Compressing Term Positions in Web Indexes

Hao Yan
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY 11201
hyan@cis.poly.edu

Shuai Ding
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY 11201
sding@cis.poly.edu

Torsten Suel
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY 11201
suel@poly.edu

ABSTRACT

Large search engines process thousands of queries per second on billions of pages, making query processing a major factor in their operating costs. This has led to a lot of research on how to improve query throughput, using techniques such as massive parallelism, caching, early termination, and inverted index compression. We focus on techniques for compressing term positions in web search engine indexes. Most previous work has focused on compressing docID and frequency data, or position information in other types of text collections. Compression of term positions in web pages is complicated by the fact that term occurrences tend to cluster within documents but not across document boundaries, making it harder to exploit clustering effects. Also, typical access patterns for position data are different from those for docID and frequency data. We perform a detailed study of a number of existing and new techniques for compressing position data in web indexes. We also study how to efficiently access position data for ranking functions that take proximity features into account.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retri eval

General Terms

Algorithms, performance

Keywords

Inverted index, search engines, index compression

1. INTRODUCTION

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 in the presence of spam and other manipulation, requiring constant tuning of the system.

Web search engines use inverted index structures to evaluate queries. The sizes of these structures are typically in the range of gigabytes to terabytes, and they are stored in highly compressed form on disk or in main memory. Compression of inverted indexes saves disk space, but more importantly also reduces disk and main memory accesses, resulting in faster query evaluation. Typically, inverted indexes include information such as the document IDs (docIDs), in-document frequencies, and in-document positions of term occurrences in the collection. There is a significant amount of work on inverted index compression; see [28] for an overview and [29] for a recent experimental evaluation of state-of-the-art techniques.

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, 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 [8, 15, 26, 16, 21].

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 different pages in the book, or different 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 paper, 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. A more detailed description of our contributions is given in Section 3, after we provide some background in Section 2.

2. BACKGROUND AND RELATED WORK

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* $I_{w_0}, \ldots, I_{w_{m-1}}$ where list I_w contains a *posting* for each document containing w. Each *posting* contains

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGIR'09, July 19–23, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-483-6/09/07 ...$10.00.

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 (positions) expressed as the number of words preceding the occurrence. The postings in each inverted list are usually sorted by docID and stored in highly compressed form on disk.

There are several possible layouts of inverted lists. One layout is to keep the inverted list I_w as a contiguous sequence of postings, each of the form $(d_i, f_i, p_{i,0}, ..., p_{i,f-1})$ [28], where $p_{i,j} = k$ if w is the k-th word in document d_i. Or we can break the index into chunks, where each chunk stores say 128 docIDs, followed by the corresponding 128 frequency values, followed by all the position information for these 128 postings (usually more than 128 values) [4, 29]. Or we may have separate lists for docIDs, frequencies, and positions, each sorted in the same order. We note that some compression schemes may naturally operate on chunks of values of the same type, and thus only apply to the latter two layouts. In this paper, we do not care about the layout for docIDs and frequencies, but assume that the positions are kept in a separate list or in separate chunks.

2.1 Inverted Index Compression

Many different inverted index compression techniques have been proposed in the literature [28]. Most techniques assume that each list of postings of the form $p_i = (d_i, f_i, p_{i,0}, ..., p_{i,f-1})$ is first preprocessed by taking the differences (d-gaps) between the docIDs of any two consecutive postings, and between the values of any two consecutive positions (p-gaps) in the same posting. 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 $p_{i,j} - p_{i,j-1} - 1$. Throughout this paper we assume that we are compressing these modified values.

One problem with this approach 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, inverted lists typically store additional shortcut pointers that allow the query processor to independently decompress blocks of some limited size. (In the case of the second layout for inverted lists above, these blocks usually correspond to the chunks in the layout.) We assume that such pointers are also available for position data, allowing us to fetch the positions belonging to a particular posting.

Thus, we have the problem of compressing sequences of integer values that tend to be small on average but may follow various distributions depending on the properties of the document collection. There are many different techniques for this, including classical approaches such as gamma and delta coding [9], LLRUN [10] and its variants [20], Golomb and Rice coding [28, 30], variable-byte coding [27, 22], and more recent techniques such as Simple9 [3] and its variants [4, 2, 1, 29], or PForDelta [31]. However, these techniques have not been previously evaluated for compression of positions data in web page collections. We now outline a few of these techniques.

Gamma coding [9] represents a value $n >= 0$ 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.

In Golomb coding [12, 28] 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 $r = n \ mod \ B$. 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 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.

Variable-byte coding [27, 22] 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. 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 [30]. However, it usually does not achieve the same reduction in index size as bit-wise methods.

Simple9 coding [3] 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. 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 is a variation of Simple9 that uses 16 instead of 9 cases and thus achieves a slightly better use of each 32-bit word [29].

PForDelta is a recent technique proposed in [13, 31] 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 [10] uses Huffman 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 [11], where the binary part is referred to as "extra bits". LLRUN was improved in [20] by using a separate Huffman table for each inverted list.

2.2 Adaptive Methods for Index Compression

As explained earlier, occurrences of terms in text tend to be clustered (or locally homogeneous [17]) rather than spread out uniformly. Several previous papers [7, 6, 18, 19, 17] 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.

Bookstein et al. [7, 6] used a compression algorithm based on a multi-state Markov model to exploit clustering of terms. The basic idea of interpolative coding [19] 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. Interpolative coding achieves good performance for clustered data but is somewhat slow [28].

Moffat and Anh proposed two binary codes [17], 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 levels by using different escalations functions. For example, the s in the next level can be computed by $f(s) = 2*s$ 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.

One common property of these methods is that they are more *adaptive* than the methods discussed further above. 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.)

Finally, we point out that several authors [5, 23, 25, 24] have proposed to improve index compression by reassigning docIDs to documents such that consecutive documents are fairly similar. This essentially induces a clustering effect in the document collection, allowing for better compression of docIDs and frequencies. We also tried this approach for positions, but it gave only very limited improvements. Also relevant is the work by Kleinberg [14] on modeling burstiness in data streams using a Hidden Markov Model, which influenced some of our ideas.

2.3 Query Processing with Position Data

We now discuss how search engines access and use the positional 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 transformations. 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 [8, 15, 26, 16, 21] 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 paper on the second use, where positions are considered by the ranking function; the first use typically only applies to a limited 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 [29]. (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 is typically performed in a second stage after traversing the lists of docIDs, only a limited amount of position data is usually retrieved, and a small compressed size may be more important than extremely fast access to positions. We evaluate the query processing performance of our techniques for this case in Section 7.

3. CONTRIBUTIONS OF THIS PAPER

We study methods for compressing position data in web search engine indexes, and describe and evaluate a number of approaches. To our best knowledge, no previous published work has focused on the case of positions in web pages, as opposed to longer linearly ordered texts. In particular:

(1) We perform a detailed experimental evaluation of many existing techniques on position data in web indexes.

(2) We propose and discuss several simple but effective compression algorithms for position data that take advantage of page-wise information, such as remaining page sizes and frequencies, and other context information, e.g., the values of previous positions. We compare our algorithms to the existing ones, showing the limits of list-oriented techniques and the potential of page-adaptive approaches. We obtain moderate but measurable improvements in compression.

(3) We propose statistics-based methods to further improve compression performance by integrating more context features. Our experiments show that these methods can further reduce the compressed size over other algorithms. We also evaluate the tradeoff between integrating more features to reduce the compressed size and the extra cost for storing the features.

(4) We discuss the use of position information in search engines, and show how to efficiently access position information during query execution. We show that ranking functions that take position information into account can be evaluated with very moderate additional CPU costs compared to more basic ranking functions.

4. POSITION GAP DISTRIBUTION

Any compression method is associated with an explicit or implicit probability model for the data to be compressed. For instance, many index compression methods assume that d-gaps conform to a monotonically decreasing distribution. In particular, Golomb coding assumes geometric distributions of d-gaps. In this section, we discuss the p-gap distribution of the TREC GOV2 data set and how it differs from the case of d-gaps and among different terms.

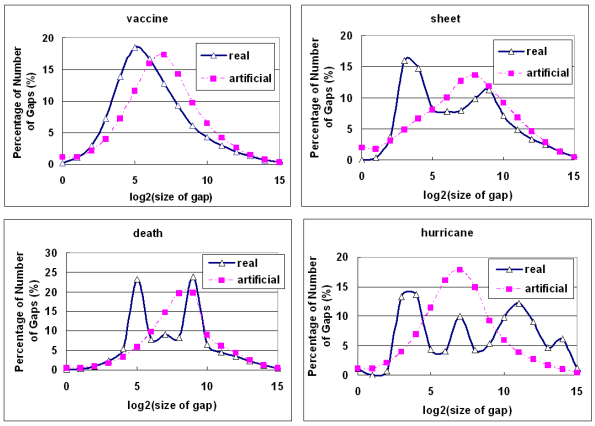

Figure 1: Distribution of p-gaps for four words on the TREC GOV2 data set. On the x-axis is the number of bits required to represent the p-gaps in binary, and on the y-axis is the percentage of p-gaps that fall in this range.

As examples, we select four terms and draw their corresponding p-gap distributions in Figure 1. We show two graphs for each term, the distribution on 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 1 we can see that the real distributions are very different from the random (i.e., geometric) distributions. (In fact, the distributions for "death" and "hurricane" are not even monotonically decreasing.) 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.

A term may occur several times in a particular document, and different occurrences may behave very differently. In Figure 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 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.

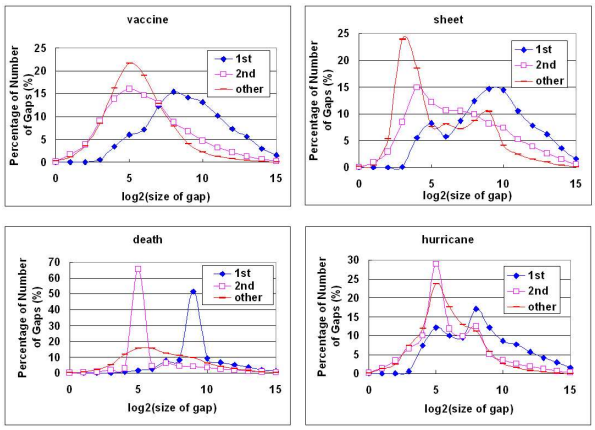

Figure 2: Distribution of p-gaps for first, second, and further occurrences, for the four words from Figure 1

5. 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.

5.1 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.

5.2 Adaptation with Smoothing

However, RPA-RC may suffer in the case in Figure 3, which shows the locations of occurrences in a document of a word.

Figure 3: An example of word locations in a document.

There are two clusters of occurrences in Figure 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:

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

where B_t is the value of B in Rice coding for tth gap, r_t is the remaining average gap for t-th gap and g_{t-1} is the value of previous gap. The second term in the model is used to tune the error of prediction 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 [17] 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 bs to predict the next b. As shown in [17], 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 [17], 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:

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

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 called this variant *Remaining Page-Adaptive BASC with Smoothing* (RPA-BASC-S).

5.3 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, 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 his 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. During decompression, for each position, we first determine which bucket the position belongs to and then retrieve the corresponding b from a global table.

Huff-4D and LLRUN-4D: Huff-4D is similar to Optb-4D except that it stores an entire Huffman table (instead of just the best value of b) for each bucket, and uses this table to encode the positions in the bucket. LLRUN-4D is similar to Huff-4D except that it builds the Huffman tables only for the unary parts of gamma codes of the positions. In other words, the difference is that each Huffman table in 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).

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.

6. EXPERIMENTAL RESULTS

We first describe our experimental setup. We used the TREC GOV2 data set of 25.2 million web pages crawled from the `gov` top-level domain. We selected 1000 random queries from the supplied query logs; these queries 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. Limited experiments involving decompression speed are provided in Section 7 in the context of a query processor that uses position data.

Throughout the paper, 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. This 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. (We believe that this is realistic given the performance characteristics of current hard disks, which strongly discourage performing multiple seeks for smaller amounts of data.)

6.1 Compression Results

In Figure 4, we compare the average compressed size of the position data per query of various methods on the TREC GOV2 data set. We show results for the following oblivious or list-adaptive methods: Gamma, variable-byte (vbyte), Simple9, Simple16 as described in [29], the version of PForDelta described in [29], list-adaptive Rice coding (list-Rice), list-adaptive riceVT as described in [28] (list-riceVT), list-LLRUN [10] (building one Huffman table for each list), RBUC and BASC [17] (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 [18, 19] (page-IPC) that is applied to the positions in each posting, a page-oriented version of riceVT (page-riceVT), PA-RC, and RPA-RC. We also show the list-wise entropy (which of course does not constitute a lower bound).

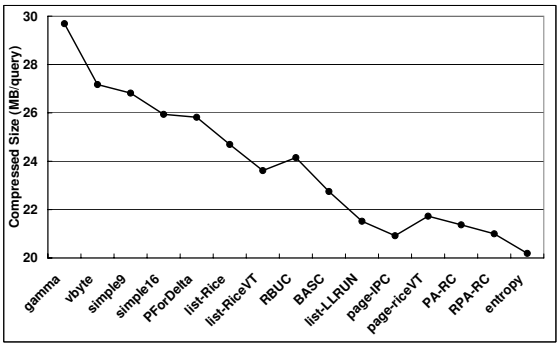

Figure 4: Compressed size per query for a variety of baseline methods on the TREC GOV2 data set.

From Figure 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 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 [18, 19]. Overall, RPA-RC is a fairly simple on-line method, and performs much better than all other methods in Figure 4 except page-IPC.

In Figure 5 we show the performance of the two regression models for different values of p (where $p = 0$ means using only the previous gaps, while $p = 1$ means using only the remaining page information). From Figure 5, 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. 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.

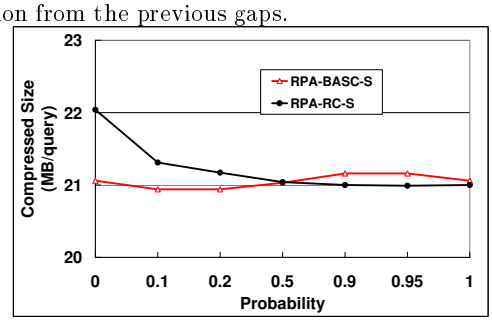

Figure 5: Compressed size per query for RPA-RC-S and RPA-BASC-S on the TREC GOV2 data set.

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.

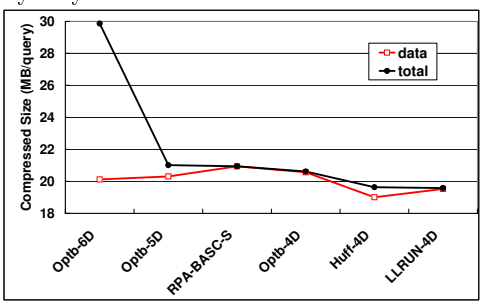

Figure 6: Compressed size per query for RPA-BASC-S, Optb-4D, Optb-5D, Optb-6D, Huff-4D and LLRUN-4D.

In Figure 6, we compare the best method from Figure 5, 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 6, 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 6, 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.

Finally, we list in Table 1 the exact compressed sizes of the methods with the best compression performance.

7. 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 do-

Table 1: 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

cID 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 are 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.

7.1 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 2: 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 3: 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 2 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.

7.2 Proximity-Aware Scoring

We now look at the use of our lookup structure in the context of ranking functions such as [8, 15, 26, 16, 21] that take term proximity, and thus positions, into account. In particular, we use the scoring model proposed by Buettcher and Clarke in [8], also used in [21], 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 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 3 and 4. In particular, Table 3 shows how likely we are to get exactly the same top-m results as an exhaustive evaluation, while Table 4 shows what

Table 4: 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%

percentage of returned results really belongs into 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 [8], justifying our claim that access patterns for position data are very different from those for docIDs and frequencies.

8. DISCUSSION AND CONCLUSIONS

In this paper, we have 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.

Overall, our improvements in compressed size are fairly moderate. 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.

There are a number of remaining open challenges concerning position data in web indexes. It would be nice to find ways to significantly improve our results, or 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.

9. REFERENCES

[1] V. Anh. Impact-based document retrieval. PhD Thesis, The University of Melbourne, Amsterdam, Netherlands, April 2004.

[2] V. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. of the 15th Int. Australasian Database Conference*, pages 61–67, 2004.

[3] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, Jan. 2005.

[4] 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.

[5] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. of the Data Compression Conference*, pages 342–351, 2002.

[6] 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.

[7] A. Bookstein, S. Klein, and T. Raita. Markov models for clusters in concordance compression. In *Proc. of the Data Compression Conference*, March, 1994.

[8] 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*, 2006.

[9] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 12(3):194–203, Mar. 1975.

[10] A. Fraenkel and S. Klein. Novel compression of sparse bit-strings – preliminary report. *Combinatorial Algorithms on Words*, 12:169–183, 1985.

[11] J. Gailly. zlib compression library. Available at `http://www.gzip.org/zlib/`.

[12] S. Golomb. Run-length encodingds. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[13] S. Heman. Super-scalar database compression between ram and cpu-cache. MS Thesis, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, July 2005.

[14] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. of the 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2002.

[15] D. Metzler and W. B. 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*, 2005.

[16] G. Mishne and M. Rijke. Boosting web retrieval through query operations. In *Proc. of the 27th European Conference on IR Research*, 2005.

[17] A. Moffat and V. Anh. Binary codes for locally homogeneous sequences. *Information Processing Letters*, 99(5):75–80, Sept. 2006.

[18] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. *Proc. of the Data Compression Conference*, pages 82–91, 1996.

[19] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.

[20] 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*, 1992.

[21] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, and G. Weikum. Efficient text proximity search. In *14th String Processing and Information Retrieval Symposium*, 2007.

[22] 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, Aug. 2002.

[23] 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, Oct. 2003.

[24] 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.

[25] 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 Appliled Comp*, pages 600–605, 2004.

[26] 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*, 2007.

[27] H. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.

[28] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.

[29] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *17th International World Wide Web Conference (WWW)*, April 2008.

[30] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.

[31] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proc. of the Int. Conf. on Data Engineering*, 2006.