameter B by looking at the entire list of integers that need to be compressed, thus making it a list-adaptive algorithm. However, during decompression of position data, we already know the number of positions in the page (the frequency) and the overall page size, and it would be smart to exploit this knowledge for better compression. A fairly obvious way to do this is to select B as the largest power of 2 such that $B \leq |d|/(f_{t,d} + 1)$, where $|d|$ is the size of the current document and $f_{t,d}$ the frequency. We call this page-adaptive variant of Rice coding *Page-Adaptive Rice Coding* (PA-RC).

A fully adaptive version, called *Remaining Page-Adaptive Rice coding* (RPA-RC), takes this idea one step further and uses a different B for each position in the posting. In particular, rather than taking the size and frequency of the complete page, we consider the currently remaining page size and frequency of the posting. Thus, after encoding a position value p , we deduct p from the page size, and 1 from the frequency, and then use these updated values to select the B for the next

position in the posting. If one of the gaps is very large, then this implies that subsequent positions occupy a smaller region towards the end of the document, and the method will use a smaller B to encode those remaining positions.

4.3.2 Adaptation with Smoothing

However, RPA-RC may suffer in the case in Figure 4.3, which shows the locations of occurrences in a document of a word.



Figure 4.3: An example of word locations in a document.

There are two clusters of occurrences in Figure 4.3 that are separated by a wide gap. In the first cluster of occurrences, the remaining average gap for its last occurrence is large, even though its gap with its previous occurrence is small, which is very useful information ignored by RPA-RC. To deal with this problem, we integrate the information about the previous gap into our method and build a regression model as follows:

$$B_t = (1 - p) \cdot g_{t-1} + p \cdot r_t$$

where B_t is the value of B in Rice coding for the t -th gap, r_t is the remaining average gap for the t -th gap, and g_{t-1} is the value of previous gap. The second term in the model is used to tune the prediction error by using the remaining average gap. When $p = 0$, it means that the current expected average gap B_t is equal to the previous gap g_{t-1} , while $p = 1$ means it is equal to the remaining average gap r_t .

We note that BASC coding [59] also exploits the previous b_{i-1} to predict the

next b_i . An extension of BASC called BASC-smooth uses the average value of k previous values of b to predict the next b . As shown in [59], this achieves better compression than the basic BASC. Motivated by this, we replace the g_{t-1} in the above regression model with the previous average gap. The modified model is called *Remaining Page-Adaptive Rice Coding with Smoothing* (RPA-RC-S).

On the other hand, list-wise BASC-smooth can also be modified to be page-adaptive as follows: First, unlike list-wise BASC in [59], where the value of b is initialized for the entire list as a fixed number, say 2, or 4, or 8, page-wise BASC-smooth initializes it as the average gap of its corresponding page. Second, for page-wise BASC-smooth, only the previous gaps within the same posting need to be checked to calculate the previous average b , thus avoiding the noise caused by previous postings. More interestingly, motivated by RPA-RC-S, where we tune the predictions by the remaining page information, we can tune the prediction of BASC-smooth by using the following model:

$$b_t = (1 - p) \cdot avg_{t-1} + p \cdot rb_t$$

where b_t is the expected number of bits to encode the current gap into a binary code, avg_{t-1} is the average number of bits to encode previous gaps, and rb_t is the number of bits to encode the remaining average gap. Thus, when $p = 0$, the current gap is encoded by the same number of bits used for the previous gap, while when $p = 1$, it is encoded by the number of bits for the remaining average gap. We call this variant *Remaining Page-Adaptive BASC with Smoothing* (RPA-BASC-S).

4.3.3 Multidimensional Adaptation

Most of the above page-adaptive methods compress the current position by exploiting two-dimensional (2D) context features: the page size (or the remaining page size) and the frequency. In fact, from experiments in later sections, we will see

that most page-adaptive methods are already much better than non-parametric or list-adaptive methods by taking advantage of these two features. Intuitively, we expect that the more context features we use, the better the compression performance we can get. For example, the above regression-based methods improve the compression performance slightly by adding as an additional feature the previous gap (or previous average gap).

However, as discussed in Section 4.2, different terms may behave so differently that it is hard to make a good prediction of the next value based on a single model. In this case, it might be better to augment general statistics-based compression methods such as, e.g., Huffman coding, with context information to improve compression.

The basic idea is as follows: For each inverted list, we first classify all p-gaps into one of a moderate number of buckets, depending on four context features: The remaining document size $rsize$, the remaining frequency $rfreq$, the previous p-gap $prev1$, and the previous previous p-gap $prev2$. To do so, we divide the values of each feature into a small number of bins such that two p-gaps are in the same bucket if they fall into the same bin for all features. We then apply for each bucket a separate model, in one of the following two ways:

Optb-4D: For each bucket, we determine the optimal value of b under Rice coding, by trying all 32 possible values and choosing the one leading to the smallest compressed size. Thus, we have to maintain and store a global table to record the optimal bs for all of the buckets. During decompression, we first retrieve the global table from the codes, then for each position, we determine which bucket the position belongs to (based on the above four features, $rsize$, $rfreq$, $prev1$, and $prev2$), and retrieve the corresponding b from the global table and use it to decompress the position.

Huff-4D: Huff-4D is similar to Optb-4D except that it stores a separate Huff-

man table (instead of just the best value of b) for each bucket, and uses this table to encode the positions in the bucket. Like Optb-4D, Huff-4D also has to store these extra Huffman tables together with the compressed positions for decompression.

LLRUN-4D: LLRUN-4D can be seen as a special case of Huff-4D. However, it classifies p-gaps based on the unary parts of gamma codes of positions, instead of the position themselves. Thus, all positions that fall into the same bucket share the same $rsize$, $rfreq$ and the unary parts of the gamma codes of $prev1$ and $prev2$. It then builds a separate Huffman table for each bucket of positions and uses that table to encode them. LLRUN-4D uses (slightly) fewer codewords than a Huffman table in Huff-4D (which employs a more fine-grained scheme for selecting codeword boundaries in the Huffman tables). Like the above two methods, LLRUN-4D also has to store those tables together with the compressed positions for decompression.

Note that while such multidimensional models can easily be extended to use more features, this does not necessarily result in smaller compressed sizes. This is because the resulting models (Huffman tables, or b values) need to be stored together with the compressed indexes, and this cost increases quickly with additional features.

4.4 Google’s Index Layout for Positions

Recall the two kinds of layouts we discussed in Chapter 2, both of which store docIDs, frequencies, and positions. We call these two kinds of layouts *normal* layouts. In contrast, we learn from a recent keynote talk [31] that Google used another new index format to store its indexes. We call this new format *Google* layout. The *Google* layout has only a single global position space and does not contain any docIDs or frequencies, which can be derived from the global positions and some other special data structures recording the document boundaries. An

intuitive way to achieve the *Google* layout is to concatenate all web pages into a huge document and then sequentially assign positions for all words in it.

From the above, we see that the inverted list under such a new layout is composed of only global positions. Unlike in the *normal* layouts, where positions are in a sorted order only within postings, in the *Google* layout the positions are sorted for the entire list. Therefore, the p-gaps in the *Google* layout can be easily achieved by taking the differences of any two consecutive positions on the list. However, this method may increase the average values of p-gaps, especially when similar documents are not clustered before positions are assigned. For example, suppose that the term *good* appears in the current document (which is the first document in the collection) as the first word and it appears only once, while the next document also containing *good* is the 1,000,000th document in the collection, and that *good* occurs in it as the first word and appears only once. We assume the average document size is 1000 words. In this case, the p-gap between these two consecutive occurrences under the *Google* layout is $1000000 \times 1000 - 1 = 999999999$, while the p-gap under the *normal* layout is just the local position within the page, that is, 1. From another point of view, since the *Google* layout saves the storage for docIDs and frequencies by embedding them into the global positions and the special data structures for document boundaries, the compressed size of the positions is increased (which will be shown in later experimental results), although the total index size may be reduced.

It appears from [31] that the p-gaps are compressed using vbyte-like methods. However, the technical details of the compression methods under the *Google* layout are not provided.

4.5 Experimental Results

We use a similar experimental setup as we used in Chapter 3. That is, we used the TREC GOV2 data set of 25.2 million web pages and the same 1000 random queries we selected from the supplied query logs, which contain 2171 unique terms. On average, there were 4.85 million postings with 20.72 million positions in the inverted lists associated with each query. We call this data set *DS1*. In addition, we created another data set, a subset of the GOV2 data set, which contains 10% of randomly selected pages out of the 25.2 million web pages in the GOV2 data set. We call this data set *DS2*.

Throughout this chapter, we will report the compression results in different forms for the above two kinds of data sets respectively. That is, for the *DS1* data set associated with 1000 random queries, we report the compressed size of the position data per query, that is, the amount of compressed position data in MB associated with the inverted lists of an average query; for the *DS2* data set (10% GOV2 data set), we report the compressed size of positions in MB, that is, the amount of compressed position data in MB in the full inverted index built for the *DS2* data set. The former representation is a rough measure of the amount of data per query that has to be transferred from disk in the case of a purely disk-based index, under the assumption that only complete lists are transferred; the latter representation is simply the total index size of positions for the data set.

Finally, limited experiments involving decompression speed are provided in Section 4.6 in the context of a query processor that uses position data.

In Figure 4.4, we compare the average compressed size of the position data per query of various methods on the *DS1* data set. We show results for the following oblivious or list-adaptive methods: Gamma, variable-byte (vbyte), Simple9, Sim-

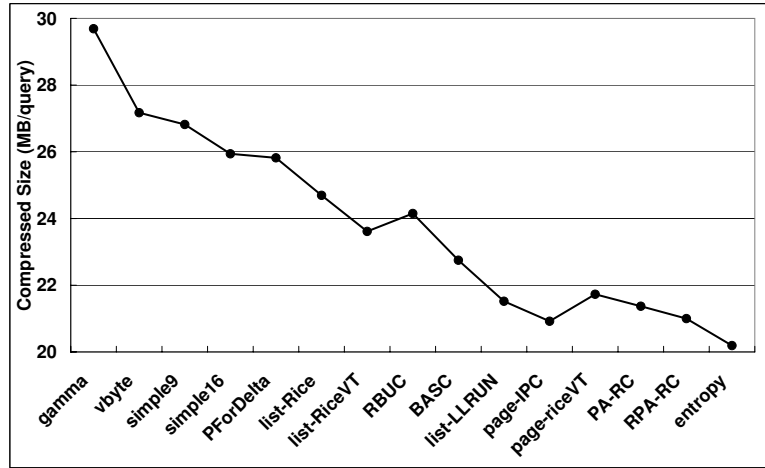


Figure 4.4: Compressed size per query for various baseline methods on the *DS1* data set.

ple16 as described in [94], the version of PForDelta described in [94], list-adaptive Rice coding (list-Rice), list-adaptive riceVT as described in [90] (list-riceVT), list-LLRUN [38] (building one Huffman table for each list), RBUC and BASC [59] (where in RBUC we choose the escalation function as $f(s) = s * s$ and where BASC is the basic version without smoothing). We also show results for four page-adaptive or fully adaptive methods: a page-oriented version of interpolative coding [60,61] (page-IPC) that is applied to the positions in each posting, a page-oriented version of riceVT (page-riceVT, which is similar to list-riceVT except that the value of b , a parameter that has the same meanings as the b in the list-Rice, is determined by the average value of positions within a page instead of in a list), PA-RC, and RPA-RC. We also show the list-wise entropy (which of course does not constitute a lower bound).

From Figure 4.4 we can see that all oblivious (non-parametric) or list-adaptive methods, including all methods except list-LLRUN to the left of page-IPC, do significantly worse than the page-adaptive methods on the right side of and including page-IPC, by 10 to 15%. Second, although list-LLRUN can achieve comparable

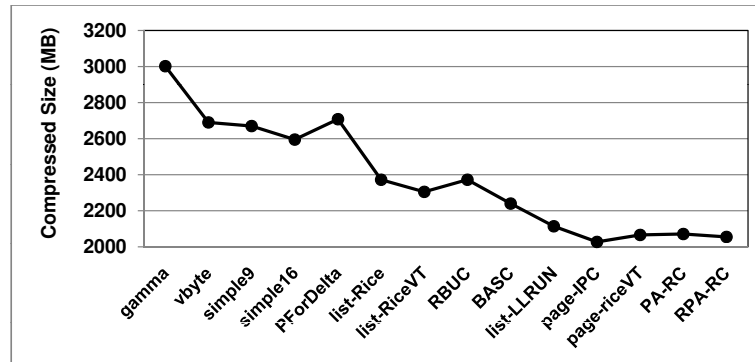


Figure 4.5: Compressed size in MB for various baseline methods on the *DS2* data set.

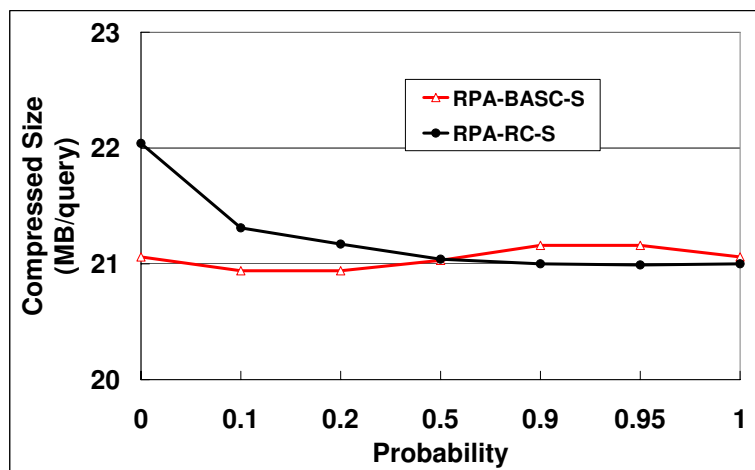


Figure 4.6: Compressed size per query for RPA-RC-S and RPA-BASC-S on the *DS1* data set.

Table 4.1: Compressed sizes (MB) for the *DS2* data set using selected compression methods under Google’s layout.

	vbyte	PForDelta	list-Rice	list-IPC
compressed size (MB)	3207	4400	3400	2899

compression performance as the page-adaptive methods, it is a semi-static method that has to first calculate the statistics information of all positions in the list before it can start encoding, while the page-adaptive methods do not need to do so. We also note that while page-wise interpolative coding (page-IPC) achieves the best result (20.92 MB/q), it is only slightly better than RPA-RC (21.00 MB/q) but slower in decompression [60, 61]. Overall, RPA-RC is a fairly simple on-line method, and performs better than all other methods in Figure 4.4 except page-IPC.

Similar experiments are performed on the *DS2* data set and the results are shown in Figure 4.5. From Figure 4.5, we can see that similar observations hold for the relative performance of various compression methods. The page-IPC method still achieves the best compression performance (2027 MB) among all methods, and RPA-RC is the second best method (2055 MB). Compared to Figure 4.5, where positions are stored in the *normal* layouts, Table 4.1 shows the compressed size of positions under the *Google* layout (for the *DS2* data set), using the following four selected methods: vbyte, PForDelta, list-Rice and list-IPC. From Table 4.1, we can see that since the positions under the Google’s index layout implicitly contain the docID and frequency information, their compressed sizes of these four methods are much larger than those shown in Figure 4.5. However, the relative performance of these methods is similar for both kinds of layouts. In the rest of this thesis, we will only focus on showing the experimental results for indexes with the *normal* layouts of positions.

In Figure 4.6 and Figure 4.7 we show the performance of the two regression models for different values of p (where $p = 0$ means using only the previous gaps,

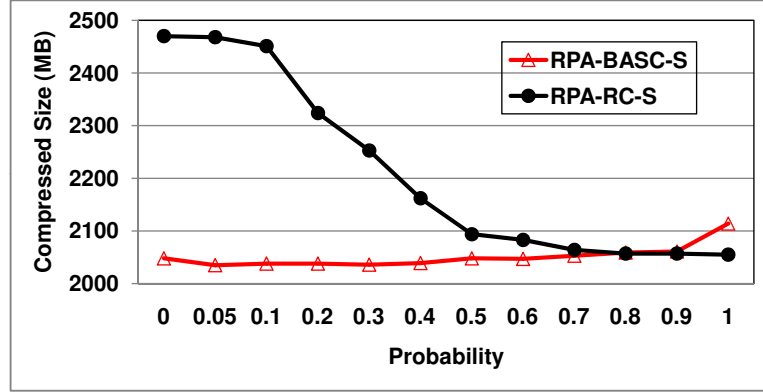


Figure 4.7: Compressed size per query for RPA-RC-S and RPA-BASC-S on the *DS2* data set.

while $p = 1$ means using only the remaining page information). From Figure 4.6, we observe that even without remaining page information, page-adaptive BASC-smoothing achieves much better compression (21.06 MB/q) than its list-adaptive version (22.75 MB/q) in Figure 4.4. Second, both models achieve their best results when using both types of information. In particular, RPA-RC-S achieves its best result (20.98 MB/q) for $p = 0.95$, while RPA-BASC-S gets its best result (20.94 MB/q) for $p = 0.2$ and $p = 0.1$. Third, the remaining average gap has more impact on RPA-RC-S than on RPA-BASC-S, while the previous average gap affects the latter more. The reason is that if the current gap to be encoded is very large while the previous average gap was fairly small, then the unary part of the Rice code for RPA-RC-S would be very large. In order to avoid this problem, RPA-RC-S exploits the remaining average gap to tune the wrong prediction from the previous gaps. Similar results are shown in Figure 4.7, especially in terms of the relative performance between page-wise methods and list-wise methods, and between RPA-RC-S and RPA-BASC-S. RPA-RC-S achieves its best result (2055 MB) for $p = 1.0$, while RPA-BASC-S gets its best result (2035 MB) for $p = 0.05$.

However, overall we see that using only the remaining-page information (without the previous gaps) is already a fairly good choice, since both methods achieve

reasonably good compression performance in this case. Thus, the benefit due to regression is only very limited. In addition, from the above, we can see that the relative performance of various methods for the *DS1* data set matches that for the *DS2* data set. Therefore, from now on we only focus on the *DS1* data set. Similar observations can be achieved on the *DS2* data set.

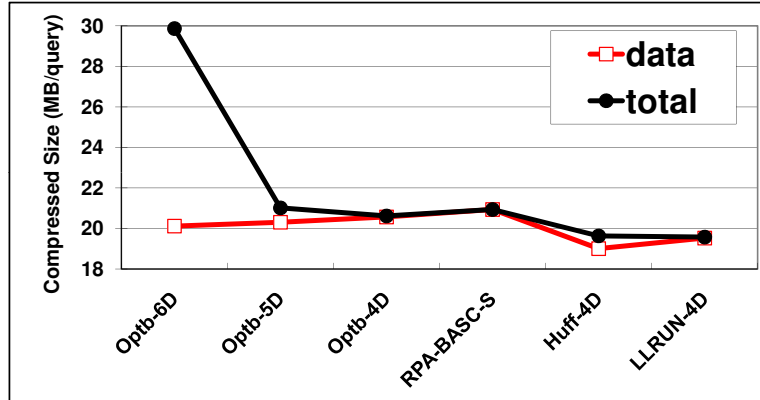


Figure 4.8: Compressed size per query for RPA-BASC-S, Optb-4D, Optb-5D, Optb-6D, Huff-4D and LLRUN-4D.

In Figure 4.8, we compare the best method from Figure 4.6, RPA-BASC-S, with Optb-4D, Optb-5D, Optb-6D, Huff-4D and LLRUN-4D. (Optb-5D and Optb-6D are variants of Optb-4D that use one and two additional previous gaps as 5th and 6th features.) We plot two lines in Figure 4.8, one for the compressed size without taking the extra cost for storing the Huffman tables or b -values for each bucket into account, and one for the compressed size including this extra cost. From Figure 4.8, we can see that although Optb-6D could get better compression if we do not consider the extra cost, in reality it is much worse. Overall, LLRUN-4D achieves the smallest compressed size among the methods, achieving about 19.58 MB per query. Huff-4D has similar performance but suffers slightly for using too many codewords in its Huffman tables. Similar experimental results can be achieved for the *DS2* data set and is not shown here.

Table 4.2: Compressed sizes (MB/q) for selected methods.

	data	extra cost	total size
list-LLRUN	21.52	N/A	21.52
page-IPC	20.92	N/A	20.92
RPA-RC-S	20.99	N/A	20.99
RPA-BASC-S	20.94	N/A	20.94
Optb-4D	20.57	0.05	20.62
Huff-4D	19.00	0.63	19.63
LLRUN-4D	19.53	0.05	19.58

Finally, we list in Table 4.2 the exact compressed sizes of the methods with the best compression performance.

4.6 Query Processing

As discussed, for typical web data the position data in the index is significantly larger (by a factor of 3 to 5) than the docID and frequency data. To minimize decompression cost, an efficient query processor should try to avoid accessing the position data for all postings in the intersection (or other Boolean filter) of the inverted lists. Instead, postings in the intersection could be first scored without taking position information into account, and then position data is fetched only for the K most promising postings, for some sufficiently large K .

Thus, while queries typically decompress substantial parts of the docID data of the inverted lists (though with some amount of skipping), accesses to position data in memory are best thought of as random accesses to individual postings. On the other hand, we still have to first fetch the complete position data for any inverted list located on disk, since random lookups are extremely inefficient with current hard disks. This fundamentally changes the trade-off between compressed size and decoding speed, in that size becomes relatively more important than speed. In this section we describe how to perform random lookups into the position data, and then evaluate the query processing performance of our compression schemes under this modified trade-off.

4.6.1 Position Look-Up Structure

To efficiently access the compressed position data associated with a particular posting, we need a suitable look-up structure. We now describe this structure for the case of Rice coding (or any other method that compresses each integer individually), and then outline how to modify the structure for methods such as PForDelta.

We employ a fairly standard hierarchical look-up structure, where position data is divided into chunks. In particular, we organize the position data for N_1 postings into one chunk, and store for each such chunk one docID and one pointer to the beginning of the chunk in uncompressed form. Within each chunk, we organize data into sub-chunks of N_2 postings each, and for each sub-chunk we store its offset from the beginning of the chunk in compressed form, using variable-byte compression. This allows us to access any posting by decompressing at most N_2 postings of position data. In particular, to find the position data for a particular docID, we first search for the right chunk using binary search on the array of uncompressed docIDs (one per chunk). Since we have already decompressed the docID data itself, we know the index of the posting within the chunk (i.e., the global index modulo N_1) and thus the correct sub-chunk, which we can then decompress. We used $N_1 = 128$ and $N_2 = 8$ in the following.

For PForDelta and other compression methods that compress batches of numbers, the look-up structure has some minor differences compared to the above. In the case of PForDelta, we compress 128 integers at a time, which does not align with posting boundaries. As a result, we need to store two rather than one compressed integer for each sub-chunk, to store an offset within a field of 128 integers. For other methods such as Simple9 that compress a variable number of integers at the same time, some other minor adjustments are needed. The first-level structure remains basically the same in either case.

Table 4.3: Space overhead and performance of the lookup structure for $K = 100$.

	PA-RC	RPA-RC	PForDelta	Optb-4D
Space per query	0.77M	0.77M	1.51M	0.77M
Space for L1	0.15M	0.15M	0.3M	0.15M
Space for L2	0.62M	0.62M	1.21M	0.62M
Decoded ints	13524	13524	43895	13524
Total time	0.63ms	0.91ms	0.31ms	1.70ms
Decode time	0.38ms	0.66ms	0.10ms	1.50ms
Seek time	0.25ms	0.25ms	0.21ms	0.20ms
Position size	21.37M	21.18M	23.67M	20.61M

Table 4.3 shows the lookup performance and size of this structure, where we perform $K = 100$ lookups each for 1000 queries. The total time per 100 lookups consists of the decoding time for the sub-chunks, plus the seek time for searching the first-level array and decompressing the second-level pointers. As we see from the results, the lookup structure adds between 0.77 and 1.51 MBs to the more than 20 MB of position data per query. We decode between 138 and 438 integers per lookup; this is since for each lookup, we need to decode an entire sub-chunk for each query term. We also see that there is a trade-off between compressed size and time, but the overall look-up time per query is at most 1.7 ms even for the method achieving the smallest size (Optb-4d). We expect some additional speed gains with proper tuning.

4.6.2 Proximity-Aware Scoring

We now look at the use of our lookup structure in the context of ranking functions such as [26, 55, 58, 73, 84] that take term proximity, and thus positions, into account. In particular, we use the scoring model proposed by Buettcher and Clarke in [26], also used in [73], which gives significant improvements in result quality over BM25-based scoring. In this model, the BM25 scoring function is combined with a proximity score for each query term that depends on how far this term is from an occurrence of some other query term. Given a query, we first compute a proximity score for each query term that depends on the distance of

Table 4.4: Percent of queries that achieve the same top- m results as an exhaustive evaluation.

	$K = 10$	20	50	100	150	200
$m = 10$	34.9%	79.3%	94.8%	97.3%	97.8%	98.2%
$m = 50$	NA	NA	27.0%	77.5%	87.8%	91.8%

Table 4.5: Percent of correct top- m results returned.

	$K = 10$	20	50	100	150	200
$m = 10$	83.7%	95.5%	98.8%	99.3%	99.4%	99.5%
$m = 50$	NA	NA	88.5%	97.2%	98.6%	99.1%

this term’s occurrences to the adjacent query term’s occurrences. The score for a document is then computed by a linear combination of the standard BM25 score and a total proximity score computed from the accumulated proximity scores.

We now show how this scoring function can be very efficiently approximated using our lookup structure as follows: We first compute the top- K results under the standard BM25 score, by accessing only docID and frequency values. Then for these K results, we fetch position information in all lists in order to compute the proximity score and thus the full ranking function. As we show, using values of K at most 200, we can compute the correct top- m results for $m = 10$ and $m = 50$ almost all the time.

The results are shown in Tables 4.4 and 4.5. In particular, Table 4.4 shows how likely we are to get exactly the same top- m results as an exhaustive evaluation, while Table 4.5 shows what percentage of returned results really belongs in the top- m . For example, using $K = 100$ we get exactly the same top-10 results as an exhaustive evaluation for 97.3% of all queries, while 99.3% of all results returned for $K = 100$ are in fact correct top-10 results. Thus, about 100 lookups per query are typically enough to match the quality of the scoring function in [26], justifying our claim that access patterns for position data can be very different from those for docIDs and frequencies.

Chapter 5

Algorithms for Low-Latency Remote File Synchronization

Recall that the file synchronization problem is to update a old version of a file, f_{old} , on a machine with the new version of the file, f_{new} , on another machine, incurring the minimum communication costs. The best-known single-round protocol is the algorithm in the widely used *rsync* open-source tool for synchronization of file systems across machines [86]. *rsync* can be considered as a delta compressor that ideally only transfers the difference between f_{old} and f_{new} over the network, although in practice it has to transfer a certain amount of extra data.

Since there are often significant similarities between successive versions of files on different machines of search engines, we would like to exploit these in order to decrease the amount of data that is transmitted and thus improve the performance of search engines, for example, by speeding up the updating of newly crawled files across multiple machines. If the new files differ only very slightly from their previous versions, then the cost should be very low, while for more significant changes, more data may have to be transmitted.

File synchronization methods are either single-round or multiple-round protocols. Multi-round protocols can provide significant bandwidth savings over the

single-round case on typical data sets [51, 83], while single-round protocols have less protocol complexity and computing and I/O overheads.

In this chapter, we focus on the single-round case. We propose a new algorithm for file synchronization based on the use of *set reconciliation* techniques [57]. The communication cost of our algorithm is often significantly smaller than that of *rsync*, especially for very similar files. However, the basic version of our algorithm assumes knowledge of some (preferably fairly tight) upper bound on the number of “differences” between the versions. To address this issue, we then study how to integrate sampling techniques into the basic protocol in order to estimate this difference, and to choose suitable approaches and parameters based on the underlying data set.

Before continuing, we point out a few assumptions in our basic setup. We assume that collections consist of unstructured files that may be modified in arbitrary ways, including insertion and deletion operations that change line and page alignments between different versions. Thus, approaches that identify changed disk blocks or bit positions or that assume fixed record boundaries do not work (though the techniques are potentially useful, e.g., for identifying which files have been changed). Note that the problem would also be easier if all update operations to the files are saved in a log that can be transmitted to the other machine, or if the machine holding the current version also has a copy of the outdated version. However, in many scenarios this is not the case. We are also not concerned with issues of consistency in between synchronization steps, or with the question of how to resolve conflicts if updates can be concurrently performed at several locations (see [12, 69] for a discussion). We assume a simple two-party scenario where it can be easily determined which file is the most current one.

5.1 Background and Related Work

In this section, we discuss in more details the state-of-the-art of file synchronization techniques. In particular, we first outline some existing single-round and multi-round protocols for file synchronization. We then provide technical background on several techniques that we employ in our approach, especially the *rsync* algorithm and tool, and a technique for *set reconciliation* and its relationship to file synchronization. Finally, we describe content-based file partitioning techniques, which have been previously employed for redundancy elimination in file systems and networks as well as for plagiarism detection and related string processing problems.

5.1.1 Single Round and Multi-Round Protocols

There are several theoretical studies of the communication complexity of the file synchronization problem that imply asymptotically tight bounds using one or more rounds [29, 65, 66]. However, the proposed optimal algorithms are not implementable in practice, as they require the receiver to invert a hash function over a large domain in order to decode the new version of the file. In practice, the most widely used protocol for file synchronization is the *rsync* algorithm [86], which is employed within a widely used open-source tool of the same name that is included in many Linux distributions. To synchronize a file, a single round of messages is exchanged, where the client sends a hash value for each block of some fixed size in the old file, which the server then uses to compress the new file that is then sent back to the client. The *rsync* algorithm does not achieve strong provable bounds on communication costs, but seems to perform well in practice.

Several authors have proposed multi-round algorithms based on a divide-and-conquer approach. The algorithms start by sending hashes for large blocks, and

then recursively divide any unmatched blocks into smaller blocks and send hashes for these, until either a block with a matching hash is found on the other side or the recursion is terminated. The earliest such algorithm [76] predates *rsync*, and subsequently a number of such algorithms have been proposed [29, 35, 51, 66, 83]. The algorithms can be efficiently implemented, and their communication costs are within a (poly)logarithmic factor of optimal under common measures of file similarity. As shown in [83], a highly optimized implementation can give significant improvements over *rsync*.

Thus, there is a trade-off between the number of communication rounds and the bandwidth consumption. Extra rounds may increase communication latencies for small files, though communication phases can be overlapped on larger collections. Moreover, multi-round protocols may require multiple passes over the data at the endpoints, and typically have a more complex control structure. In practice, single-round protocols such as *rsync* are implemented in two rounds, an initial exchange of meta data (e.g., directory data and MD5 hashes for all files), and then the actual *rsync* algorithm. Our algorithms take this form, maybe best described as “one to two rounds”.

There are two other recent protocols that try to improve on *rsync* by using only one or two rounds. Work in [46] studies several optimizations to the *rsync* approach, and also proposes a new approach to file synchronization based on erasure codes. In particular, it is shown how to simulate a basic multi-round protocol in a single round of messages, assuming an upper bound on the difference between the files.

Closely related to our approach is the protocol in [1], which is also based on the use of *set reconciliation* techniques. The *set reconciliation* problem, defined further below, was originally introduced in [57] in the context of synchronizing structured data items (e.g., database records). The algorithm in [1] is a two-

round protocol, and it achieves provable bounds with respect to certain measures, although it has several shortcomings that we think make it impractical for many common application scenarios (more details are provided later).

There is a significant amount of related work by the OS and Systems communities that attempts to detect redundancies in data in order to reduce storage or transmission costs, such as the *Low Bandwidth File System* [63], storage and backup systems [30, 33, 50, 68, 85], value-based web caching [47, 70], and compression of network traffic [81]. Most of these schemes operate on a lower level within network or storage protocols and thus have no notion of file or document versions. The approaches usually rely on content-based partitioning techniques such as [48, 74, 85], rather than the fixed-size blocks in *rsync*, in order to divide the data into blocks.

Finally, a number of researchers have used sampling to estimate similarities between files [23, 41, 85, 87]. We employ fairly standard sampling techniques, and focus on how to engineer these techniques to fit into our application.

5.1.2 The *rsync* Algorithm

The basic idea in *rsync* is to split a file into blocks and use hash functions to compute hashes or “fingerprints” of the blocks. These hashes are sent to the other machine, where the recipient attempts to find matching blocks in its own file. In the case of *rsync*, the client splits its file into disjoint blocks of some fixed size b and sends their hashes to the server. Note that due to possible misalignments between the files, it is necessary for the server to consider each window of size b in f_{new} for a possible match with a block in f_{old} . Thus the algorithm is as follows (slightly simplified):

1. At the client:

- (a) Partition f_{old} into blocks $B_i = f_{old}[ib, (i + 1)b - 1]$ of some block size b .

- (b) For each block B_i , compute a hash value $h_i = h(B_i)$ and communicate it to the server.

2. At the server:

- (a) For each received hash h_i , insert an entry (h_i, i) into a dictionary, using h_i as key.
- (b) Perform a pass through f_{new} , starting at position $j = 0$, and involving the following steps:
 - (i) Compute the hash $h(f_{new}[j, j + b - 1])$ of the block starting at j .
 - (ii) Check the dictionary for any matching hash.
 - (iii) If found, transmit the index i of the matching block in f_{old} to the client, increase j by b , and continue.
 - (iv) If none found, transmit the symbol $f_{new}[j]$ to the client, increase j by one, and continue.

There are various additional optimizations in the algorithm. All symbols and indices sent to the client in steps (iii) and (iv) are also compressed using an algorithm similar to *gzip*; this is crucial for performance in practice. A 128-bit hash on the entire file is used to detect any (fairly unlikely) collisions in the block hashes, in which case we retransfer f_{new} in compressed form. The block hash function $h()$ itself is carefully chosen such that it combines efficiency and robustness. In particular, a “rolling” hash function is used such that the hash of $f[j + 1, j + b]$ can be computed in constant time from that of $f[j, j + b - 1]$.

One critical issue in *rsync* as well as other single-round algorithms is the choice of the block size b , and the best choice depends on both number and granularity of changes between the two versions. If a single character is changed in each block of f_{old} , then no match will be found by the server and *rsync* will be completely ineffective; on the other hand, if changes are clustered in a few areas of the file, *rsync* will do well even with a large block size. Fortunately, the latter case is common in many applications. In practice, *rsync* uses a default block size of 700 bytes (except for large files), though this is clearly not the best choice for every data set.

Each block hash sent to the server in *rsync* has a size of 48 bits. Some improvements can be obtained by choosing fewer bits per hash, depending on the sizes of the files; see, e.g., [46, 83, 86]. Given two files of length n , where each hash in f_{old} is compared to the hashes of all $n - b + 1$ blocks of size b in f_{new} , we need about $\lg(n) + \lg(n/b)$ bits per hash just to have an even chance of not having a false match, while about d more bits are needed to get a probability less than $1/2^d$ of having any false match between the files.

Finally, we discuss the structure of the *rsync* open source tool, based on [88]. Strictly speaking, *rsync* involves two roundtrips. First, the client requests synchronization of a directory, and the server replies with some meta information which allows the client to decide which files need to be updated. Next, the synchronization is performed as described above. This is implemented in a highly pipelined fashion, where the two endpoints are implemented as separate independent processes that consume and produce streams of data. Thus, round-trips are not incurred on separately for each file, and it might seem reasonable to add some extra rounds to obtain additional bandwidth savings. However, this would require either a larger fixed set of consumer/producer processes, or we would have to keep some state for each file. In our approach, we adhere to this simple structure, with a first round for exchanging meta data, and one subsequent round where the actual synchronization is performed.

5.1.3 The Set Reconciliation Problem

We now discuss the *set reconciliation problem* defined in [57, 82] and its relation to file synchronization. In the *set reconciliation problem*, we have two machines A and B that each hold a set of integer values S_A and S_B , respectively. Each host needs to find out which integer values are in the intersection and which are in the union of the two sets, using a minimum amount of communication.

The goal is to use an amount of communication that is proportional to the size of the *symmetric difference* between the two sets, i.e., the number of elements in $(S_A - S_B) \cup (S_B - S_A)$. A protocol based on characteristic polynomials [57] achieves this with a single message. We define the characteristic polynomial $\chi_S(Z)$ of a set $S = \{x_1, x_2, \dots, x_n\}$ as the univariate polynomial

$$\chi_S(Z) = (Z - x_1)(Z - x_2)\dots(Z - x_n). \quad (5.1)$$

An important property of the characteristic polynomial is that it allows us to cancel out all terms corresponding to elements in $S_A \cap S_B$, by considering the ratio between the characteristic polynomials of S_A and S_B

$$\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)} = \frac{\chi_{S_A \cap S_B}(Z) \cdot \chi_{\Delta_A}(Z)}{\chi_{S_A \cap S_B}(Z) \cdot \chi_{\Delta_B}(Z)} = \frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}, \quad (5.2)$$

where $\Delta_A = S_A - S_B$ and $\Delta_B = S_B - S_A$. In order to determine the set of integers held by the other party, it suffices to determine the ratio of the two polynomials. As observed in [57], if we know the results of evaluating both polynomials at only k *evaluation points*, where k is the size of the symmetric difference, then we can determine the coefficients of $\chi_{\Delta_A}(Z)/\chi_{\Delta_B}(Z)$. By factoring $\chi_{\Delta_A}(Z)$ and $\chi_{\Delta_B}(Z)$ we can recover the elements of Δ_A and Δ_B . Thus, if the difference between the two sets is small, then only a small amount of data has to be communicated.

This problem was initially studied in the context of synchronizing structured data items. The idea is that by representing each data record by an integer hash, we can determine which records already exist at the recipient. If each hash value consists of x bits, then all arithmetic can be performed modulo 2^x , and thus each evaluation point can be communicated in x bits. If a record does not exist at the recipient, then the entire record is transmitted. However, in file synchronization, we want to exploit similarities between different versions, even if the file has

changed to some degree.

Recent work in [1] proposed a new algorithm for file synchronization, called *reconciliation puzzles*, that uses set reconciliation as a main ingredient. Each machine converts its file (string) into a multi-set of overlapping pieces, where each piece is created at every offset of the file according to a predetermined mask. The hosts also create a modified de Bruijn digraph to enable decoding of the original string from the multi-set of pieces: The correct Eulerian path on this digraph determines the ordering of the pieces in the original string, and thus the original string if we know all the pieces.

To reconstruct f_{new} , the client needs to know all the pieces of the file f_{new} , plus the Eulerian path on the digraph. Both sides transform their sets of pieces into integer values by concatenating each piece with the number of times it occurs and hashing the resulting string. Then the set reconciliation protocol from [57] is used to reconcile the sets of hash values, at which point the server knows which pieces are unknown to the client and thus need to be transmitted. If the two versions are very similar, then they will have most of their hash values in common, and the set reconciliation algorithm only needs to transmit a very small amount of data. Finally, a compact description of the Eulerian path is transmitted to the client together with the missing pieces, resulting in a two-round protocol.

5.1.4 Content-Dependent File Partitioning

Using a fixed block size as in *rsync* has disadvantages, since the recipient needs to compute hashes for all possible alignments in its file to find the largest match. This requires not only more computation, but also more bits in each hash. In the set reconciliation approach, the files cannot be simply partitioned into disjoint blocks for the reconciliation step; chances are that the resulting sets would be almost disjoint due to different alignments of the common content in both versions.

In [1], this problem is avoided by creating many more blocks (pieces) from each file by considering every possible alignment. However, this significantly increases the size of the resulting sets and graph, resulting in extra cost.

We would like a partitioning that allows us to find many common blocks in similar files and that can be locally applied to each file. One such technique, based on Karp-Rabin fingerprints [48], has been extensively used to identify redundant data during storage or transmission [30, 47, 50, 63, 70, 85]. The basic idea is very simple: We hash all substrings of some fixed length c , say $c = 20$, to integer values, and define a block boundary before position i of the file f if the substring $f[i, i + c - 1]$ hashes to a value $s \bmod w$, say for $w = 200$ and some s . Then we use the blocks defined by these boundaries, which have an average length of about w (under certain assumptions). Since boundaries are chosen in a local manner, any large substrings common to both files are partitioned in the same way and result in identical blocks. After identifying the boundaries using the above method, we then hash the resulting blocks to integers as part of the synchronization protocol.

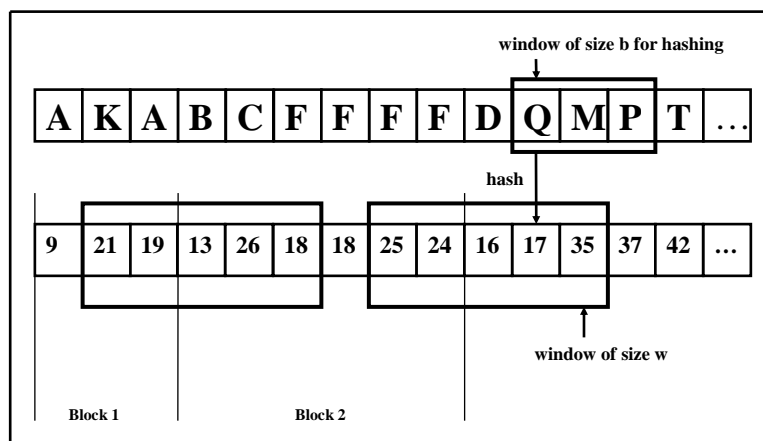


Figure 5.1: Example of the file partitioning technique in [85]. We are hashing windows of size $c = 3$, and then selecting block boundaries based on these hashes for $w = 2$.

More recently, several new partitioning techniques have been proposed, includ-

ing the *winnowing* technique in [74], and the approach in [85] which we refer to as *2-way min*. We focus on the latter technique, defined as follows: We again hash all substrings of some fixed length c to integer values. We now define a block boundary before position i of the file f if the hash $h(f[i, i + c - 1])$ associated with position i is strictly smaller than the hash values of the w preceding and following positions. In other words, we choose those positions where the hash assumes a local minimum. The method results in an average block size of $2w$ and is illustrated in Figure 5.1.

Most applications, such as [30,63,70], use the older method to select the boundaries. However, analysis in [85] and unpublished experiments indicate the methods in [74,85] outperform this method in redundancy detection.¹ To verify this expectation, we compared the three methods in terms of their tradeoff between average block size and the amount of redundancy (size of identical blocks of data) that is detected between the two versions. The results are shown in Figure 5.2.

Clearly, as block size decreases all methods can identify more redundancy, but a small block size requires more hashes to be sent from server to client. In Figure 5.2, we observe that the *2-way min* approach does better than *winnowing*, and *winnowing* does better than the above basic approach, in terms of identifying redundancies across files. Some analytical support for this conclusion is also provided in [85]. Overall, fixed block size does best, as should be expected, but note that we need more bits per hash value in this case, which eliminates the advantage. We also experimented with other data sets, and with several other methods not described here that performed almost as well as *2-way min*, but none did better.

Thus, we focus on the *2-way min* method in our experiments, but any of the others would also work at slightly decreased performance. Note that these methods can be implemented at speeds of tens to hundreds of MB per second

¹Note that a claim in [46] that the winnowing method in [74] does not improve on the older method is incorrect and due to a bug in the implementation.

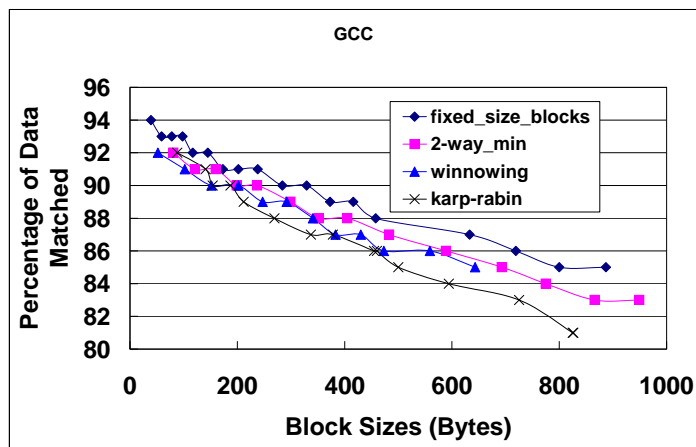


Figure 5.2: Comparison of different partitioning techniques on various block sizes for *gcc*. On the *x*-axis we show the average block size of the resulting blocks.

through careful optimization of the hash computation [15].

We also experimented with two different types of blocks once the boundaries are identified: Non-overlapping blocks, and overlapping blocks where each block is extended c characters into the next block (putting the window with the locally minimal hash into the intersection). Blocks with significant overlap are more suitable for use with set reconciliation techniques, since each block can only be followed by another block that “fits”, i.e., that starts with the last c characters of the previous block. In terms of the algorithm in [1], this implies that the resulting digraph is very sparse, since only few puzzle pieces fit to each other, and has few Eulerian paths that can thus be encoded very compactly.

5.2 An Algorithm Using Set Reconciliation

We now present our new algorithm based on set reconciliation. We limit ourselves to a single round of messages between client and server, and measure the communication cost in terms of the total number of bits exchanged between the parties. We assume that the client has knowledge of (an upper bound on) the size of the symmetric difference between the sets of hashes on both sides; we address this assumption in the next section.

The main idea of our algorithm is as follows: We locally partition both versions of the file into overlapping blocks using the *2-way min* technique, and represent the blocks by their hashes. We then use a set reconciliation protocol consisting of a single message from client to server, such that the server knows which of the blocks in f_{new} are already known to the client. Then the server transmits f_{new} to the client in two parts: Blocks not known to the client are encoded using a compression algorithm similar to *gzip*, while the information about the ordering of blocks within the new file is communicated in an optimized manner that exploits the fact that for each block there is usually only a very small number (often just one) of other blocks that can follow this block (i.e., that start with exactly the right characters). Here are the details:

0. At both server and client:

- (a) Use *2-way min* to partition the local file into a number of blocks, and compute a hash for each block. Let S_c and S_s be the sets of hashes at the client and server, respectively.

1. At the client:

- (a) Let d be the symmetric difference between the two sets of hashes. Use the set reconciliation algorithms described in [57] to evaluate the characteristic polynomial on S_c on d randomly selected points, and transmit the results to the server.

2. At the server:

- (a) Use the d evaluations to calculate the symmetric difference between S_c and S_s , i.e., the hashes in $S_c - S_s$ and $S_s - S_c$.
- (b) The server goes through f_{new} to identify all blocks that are not known by the client. Any two consecutive blocks not known to the client are merged into one.
- (c) The server now sends to the client the following information in suitably encoded form:
 - (i) the number of blocks in f_{new} ,
 - (ii) a bit vector specifying which of the hashes of f_{old} (sorted by value) also exist in f_{new} ,
 - (iii) a bit vector specifying which of the blocks in f_{new} (sorted by position) also exist in f_{old} ,
 - (iv) the lengths of all blocks in f_{new} that are not in f_{old} ,
 - (v) the interiors of these blocks themselves in suitably coded form, and
 - (vi) an encoding of the sequence of matched blocks in f_{new} .

Recall that in the case of fixed block size, we need hash values of about $\lg(n) + \lg(n/b) + k$ bits [46] in order to keep the chance of a collision below $1/2^k$, where n is the file size and b is the (average) block size. However, in the case of a content-dependent partitioning such as *2-way min* we only need $2\lg(n/b) + k$ bits, since each hash is only compared to n/b blocks on the other side. Thus, we use $2\lg(n/b) + k$ bits per hash in the set reconciliation protocol. Because fewer bits per hash are needed than in the fixed block-size case, this in turn allows profitable use of smaller block sizes. Moreover, the significant overlap between blocks makes it highly efficient to code the ordering of the blocks, since each block can only be followed by another block that starts with the last c characters of the previous block. Overlapping blocks also assure that only the interior of unmatched blocks must be sent.

5.2.1 Details on Encoding and Decoding

Now a few more details on the precise encoding of the various items. The two bit vectors sent from server to client can be compressed using arithmetic coding, though they are fairly compact already. The lengths of the unmatched blocks in f_{new} are Golomb-coded [90], while the interior parts of the unmatched blocks are concatenated together and compressed using a *gzip*-like algorithm similar to the one employed in *rsync*. The sequence of matched blocks is coded as follows: If the previous block was not matched, then $\lg_2(x)$ bits are used to specify the next block, where x is the total number of matched blocks. Else $\lg_2(y)$ bits are used, where y is the number of possible blocks that can follow the previous one. In many cases, $y = 1$, and there is a unique continuation.

Finally, the client reconstructs f_{new} by determining which blocks exist in both files, and determining for each such block which other such blocks can follow it. Then the new blocks in f_{new} unknown to the client are decoded, and finally the matched and new blocks are assembled into f_{new} based on the other information received.

5.2.2 A Variant Based on Golomb Coding

Note that set reconciliation is used in our algorithm simply as an efficient way to transmit a set of hashes from client to server, and that we could easily replace it by another method without changing anything else. In particular, we propose as an alternative to simply sort all the client hashes and then send them to the server by Golomb-coding (see [90]) the gaps between hash values, resulting in additional compression. As we will see later, set reconciliation is a good method for transmitting the hashes when the two files are very similar, but not suitable for files with significant differences.

5.2.3 Comparison to Previous Work

While our approach is based on very similar ideas as the algorithm in [1], there are several important differences. First, our method uses a single round, compared to two rounds used in [1]. Second, we use content-dependent partitioning techniques, instead of fixed masks, to partition files into blocks. This has significant benefits since it means that every edit operation (difference) between the two files can only impact a constant number (often only one) of blocks and block hashes. In contrast, in [1] the number of impacted blocks is proportional to the mask size, which is usually chosen as $\lg n$. As we discuss later, this in fact results in an asymptotic difference under certain assumptions. Third, the overall structure of our method is quite similar to that of *rsync*. This allows an implementation similar to *rsync* as a set of simple stateless communicating processes. In fact, our method can be seen as a unified approach that combines *rsync* and the approach in [1].

There are also several more minor differences. Because of the similarity to *rsync*, we have the choice of using reconciliation or Golomb coding for the hashes, depending on the characteristics of the data. Our method also includes compression for unmatched literals; in our experience this makes a very significant difference in practice on many data sets. Finally, we use a simpler approach for specifying and coding the sequence of blocks (i.e., the Eulerian path in [1]) whose running time does not depend significantly on the number of paths in the graph. Note that the actual number of round-trips of course depends on the set reconciliation protocol that is used. A recent recursive protocol in [56] gives a significant reduction in computational cost at the receiver at the cost of additional communication rounds, when compared to the basic single-message approach in [57].

5.2.4 Communication Complexity

We now discuss the asymptotic communication complexity of our algorithm and compare it to that of other approaches. In the discussion, we assume two files f_{new} and f_{old} of length n each with edit distance k , where the allowed edit operations are change, insertion, and deletion of single characters and moves of blocks of characters. (More general edit distance measures also allow copies and deletions of blocks, but these are tricky to analyze as distances between pairs of files are not symmetric.) There is a well known lower bound of $\Omega(k \lg n)$ bits of communication in the worst case, and a matching upper bound was established in [65]. However, that algorithm cannot be implemented efficiently in practice as it requires the inversion of certain hash functions at the receiver. The practical multi-round algorithms for synchronization in [29, 51, 66, 76, 83] achieve a communication complexity of $O(k \lg n \lg(n/k))$ bits using $\lg(n/k)$ communication rounds. A recent result in [46] matches this bound with a single round. Thus, there remains a logarithmic gap between the lower bound and the best practical protocols.

The analysis of the algorithm based on set reconciliation in [1] implies a bound of $O(k \lg^2 n)$ bits with two round-trips for average-case files, i.e., the case where one file is created at random while the other file is any file of edit distance at most k . (This is for the case of a mask length of $\Theta(\lg n)$.) Our algorithm, in contrast, can be shown to achieve a bound of $O(k \lg n)$ bits in a single round on such average-case files; this is due to our use of content-dependent partitioning techniques to define overlapping blocks – as a result each edit operation only affects a constant number of blocks. We note that this type of average-case analysis is somewhat problematic as many string processing problems are much easier on randomly generated strings (which are unlikely to have any large repeated substrings). Thus, we do not give further details here as this result is only of limited theoretical interest. A practical algorithm with the same bound in the worst case, on the other hand, would be a

major result.

5.2.5 Preliminary Experimental Results

We now present some preliminary experimental results that show the potential of the new algorithm. For the experiments, we used the *gcc* and *emacs* data sets also used in [45, 46, 83], consisting of versions 2.7.0 and 2.7.1 of *gcc* and versions 19.28 and 19.29 of *emacs*. The newer versions of *gcc* and *emacs* consist of 1002 and 1286 files, and each collection has a size of around 27 MB. We also ran additional experiments on the *html* data sets that were used in [83], consisting of a set of ten thousand pages crawled randomly from the web, and the same pages recrawled two days later. Each set of pages has a total size of around 140MB, with about 14KB per page on average. Some of the files are not updated at all between crawls, while others change only slightly. For each data set, we measured the cost of updating all files in the older version to the newer one. We assume that only files that have changed are updated, while unchanged files are detected through an MD5 hash on the entire file that is exchanged before the algorithm, at a cost of 16 bytes per file. We also assume at first that we know the upper bound of the amount of changes between old and new files, especially when we use set-reconciliation techniques.

In Figure 5.3 and 5.4, we compare the performance of the following five approaches on two data sets, *gcc* and *html*: The algorithm based on set-reconciliation, the variant based on Golomb-coded hashes in Subsection 5.2.2, a combination of these two that magically always chooses the best of the two depending on the data, *rsync*, and an optimization of *rsync* from [46] called FBS which was shown to always (at least slightly) outperform *rsync* through various minor improvements. For each algorithm, we did multiple runs with different block sizes, and plot the average resulting block size versus the total communication cost.

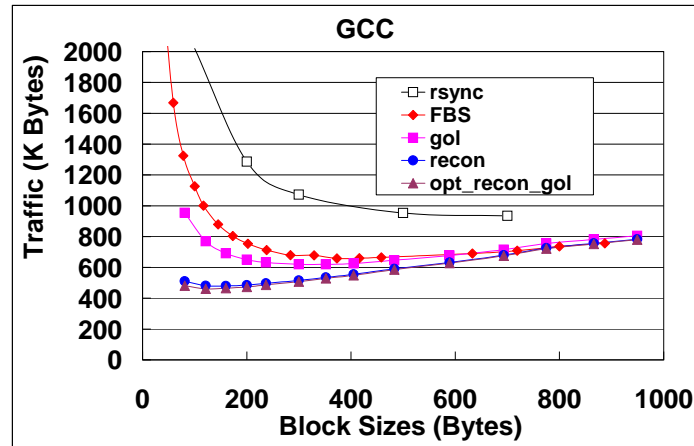


Figure 5.3: Comparison of algorithms on the *gcc* data set. The graphs from top (worst) to bottom are rsync, FBS, Golomb, reconciliation, and the optimal combination of the last two.

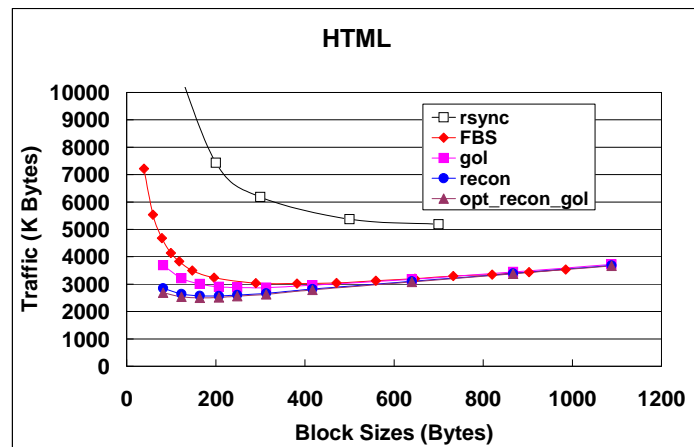


Figure 5.4: Comparison of algorithms on the *html* data set. The graphs from top (worst) to bottom are rsync, FBS, Golomb, reconciliation, and the optimal combination of the last two.

We see that for both *gcc* and *html*, the approach based on reconciliation does significantly better than the Golomb-based approach, which itself outperforms FBS. Moreover, the new methods achieve their optimum at smaller block sizes than FBS, as they communicate fewer bits per hash. In particular, Golomb saves over FBS both by using fewer bits per hash, and by sorting and Golomb-coding the hashes before transmission. However, set reconciliation is better than Golomb on almost all files, and thus the approach that chooses the best of the two does only marginally better than always using set reconciliation. In total, the best approach for *gcc* needs about 460 KB, compared to about 760 KB for FBS; and the best approach for *html* needs about 2500 KB, compared to about 3018 KB for FBS. Thus, the experiments show that our basic algorithm performs very well on fairly similar data sets.

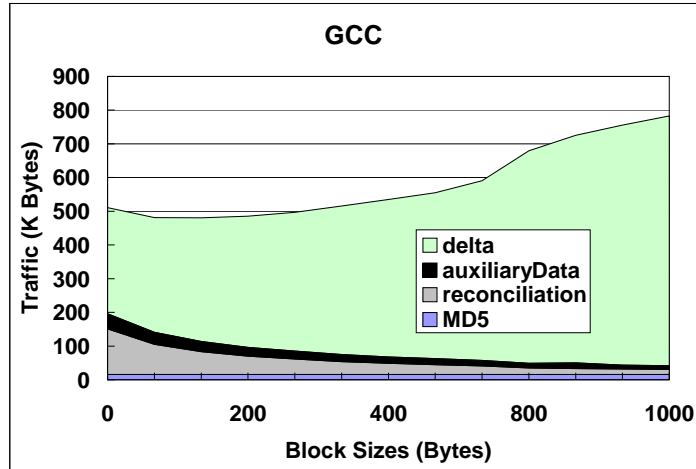


Figure 5.5: Costs of the different data structures transmitted in the reconciliation-based protocol, for *gcc*.

Next, in Figure 5.5, we look at the various contributions to the overall trans-

mission cost, for the *gcc* data. Most of the cost is due to the transmission of blocks in f_{new} that do not appear in f_{old} (called *delta* in the figure). This cost of course increases with the block size. On the other hand, the cost of sending the evaluations in the actual set reconciliation operation increases for smaller blocks. The other data structures sent from server to client, i.e., the bit vectors, block lengths, and the encoding of the Eulerian path segments, make up only a small part of the overall cost (labeled as *auxiliary*), while the cost of the MD5 hash of each file is negligible.

5.2.6 Effect of Similarity on Performance

We expect the reconciliation-based algorithm to significantly outperform other methods on fairly similar files, but less on files with more significant changes. We verified this conjecture by creating artificial data sets with varying degrees of similarity, by *morphing* the *gcc* data with other unrelated data through a simple Markovian copy process. The results, shown in Figure 5.6, clearly show that the relative advantage of reconciliation over Golomb coding is most significant when the files are fairly similar.

However, the situation is different on data sets with fairly different files, e.g., *emacs*, where the two versions are substantially less similar than for *gcc* or *html*. In this case, shown in Figure 5.7, the FBS algorithm wins out against both Golomb and reconciliation, and even does slightly better than the optimal combination of the other two for the best block size (about 3120 KB vs. 3200 KB). Note that in the case of *emacs*, fairly small block sizes are best for all algorithms, since larger block sizes are not sufficient for identifying the limited amount of redundancy between the two sets. Also, Golomb is on average slightly better than reconciliation (as expected from the previous figure).

To summarize, the results indicate that the new approach has significant po-

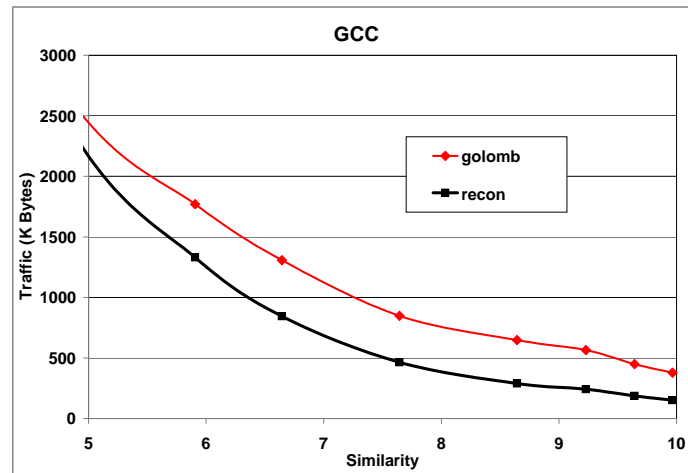


Figure 5.6: Comparison of Golomb and reconciliation methods for data with varying degrees of similarity. A value of k on the x-axis means that about a $1/2^k$ fraction of the content of each file has been changed (where each changed region has an expected length of 5 characters).

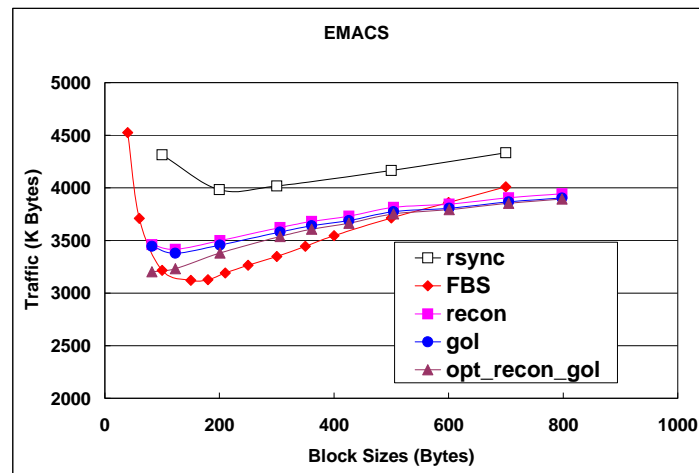


Figure 5.7: Comparison of different algorithms on *emacs* data. The graphs from top (worst) to bottom are for rsync, reconciliation, Golomb, the optimal combination, and FBS.

tential for collections with fairly similar versions. However, to make the algorithm practical, we still need to find a way to estimate the symmetric difference, and ideally also to adaptively choose either the Golomb or set reconciliation strategy based on the underlying data.

5.3 Integrating Sampling into the Algorithm

Recall that in the previous section, we assumed that the client knows a good upper bound on the size of the symmetric difference between the two sets of hashes. However, such information is usually not available in practice. In order to overcome this problem, we propose to use random sampling to estimate the symmetric difference. This can be done during the first exchange of meta data in the *rsync* framework, and it also allows us to choose either set reconciliation or Golomb coding depending on the degree of similarity between the files.

5.3.1 A More Practical Algorithm

Our goal is to estimate the symmetric difference d between the sets of hashes of f_{old} and f_{new} by exchanging suitable samples before running the algorithm. We note that this can be done without increasing the number of round-trips within an *rsync*-like structure, by sending samples from server to client (instead of from client to server) during the first exchange of meta data (such as MD5 and directory info). This allows the client to estimate the number of evaluations that have to be sent to the server, at some cost in bandwidth.

Our goal is not merely to estimate the difference d , but to provide an upper bound that is correct with fairly high probability. If our estimate for d is too low, then the server will be unable to derive the set difference from the d evaluation points sent by the client; in this case we assume that the server sends the entire f_{new} in compressed form resulting in a significant extra cost. Thus, we cannot

use a “best estimate” for d but need to pad this such that the probability of an underestimate is very small, say less than 0.1%. The important point here is that a larger sample size can decrease costs by allowing us to provide a much tighter upper bound for d than a very small sample.

We choose as our sample the subset of those hashes that are $x \bmod 2^y$ for some x and y . We can decrease the transmission cost of the sample by sending fewer bits per sampled hash. In particular, we do not have to send the y bits used to select the sample (since the other side will only compare the sample to its own hashes $x \bmod 2^y$), and we can further reduce the cost of each hash by removing additional bits and later correcting for the expected number of resulting collisions. (Note that this is somewhat similar to using a compressed Bloom filter for this purpose.)

In the end, we found that using about $\lg(S) + 8$ bits per sample element performed best, where S is the number of samples. The samples were then sorted and again Golomb coded, for a net cost of slightly more than 9 bits per sample element. Sample size was chosen based on the number of blocks in the file. For files of size less than a few KB, no sampling was done, and the Golomb-based algorithm was used instead of set reconciliation. Since samples are sent together with the MD5 and directory information, they are sent even for those files that were completely unchanged. (We also experimented with an alternative technique for estimate the difference between the sets based on a reduction to Hamming distance as described in [29], but the observed differences were small and thus we omit the results from this thesis.)

Once we have an estimate for d , we can also use this in order to decide whether to use reconciliation or Golomb coding to transmit the hashes. While such a hybrid approach did not seem to give much benefit in the ideal case considered in the previous section, it is actually more useful now. The reason is that the required

padding for the estimated file distance d increases the cost of the reconciliation-based approach, and sometimes it is now better to use Golomb coding (which does not rely on d) where with exact knowledge of d we would choose reconciliation.

In Figure 5.8, we compare the following four methods on the *gcc* data sets: (1) FBS with no sampling, (2) the idealized algorithm from the previous section that “knows” d without sampling and chooses the best of Golomb and set reconciliation, (3) an algorithm that tries to make this choice based on sampling, and (4) an algorithm that always uses set reconciliation (except for very small files) and that uses sampling only to select d . Both sampling-based methods do significantly better than FBS, though not quite as well as the idealized algorithm due to the overheads of sampling and padding. Overall, the best method with sampling achieves about 533 KB, versus 760 KB for FBS.

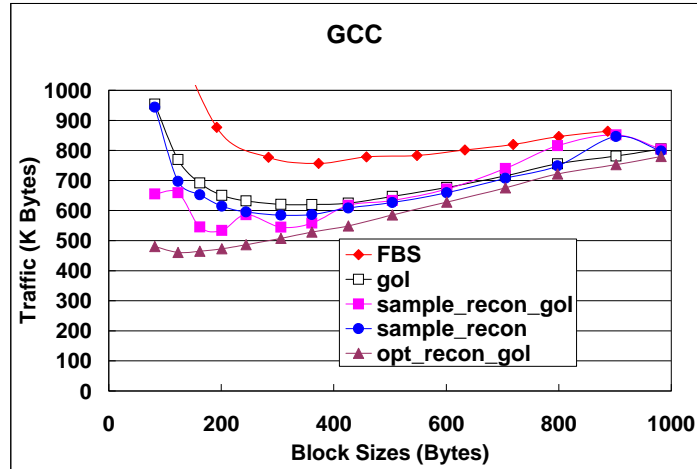


Figure 5.8: Performance of sampling-based methods on *gcc*.

In Figure 5.9, we see the contributions of the various data structures to the overall transmission cost for three block sizes and three of the methods (excluding

FBS, whose internals we have not introduced here). We note that the cost of the sampling itself is fairly low, while the cost due to padding is more significant. Note that in the two methods that can choose from either Golomb or reconciliation, the cost of the evaluations for the best estimates for d (i.e., excluding padding) is very small² and hardly visible (the second color from the top in the figure).

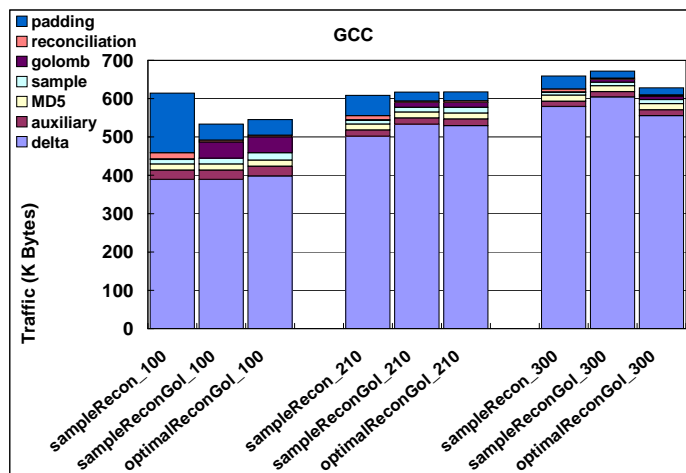


Figure 5.9: Costs of various parts of the three algorithms on *gcc*, for w values of 100, 210, and 300, with resulting average block sizes of 199, 405, and 589, respectively. (Colors in the charts are in the same order.)

For *emacs*, which is not shown here, we observed that the method that uses sampling plus the best of Golomb or reconciliation does somewhat worse than FBS (3361 KB vs. 3120 KB for FBS), as expected.

²This is not the same as the cost for reconciliation in the ideal method in Figure 5.5, since we are more likely to choose reconciliation over Golomb in cases where we underestimate the difference between the hash sets.

5.3.2 A Hybrid Method with FBS

From the above experiments, we can see that FBS sometimes performs better than methods based on variable block sizes on fairly different files. To address this problem, we investigated a hybrid method that combines set reconciliation, Golomb, and FBS. In particular, using our sampling-based estimate of the similarity between the two files, the new hybrid applies FBS on files with similarity below some threshold, and chooses between Golomb and reconciliation otherwise. In Figure 5.10, we compare this new hybrid to the previous methods as well as to an idealized version of the new hybrid that always chooses the best of the methods without sampling.

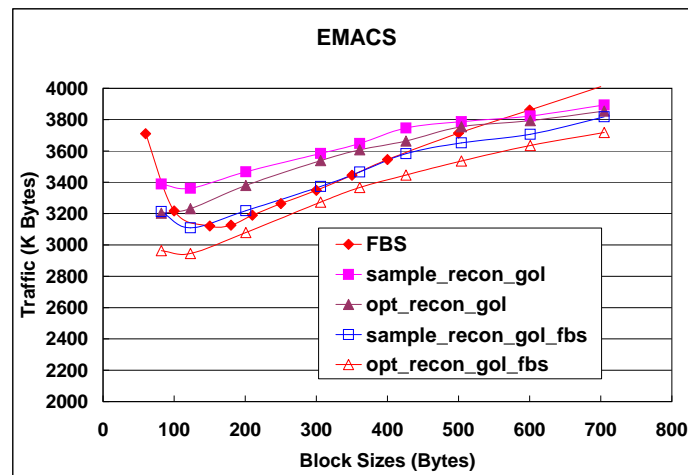


Figure 5.10: A hybrid algorithms with FBS on *emacs* data.

From Figure 5.10, we observe that the new hybrid algorithm with sampling does only very slightly better than FBS, while the idealized version of it does significantly better. On the other hand, when we ran experiments on *gcc*, adding

FBS into the hybrid made almost no difference at all both in the idealistic and sampling-based case; this is because in *gcc* most files are very similar and thus FBS is almost never a good choice. However, this does not mean that the new hybrid is not useful. In fact, the point here is that this hybrid performs well over many different types of data sets, matching the best method in each particular case. Also, results for the idealized case on *emacs* indicate that additional benefits could be achieved through improved sampling techniques.

5.3.3 Estimating a Good Block Size

One problem with single-round synchronization algorithms is that performance depends on the choice of the block size. It would be nice if we could also use our sample in order to select a good block size; unfortunately this sounds easier than it is. We experimented with several approaches for this. There are two main problems: First, we may need to sample hashes for several block sizes in order to select the best one, increasing the sample size. Second, to choose a good block size, it does not suffice to estimate the number of blocks that are in common. Instead, we also need to know the compressibility of the unmatched parts of f_{new} , since there is a trade-off between the size and number of blocks and the size of the unmatched parts of the file. Our results indicate that this compressibility varies widely within our data sets, from a factor of 2 to more than 10, which means that simply dividing the size of the unmatched parts by an “average compression factor” of, say, 4 for *gzip* methods does not give a good cost estimate. Note that this problem of choosing a good block size is not unique to our new methods, but also exists in *rsync* and all previous single-round methods.

5.4 A Two-Level Approach

In this section, we explore an additional optimization of our approach that manages to use two different block sizes without incurring additional rounds of communication, resulting in further gains.

Research on multi-round algorithms has shown significant benefits even from using just a few rounds with different block sizes [51, 64, 83]. In our case, we do not allow additional communication rounds, but would still like to get some of the benefit. The idea is again very simple: in addition to sending the MD5 and samples, we have the server also send hashes for fairly large blocks in f_{new} (say, of size 1 KB or more) in Golomb-encoded form. Then the client sends hashes for smaller blocks to the server, but only for those parts of f_{old} that are not matched by any of the large hashes. We would expect that this significantly decreases the cost of the small hashes, thus allowing the client to use a smaller block size for the next phase, which then leads to more redundancy being eliminated. Note that *2-way min* partitionings are by definition hierarchical in the sense that a boundary for one value of the window size w is also a boundary for any $w' < w$.

In our experiments, we consider the following algorithms: (1) a Golomb-only solution where the server sends hashes for large blocks and then the client sends hashes for small blocks that are not covered, without using set reconciliation, (2) a version based on sampling where the client decides to use either Golomb coding or set reconciliation to transmit the smaller hashes, and (3) an idealized algorithm without sampling where the client somehow knows the precise set difference between the hashes for the parts of the two files not covered by any large match and then chooses either Golomb or set reconciliation,

In Figure 5.11 we show results for the first algorithm on *gcc*. We see that results are best for a large block size of around 900 and a smaller block size of

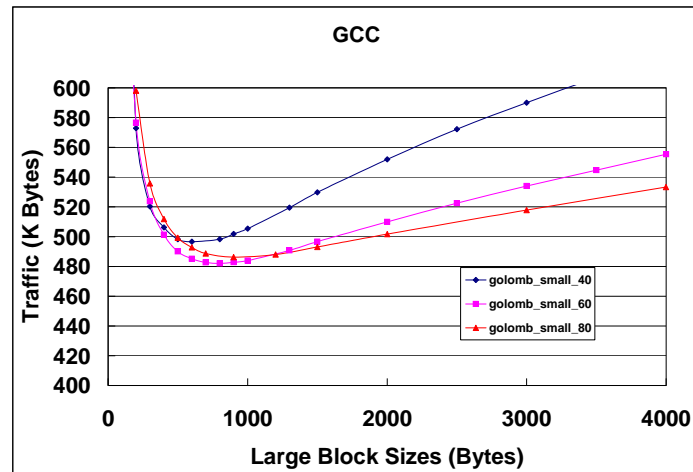


Figure 5.11: Using two different block sizes on the *gcc* data set, without sampling and reconciliation. On the x -axis we show the average size of the larger blocks, and on the y -axis the total communication cost. We look at three different settings of w for determining the smaller blocks, 40, 60, and 80, with resulting average block sizes of 82, 121, and 160, respectively.

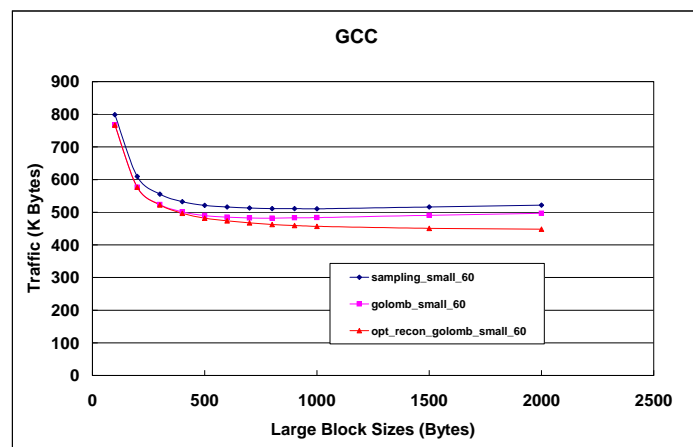


Figure 5.12: Comparison of the Golomb-only method, the sampling-based method, and an idealized method for $w = 60$. Even the idealized method cannot do better than about 450 KB, while the best sampling-based method achieves a cost of about 510 KB.

around 120 ($w = 60$), where we achieve a cost of about 482 KB versus 533 KB for the best previous method. Next, we fix $w = 60$ for the smaller blocks, and consider the two other algorithms as well; the result is shown in Figure 5.12. Note that even the idealized method cannot do better than about 450 KB, while the practical sampling-based method is actually worse than the Golomb-based approach. This is because we now only exchange hashes for those small blocks that are not contained in a large matched block; the symmetric difference between these sets of hashes is relatively much larger than for the entire file. To put it another way, the sending of large blocks decreases the cost of using Golomb in the second phase while the cost of using set reconciliation stays the same because the elements in the symmetric difference of the small-block hashes are by definition disjoint from the matched large blocks. Thus, we pay the cost of sampling, but rarely get any benefit from it. Hence, our best method here uses Golomb coding only, with savings of about 280 KB (more than 30%) over the best value of FBS (480 KB vs. 760 KB).

We also provide results for the *emacs* data set in Figure 5.13, which shows that the Golomb-based method with two block sizes also provides a moderate improvement in performance on this data set, from about 3250 KB for FBS to around 3170 KB for the best method, which in this case uses $w = 40$ for the smaller blocks.

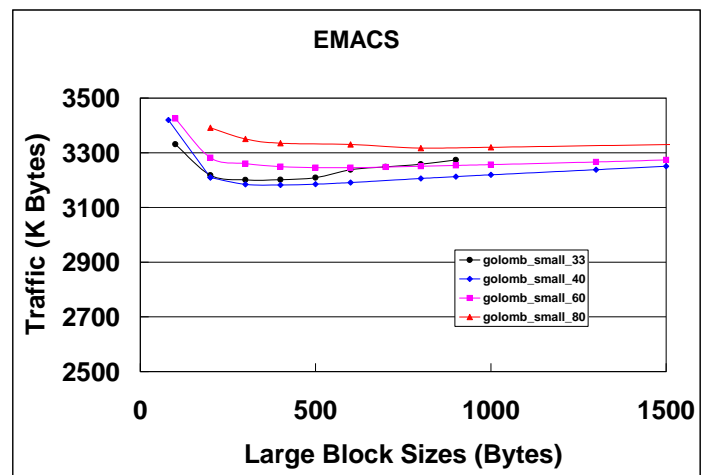


Figure 5.13: Using two different block sizes on the *emacs* data set.

Chapter 6

Conclusions and Future Work

6.1 Concluding Remarks

In this thesis, we have studied inverted index compression and file synchronization techniques to eliminate data redundancy in large textual collections and especially in search engines.

First, we studied the compression of docIDs and frequencies and the efficiency of query processing using an optimized document ordering. Previous work focused on finding document orderings that minimize index size under standard compression schemes, while we focus on how to tune compression schemes and maximize query throughput given a good ordering. Our experimental results show that our methods can achieve significant benefits in compressed index size and query throughput. We believe that the lessons learned from this work are as follows: First, although our experiments are based on the particular reordering technique, sorting documents by their URLs, our compression methods are orthogonal to other reordering techniques and thus can be integrated with them to improve compression as long as the reordering techniques cluster documents with similar contents. Second, context-sensitive methods can take better advantage of the clustering property than context-free methods. Third, one interesting result of our

experiments is that reordering of documents, in addition to improving compression, also speeds up index traversal in a DAAT query processor. In particular, our query processor (with no changes in the software, and independent of compression method) performs more large forward skips during index access in the reordered case, and as a result decompresses less than half as many blocks per query as in the unordered case.

Second, we studied compression techniques for position data in web indexes. We proposed two simple but effective techniques, Remaining Page-Adaptive Rice Coding with Smoothing (RPA-RC-S) and Remaining Page-Adaptive BASC with Smoothing (RPA-BASC-S). We also proposed several statistics-based methods (Optb-4d, Huff-4d, and LLRUN-4d) and show that they achieve even better compression performance. Finally, we studied the efficient use of position information during query execution. We believe that the lessons learned from this work are as follows: First, word positions in web pages do not seem to follow simple distributions that could be easily exploited. Second, additional context, such as document size, frequency, and nearby previous gaps, is highly useful, but there is a trade-off between the benefits of more features and the cost of storing more complex models. Third, during query processing, access to position data should be performed in a second stage after traversing the lists of docIDs such that only a limited amount of position data is retrieved; in this case, a small compressed size may be more important than extremely fast access.

Finally, we described and evaluated new algorithms for remote file synchronization that use a single communication round. We also proposed and explored the use of random sampling techniques for further optimizations. Our results show that significant improvements in communication costs can be achieved in many cases, as long as the files are quite similar. We believe that the lessons learned from this work are as follows: First, while earlier approaches [46] based on content-

dependent block partitioning did not do as well as the fixed-block partitioning in *rsync*, the newer partitioning techniques in [74, 85] do much better and provide an interesting alternative to fixed-size blocks. Second, using set reconciliation techniques with overlapping content-dependent partitioning is a promising approach that does well on collections where versions are quite similar. Third, sampling techniques can efficiently decide whether set reconciliation should be used and how many evaluations to send, but it is not easy to use sampling to decide on the best block size.

6.2 Future Work

For the index compression techniques with optimized document reordering, our work motivates several open questions. First, since our methods are orthogonal to the reordering techniques, it will be interesting to combine our methods with other better reordering techniques, for example, the TSP-based methods in [32], to further reduce index sizes. Second, since we showed that query processing benefits from more efficient skipping in reordered indexes, it would be interesting to see if our approach can be combined with other skipping techniques and whether the reordering could be improved by considering query load. For example, our work is related to, but different from, recent work in [19, 27] that shows how to choose an optimal set of forward pointers (basically, how to choose variable block boundaries) for each list based on an analysis of the query load. Third, it would be interesting to coordinate the document reordering techniques used to improve compression with the index ordering methods used in early termination techniques and explore trade-offs between them. Fourth, there is an interesting relationship between compression of reordered indexes and efficient indexing of archival collections. In fact, we have applied some ideas in this thesis to compression techniques for versioned document collections [42].

For the position compression techniques, there are still a number of remaining open challenges. First, it would be nice to find better ways to exploit page reordering for better position compression. More generally, it is an interesting question whether there are other organizations for position data, different from the standard inverted-list organizations, that allow efficient query processing while enabling better compression. For instance, one could even consider storing the parsed documents themselves in highly compressed form and accessing these during a position data lookup, instead of keeping the positions in inverted lists. Second, it will be very interesting to integrate our work with early termination techniques [3, 7, 36, 53, 67, 73, 93, 96]. Early termination techniques are widely used optimization techniques to improve search engine performance. The common goal of such techniques is to speed up document retrieval by avoiding fully traversing complete inverted lists that are relevant to the given query. Unfortunately, most existing early termination techniques do not consider position information, while real search engines do take positions into consideration in their ranking functions. One possible solution [73, 96] is to build auxiliary phrase indexes or pair indexes for certain phrases or pairs of terms. However, one disadvantage of such methods is that the size of auxiliary indexes may be very large and thus degrade the overall performance. Thus, it would be interesting to find better compression methods for phrase indexes or pair indexes since they have different properties than single terms. For instance, a rare phrase may be composed of two very popular terms, and thus the resulting index size of the phrase may be much smaller than that of either of its terms.

For the file synchronization techniques, there are several open questions and opportunities for future research. These include a better understanding of the use of sampling to choose the best (expected) block size for the partitioning, and how to integrate the resulting method and our reconciliation approach into the *rsync*

tool. In addition, there are interesting problems related to content-dependent partitioning schemes (see, e.g., [74, 85]), such as analyzing the trade-off between the number of blocks and the amount of redundancy that is detected.

Bibliography

- [1] S. Agarwal, V. Chauhan, and A. Trachtenberg. Bandwidth efficient string reconciliation using puzzles. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1217–1225, November 2006.
- [2] V. Anh. Impact-based document retrieval. PhD Thesis, The University of Melbourne, April 2004.
- [3] V. Anh, O. Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. of the 24th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 35–42, September 2001.
- [4] V. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. of the 15th Int. Australasian Database Conference*, pages 61–67, 2004.
- [5] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, January 2005.
- [6] V. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, 2006.
- [7] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impact scores. In *Proc. of the 29th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 372–379, 2006.
- [8] V. Anh and A. Moffat. Index compression using 64-bit words. *Software:Practice and Experience*, 40(2):131–147, January 2010.
- [9] R. Baeza-Yate¹, F. Junqueira¹, V. Plachouras¹, and H. Witschel. Admission policies for caches of search engine results. In *Proc. of the 14th String Processing and Information Retrieval Symposium (SPIRE)*, pages 74–85, 2007.
- [10] R. Baeza-Yate¹ and F. Saint-Jean. A three level search engine index based in query log distribution. In *Proc. of the 10th String Processing and Information Retrieval Symposium (SPIRE)*, pages 56–65, 2003.
- [11] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. of the 30th*

- Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 183–190, 2007.
- [12] S. Balasubramaniam and B. Pierce. What is a file synchronizer? In *Proc. of the ACM/IEEE MOBICOM Conference*, 1998.
 - [13] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. pages 320–330.
 - [14] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A time machine for text search. In *Proc. of the 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 519–526, 2007.
 - [15] N. Bjorner, A. Blass, and Y. Gurevich. Content-dependent chunking for differential compression, the local maximum approach. Technical Report MSR-TR-2007-102, Microsoft Research, Aug 2007.
 - [16] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. of the 27th European Conf. on Information Retrieval*, pages 375–387, 2005.
 - [17] R. Blanco and A. Barreiro. TSP and cluster-based solutions to the reassignment of document identifiers. *Information Retrieval*, 9(4), 2006.
 - [18] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. of the Data Compression Conference*, pages 342–351, 2002.
 - [19] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proc. of the 12th Int. Conf. on String Processing and Information Retrieval (SPIRE)*, pages 25–28, 2005.
 - [20] A. Bookstein, S. Klein, and T. Raita. Modeling word occurrences for the compression of concordances. *ACM Transactions of Information Systems*, 15(3):254–290, 1997.
 - [21] A. Bookstein, S. Klein, and T. Raita. Markov models for clusters in concordance compression. In *Proc. of the Data Compression Conference*, pages 116–125, March, 1994.
 - [22] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh World Wide Web Conference*, pages 107–117, 1998.
 - [23] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.
 - [24] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th Int. Conf. on Information and Knowledge Management*, pages 426–434, November 2003.

- [25] A. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. Shekita. Indexing shared content in information retrieval systems. In *Proc. of the 10th Int. Conf. on Extending Database Technology*, pages 313–330, 2006.
- [26] S. Buettcher, C. Clarke, and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proc. of the 29th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 621–622, 2006.
- [27] F. Chierichetti, S. Lattanzi, Federico Mari, and A. Panconesi. On placing skips optimally in expectation. In *Proc. of the Int. Conf. on Web Search and Data Mining*, pages 15–24, 2008.
- [28] G. Cormode. *Sequence Distance Embeddings*. PhD thesis, University of Warwick, January 2003.
- [29] G. Cormode, M. Paterson, S. Sahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proc. of the ACM–SIAM Symp. on Discrete Algorithms*, pages 197–206, January 2000.
- [30] L. Cox, C. Murray, and B. Noble. Pastiche: Making backup cheap and easy. In *Proc. of the 5th Symp. on Operating System Design and Implementation*, pages 285–298, December 2002.
- [31] Jeff Dean. Challenges in building large-scale information retrieval systems. In *Proc. of the Int. Conf. on Web Search and Data Mining*, 2009. Keynote talk.
- [32] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *19th International World Wide Web Conference (WWW)*, pages 305–312, April 2010.
- [33] P. Eaton, E. Ong, and J. Kubiatowicz. Improving bandwidth efficiency of peer-to-peer storage. In *Proc. of the 4th IEEE Int. Conf. on Peer-to-Peer Computing*, pages 80–89, Aug 2004.
- [34] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 12(3):194–203, March 1975.
- [35] A. Evfimievski. A probabilistic algorithm for updating files over a communication link. In *Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–305, January 1998.
- [36] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, June 2002.

- [37] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems (TOIS)*, 24(1):51–78, 2006.
- [38] A. Fraenkel and S. Klein. Novel compression of sparse bit-strings – preliminary report. *Combinatorial Algorithms on Words*, 12:169–183, 1985.
- [39] J. Gailly. zlib compression library. Available at <http://www.gzip.org/zlib/>.
- [40] S. Golomb. Run-length encoding. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.
- [41] T.H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proc. of the Workshop on Web Databases (WebDB)*, 2000.
- [42] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *Proc. of the 18th ACM Conf. on Information and Knowledge Management (CIKM)*, pages 415–424, 2009.
- [43] S. Heman. Super-scalar database compression between ram and cpu-cache. MS Thesis, Centrum voor Wiskunde en Informatica (CWI), July 2005.
- [44] M. Herscovici, R. Lempel, and S. Yogev. Efficient indexing of versioned document sequences. In *Proc. of the 29th European Conf. on Information Retrieval*, pages 67–87, 2007.
- [45] J. Hunt, K.-P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Trans. on Software Engineering and Methodology*, 7:192–214, 1998.
- [46] U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In *Proc. of the IEEE INFOCOM Conference*, pages 156–160, March 2005.
- [47] U. Irmak and T. Suel. Hierarchical substring caching for efficient content distribution to low-bandwidth clients. In *Proc. of the 14th Int. World Wide Web Conference*, pages 43–53, May 2005.
- [48] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31(2):249–260, 1987.
- [49] M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems (TOIS)*, 17(4):406–439, October 1999.
- [50] P. Kulkarni, F. Douglass, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, pages 5–5, 2004.

- [51] J. Langford. Multiround rsync. January 2001. Unpublished manuscript.
- [52] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. of the 12th Int. World-Wide Web Conference*, pages 19–28, 2003.
- [53] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. of the 29th Int. Conf. on Very Large Data Bases*, pages 129–140, August 2003.
- [54] E. Markatos. On caching search engine query results. In *5th International Web Caching and Content Delivery Workshop*, May 2000.
- [55] D. Metzler and W. Bruce Croft. A markov random field model for term dependencies. In *Proc. of the 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 472–479, 2005.
- [56] Y. Minsky and A. Trachtenberg. Practical set reconciliation. Boston University Technical Report BU ECE-2002-01, 2002.
- [57] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with almost optimal communication complexity. Technical Report TR2000-1813, Cornell University, 2000.
- [58] G. Mishne and M. Rijke. Boosting web retrieval through query operations. In *Proc. of the 27th European Conference on IR Research*, 2005.
- [59] A. Moffat and V. Anh. Binary codes for locally homogeneous sequences. *Information Processing Letters*, 99(5):75–80, September 2006.
- [60] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. *Proc. of the Data Compression Conference*, pages 82–91, 1996.
- [61] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.
- [62] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. of the 15th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 274–285, 1992.
- [63] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 174–187, October 2001.
- [64] P. Noel. An efficient algorithm for file synchronization. Master’s thesis, Polytechnic University, 2004.
- [65] A. Orłitsky. Interactive communication of balanced distributions and of correlated files. *SIAM J. of Discrete Math*, 6(4):548–564, 1993.

- [66] A. Orlitsky and K. Viswanathan. Practical algorithms for interactive communication. In *IEEE Int. Symp. on Information Theory*, 2001.
- [67] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, May 1996.
- [68] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, 2002.
- [69] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *Proc. of the 9th ACM Int. Symp. on Foundations of Software Engineering*, pages 175–185, 2001.
- [70] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proc. of the 12th Int. World Wide Web Conference*, pages 619–628, May 2003.
- [71] M. Richardson, A. Prakash, and E. Brill. Beyond pagerank: machine learning for static ranking. In *Proc. of the 15th Int. World Wide Web Conference*, pages 707–715, 2006.
- [72] K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *First Latin American Web Congress*, pages 132–143, 2003.
- [73] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, and G. Weikum. Efficient text proximity search. In *14th String Processing and Information Retrieval Symposium*, pages 287–299, 2007.
- [74] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 76–85, 2003.
- [75] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 222–229, August 2002.
- [76] T. Schwarz, R. Bowdidge, and W. Burkhard. Low cost comparison of file copies. In *Proc. of the 10th Int. Conf. on Distributed Computing Systems*, pages 196–202, 1990.
- [77] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Information Processing and Management: an International Journal*, 39(1):117–131, October 2003.
- [78] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of 29th European Conf. on IR Research*, pages 101–112, 2007.

- [79] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. of the 27th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 305–312, 2004.
- [80] F. Silvestri, R. Perego, and S. Orlando. Assigning document identifiers to enhance compressibility of web search engine indexes. In *Proc. of the 19th ACM Symp. on Applied Comp*, pages 600–605, 2004.
- [81] N. Spring and D. Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proc. of the ACM SIGCOMM Conference*, pages 87–95, 2000.
- [82] D. Starobinski, A. Trachtenberg, and S. Agarwal. Efficient PDA synchronization. *ACM Transactions on Mobile Computing*, 2(1):40–51, 2003.
- [83] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proc. of the Int. Conf. on Data Engineering*, pages 153–164, March 2004.
- [84] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *Proc. of the 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 295–302, 2007.
- [85] D. Teodosiu, N. Bjorner, Y. Gurevich, M. Manasse, and J. Porkka. Optimizing file replication over limited bandwidth networks using remote differential compression. TR2006-157-1, Microsoft, 2006.
- [86] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.
- [87] U. Manber. Finding similar files in a large file system. In *Proc. of the 1994 USENIX Conference*, pages 2–12, May 1994.
- [88] Author Unspecified. How rsync works – a practical overview. <http://samba.anu.edu.au/rsync/how-rsync-works.html>.
- [89] H. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.
- [90] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.
- [91] Y. Xie and D. O’Hallaron. Locality in search engine queries and its implications for caching. In *IEEE Infocom 2002*, pages 1238–1247, 2002.
- [92] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *The 18th Int. World Wide Web Conference (WWW)*, pages 401–410, 2009.

- [93] F. Zhang, S. Shi, H. Yan, and J. Wen. Revisiting globally sorted indexes for efficient document retrieval. In *Proc. of the Int. Conf. on Web Search and Data Mining*, pages 371–380, 2010.
- [94] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *17th International World Wide Web Conference (WWW)*, pages 387–396, April 2008.
- [95] J. Zhang and T. Suel. Efficient search in large textual collection with redundancy. In *Proc. of the 16th Int. World Wide Web Conference*, pages 411–420, 2007.
- [96] M. Zhu, S. Shi, N. Yu, and J. Wen. Can phrase indexing help to process non-phrase queries? In *Proc. of the 16th ACM Conf. on Information and Knowledge Management (CIKM)*, pages 679–688, 2008.
- [97] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.
- [98] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proc. of the Int. Conf. on Data Engineering*, pages 59–59, 2006.