Unaligned Binary Codes for Index Compression in Schema-Independent Text Retrieval Systems

Stefan Büttcher Charles L. A. Clarke
School of Computer Science
University of Waterloo, Canada
{sbuettch,claclark}@plg.uwaterloo.ca

ABSTRACT

We examine index compression techniques for schema-independent inverted files used in text retrieval systems. Schema-independent inverted files contain full positional information for all index terms and allow the structural unit of retrieval to be specified dynamically at query time, rather than statically during index construction. Schema-independent indices have different characteristics than document-oriented indices, and this difference can affect the effectiveness of index compression algorithms greatly.

Our experimental results show that unaligned binary codes that take into account the special properties of schema-independent indices achieve better compression rates than methods designed for compressing document indices and that they can reduce the size of the index by around 15% compared to byte-aligned index compression. Moreover, we present a number of performance-enhancing techniques that may be used to very efficiently decode unaligned codes. Thus, their more compact index representation does not carry the cost of a substantially decreased query processing performance. This contradicts most earlier results on the decoding performance of unaligned codes.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods; E.4 [**Coding and Information Theory**]: Data compaction and compression

General Terms

Experimentation, Performance

Keywords

Schema-Independent, Index Compression

1. INTRODUCTION

Text retrieval systems are usually based on inverted files. For every term that appears in a given text collection, an inverted file for that collection contains a list of all occurrences of the term in the collection. These *posting lists* can be thought of as sequences of integers, representing the positions of all such occurrences in the text. At query time, the inverted lists of all query terms are loaded from the index to produce the results to a given search query.

University of Waterloo technical report CS-2006-40.

For a large text collection, the size of the corresponding inverted file can be quite substantial, and it seems appropriate to apply index compression methods to decrease the size of the index. As a by-product, this size reduction can also lead to a faster search engine, as it decreases the amount of data that has to be read from disk during query processing.

Various index compression algorithms exist that can be used to cut down the size of the index and the search engine's disk activity during query processing. However, some of these methods involve rather complicated computations during decompression. Care has to be taken that the computational overhead associated with decompressing posting lists does not outweigh the savings achieved by reducing the size of the index. This is especially important for codes that are not byte- or word-aligned, as they can be very expensive to decode, due to bit-by-bit decoding operations.

This paper makes two main contributions. First, we evaluate index compression methods for *schema-independent* inverted indices [5] — purely positional indices, recording the position of each term from the beginning of the collection. In schema-independent retrieval, no pre-determined definition of "document" is required during indexing. Instead, the unit of retrieval is defined at query time, on a query-by-query basis. We have found schema-independent retrieval to be an essential part of extensible filesystem search. For example, email might be retrieved by restricting the search to mail folders and treating the individual messages as the units of retrieval. Similarly, filesystem security restrictions can be efficiently implemented by queries over the same schema-independent index, with files as the units of retrieval [3]. A schema-independent index also supports general retrieval from XML [4] where potential units of retrieval (paragraphs, sections, subsections) may overlap. We show that a schema-independent representation imposes a bimodal distribution on the gaps in the posting lists of most terms in the index, which is different from the d-gap distribution in document-level indices. We propose a novel compression technique that takes this bimodal distribution into account.

As our second contribution, we present performance-enhancing techniques for unaligned binary codes and compare their overall effect on the search engine's query processing performance. Our experimental results show that unaligned codes can provide a performance level that is equivalent to that of byte-aligned compression methods. This finding contradicts previous research, which suggests that aligned codes are preferable if high decoding performance is important. The algorithmic techniques we used to achieve this result have application to both schema indepen-

dent indices and traditional document-oriented frequency indices.

2. INVERTED FILES

An inverted file realizes a mapping from terms to their posting lists (or *inverted lists*). A posting list is a list of all occurrences of a given term within a text collection. Within the limits of this general definition, two extremes exist:

- A **frequency index** is an inverted file that does not contain exact word positions. Postings in a frequency index are of the form (*docID*, *freq*), where *docID* is an integer describing a document in the text collection, and *freq* is the frequency (number of occurrences) of a given term within that document. If postings are stored in compressed forms, it is advisable to treat the *docID* part and the *freq* part independently [15].
- A **schema-independent positional index** is an inverted file containing exact positional information (word positions) for all terms in the index. Each posting in such an index is a simple integer describing the distance of the term occurrence *from the beginning of the text collection*.

Other index representations, combinations of positional and frequency indices, that store positional information by adding a list of occurrences to each (*docID*,*freq*) pair in a frequency index, are described in the literature [11] [15].

In our framework, each posting list stored in an inverted file is divided into list segments, containing approximately 2^{15} postings each. If compression techniques are applied to the index, every list segment is compressed separately, making it easier to skip segments and also to react to local changes in the distribution of a given term. In addition, an inverted file contains some auxiliary data structures that can be used to efficiently locate the posting list for a given term within the file and to associate schema-independent postings with their respective retrieval units during query processing. We do not further discuss these data structures.

3. COMPRESSING INVERTED FILES

There is an abundance of different index compression methods in the literature. Here, we limit ourselves to a few key algorithms that provide insight into the general problem.

Most index compression methods only deal with the compression of posting lists, while leaving all other data in the index, such as the index terms themselves, untouched. This is because the vast majority ($> 90\%$) of all data in an inverted file is postings data. Existing methods for compressing posting lists usually work by transforming a given posting list to a list of d-gaps – differences between consecutive postings. For example, in a schema-independent representation the list of all occurrences of the word "the" in the TREC GOV2 text collection is:

$$96, 112, 122, 410, 423, 426, 440, 447, 571, 1077, \ldots$$

The corresponding list of d-gaps is:

$$96, 16, 10, 288, 13, 3, 14, 7, 124, 506, \ldots$$

Since the postings in a given list are sorted in increasing order, all d-gaps are positive, and the problem of compressing an inverted file can be thought of as a special case of the general problem of encoding sequences of positive integers.

Unary Encoding

If the probability distribution of the d-gaps in a posting list is of the general form

$$p(d = k + 1) \leq \frac{p(d = k)}{2} \quad \forall\, k \geq 1, \tag{1}$$

then it is very space-efficient to encode each gap as a unary number. A gap $d = k$ would then be encoded as $k - 1$ "1" bits, followed by a single "0" bit. Unfortunately, this is almost never the case for text in a natural language.

Elias' γ Code

In γ coding [7], the encoded representation of a positive integer n consists of two parts:

- $\lambda = \lfloor \log_2(n) + 1 \rfloor$, the number of bits required to encode n as a binary number, in unary representation;
- n itself, encoded as a λ-bit binary number.

We will refer to λ as the *selector* and to n as the *body* of the encoded d-gap. The γ-encoded representation of the d-gap sequence shown above is:

$$1111110\ 1100000\ 11110\ 10000\ 1110\ 1010\ \ldots,$$

where "1111110" is the unary representation of

$$\lfloor \log_2(96) + 1 \rfloor = 7,$$

and the following 7 bits encode 96 as a 7-bit binary number.

This version of γ coding can be improved by using the implicit information stored in the selector: If $\lambda = k$, then this implies that n cannot be stored using $k - 1$ bits. Thus, the most significant bit in the k-bit representation of n will always be 1 and therefore does not need to be stored explicitly, saving us 1 bit per posting:

$$1111110\ 100000\ 11110\ 0000\ 1110\ 010\ \ldots$$

γ coding is based on the probability assumption that

$$p(|d| = k + 1) \approx \frac{p(|d| = k)}{2}, \tag{2}$$

where $|d| = \lfloor \log_2(d) + 1 \rfloor$, d's length as a binary number.

Huffman Codes

By storing the selector value in unary, γ coding assumes that most d-gaps can be represented in a relatively small number of bits. Obviously, this is only true for high-frequency terms. For less frequent terms, a substantial amount of space is wasted by choosing unary representation for λ.

A solution to this problem is to not assume anything about the d-gap distribution, but use a Huffman code instead of unary to encode the selectors. Since the set of possible λ values – the symbol set to be encoded by Huffman – is very small (< 64), compared to the total size of the message (posting lists grow linearly with the size of the collection), the overhead introduced by prepending the Huffman tree to the encoded postings is negligible, and Huffman coding is a perfect candidate for compressing posting lists. Since Huffman codes, like the unary code employed by the γ method, are prefix-free, the resulting bit sequence can still be decoded unambiguously.

In contrast to the parameterless γ code, Huffman codes are parameterized and thus require the compressor to traverse the inverted list twice – once to gather statistical data, and a second time to perform the actual compression. This

makes the encoding process slightly less efficient, but has no effect on the decoding performance.

Golomb-Rice Codes

Consider a document-level index, without any positional or frequency information, for a document collection in which all documents are independent of each other. If the collection consists of N documents and a term T appears in n_T of them, then this implies the following d-gap distribution:

$$p(d = k+1) = \left(1 - \frac{n_T}{N}\right) \cdot p(d = k) \quad (3)$$

$$= \left(1 - \frac{n_T}{N}\right)^{k-1} \cdot \frac{n_T}{N}. \quad (4)$$

In an exhilarating essay about a secret agent in a casino, Golomb [8] proposes to encode integers sequences $a_{i \geq 1}$ that follow this distribution by choosing a parameter b, encoding $\lfloor \frac{a_i}{b} \rfloor$ as a unary number and the remainder of the division operation (a_i mod b) as a $|b|$-bit binary number. The parameter b realizes a transformation to a different probability distribution – one that is compatible with the assumption behind unary encoding – and is usually chosen as

$$b \approx \frac{n_T}{N} - 1. \quad (5)$$

When b is a power of 2, Golomb codes are also referred to as Golomb-Rice codes, or simply Rice codes.

Interpolative Coding

Moffat and Stuiver [11] [12] present a compression method that is not based on d-gaps, but instead uses a recursive descent to encode a given list of postings. Like d-gap-based techniques, it exploits the fact the postings in an inverted list are sorted in increasing order. For a sequence of n postings $(a_1, a_2, \ldots, a_n)$, interpolative coding assumes that a_1 and a_n have already been encoded (possibly using some other method, such as γ coding) and then encodes the integer $a_{\lfloor n/2 \rfloor} - a_1 - 1$ as a k-bit binary number, where k is $\lceil \log_2(a_n - a_1 - 1) \rceil$. That is, it uses the fact that

$$a_1 < a_{\lfloor n/2 \rfloor} < a_n$$

(postings are sorted in increasing order) to find an upper bound for the number of bits needed to represent the integer $a_{\lfloor n/2 \rfloor} - a_1 - 1$ (and thus the posting $a_{\lfloor n/2 \rfloor}$) as a binary number with that many bits. It then continues recursively for the subsequences $(a_1, \ldots, a_{\lfloor n/2 \rfloor})$ and $(a_{\lfloor n/2 \rfloor}, \ldots, a_n)$. Moffat and Stuiver report that interpolative coding in practice achieves excellent compression rates for document-level indices, especially when the documents are not independent, but term occurrences are clustered.

Byte-Aligned Coding

The coding techniques described above all have in common that they produce unaligned codes. Code words may consume an arbitrary number of bits and cross byte boundaries, presumably necessitating a bit-by-bit encoding and decoding process. On most computer architectures, bit operations are costly, and it is far more efficient to operate on entire bytes (or even words) instead of individual bits.

Williams, Scholer et al. [17] [15] describe a method that encodes/decodes sequences of d-gaps by only accessing entire bytes in memory. In each byte of a code word, 7 bits are used to store actual d-gap data, while the 8th bit, as a continuation flag, is used to indicate whether there are more data to follow for the current code word. We refer to this byte-aligned method as vByte. A C++ implementation of the encoding routine is given by Figure 1.

```
int compressVByte(long postings[], int cnt, byte out[]) {
  int result = 0;
  long previous = -1;
  for (int i = 0; i < cnt; i++) {
    long delta = postings[i] - previous - 1;
    while (delta >= 128) {
      out[result++] = 128 + (delta & 127);
      delta = delta >> 7;
    }
    out[result++] = delta;
    previous = postings[i];
  }
  return result;
}
```

Figure 1: A C++ implementation of the vByte compression algorithm. The procedure reads postings from the `postings` array and writes their vByte-encoded representations to `out`, returning the number of bytes written.

The compression rates achieved by byte-aligned coding usually are not competitive with those achieved by unaligned methods, such as interpolative coding or Huffman. On the other hand, decompression is much faster, due to the absence of bit-wise decoding operations. This is why vByte (or one of its variations) is the default compression method in many search engines (e.g., Lucene, Wumpus, Zettair).

Word-Aligned Coding

Recently, Anh and Moffat [1] [2] have proposed a family of coding schemes that combine postings into groups and encode each such group in a 32-bit machine word, using the same number of bits for every posting in the group, but allowing to change the number of bits per posting between groups. For document-level indices, their technique achieves compression rates better than vByte, while offering similar decoding performance — due to the simplicity of the decoding routine and the fact that all memory accesses can be carried out in a word-aligned fashion.

4. SCHEMA-INDEPENDENT INDICES AND BIMODAL DISTRIBUTIONS

Existing work on compression techniques for inverted files usually focuses on document-level frequency indices without any positional information [14] [11]. Compression methods have also been studied for other index representations, such as impact-ordered indices [2], which, like ordinary frequency indices, are document-oriented and position-unaware. Even when compression techniques for positional indices are discussed, the underlying index representation is usually still document-oriented; document identifiers and within-document positional information are encoded separately, potentially using different techniques for each component [13] [16]. All these approaches have in common that they are based on documents, or some sort of retrieval unit that has to be known during index construction. Thus, if a user wants to build an index for a text collection, before starting the process she has to decide what the retrieval unit is that she will be targeting with her queries. These retrieval units are then called "documents", but in fact may be email messages, web pages, or book chapters. This approach becomes more problematic in the context of XML

Term	Coll. Freq.	Doc. Freq.	Mean TF
landfilling	125	79	1.58
rattlesnake	250	181	1.38
corsica	500	299	1.67
ymca	1,000	571	1.75
semester	2,000	1,316	1.52
divorced	4,000	3,178	1.26
absent	8,003	6,552	1.22
certified	16,004	8,526	1.88
streets	32,000	23,019	1.39
church	64,290	24,808	2.59
agreed	128,159	96,053	1.33
director	256,585	156,358	1.64
high	510,958	273,802	1.87
there	1,024,255	435,736	2.35
which	2,026,126	731,173	2.77

Table 1: Term statistics for some terms more or less randomly selected from TREC disks 1-5. For many terms, a single occurrence of the term within a document is a good indicator for another occurrence within the same document (the mean within-document term frequency is greater than 1).

retrieval, where retrieval units may overlap [4], and it breaks down completely in the context of file system search, where retrieval units may be defined dynamically depending on the application [3].

Schema-Independent Indexing

The schema-independent approach to text indexing, as described by Clarke et al. [6], allows a more flexible data representation. During index construction, the search engine processes a sequence of

$$\text{(token, position)}$$

pairs, as returned by the input tokenizer, and builds an index without making any assumptions about the structure of the text being indexed. At query time, arbitrary retrieval units may be chosen on a per-query basis, such as

$$\text{“}\langle\text{DOC}\rangle\text{”} \cdots \text{“}\langle/\text{DOC}\rangle\text{”}$$

to search for TREC-style documents,

$$\text{“}\langle\text{sec}\rangle\text{”} \cdots \text{“}\langle/\text{sec}\rangle\text{”}$$

to search for sections (instead of entire articles) in the INEX XML collection, or

$$\begin{aligned} &(\text{“}\langle\text{MedlineCitation}\rangle\text{”} \cdots \text{“}\langle/\text{MedlineCitation}\rangle\text{”}) \\ \rhd\ &((\text{“}\langle\text{PubDate}\rangle\text{”} \cdots \text{“}\langle/\text{PubDate}\rangle\text{”}) \\ \rhd\ &\quad \text{“}\langle\text{Year}\rangle 1995 \langle/\text{Year}\rangle\text{”}) \end{aligned}$$

to restrict the search to PubMed Medline records that were published in 1995. A description of available operators to impose structural constraints on the retrieval unit that is targeted by a search operation is given by Clarke et al. [5].

Schema-independent indexing gives the user greater flexibility at query time, but makes it impossible to exploit the inherent structure of the text collection at indexing time and for index compression purposes. This results in posting lists that have a different distribution than the ones found in a document-based index, which greatly affects the effectiveness of index compression techniques. The only discussion of encoding integers under such an index representation we are aware of is given by Williams and Zobel [17] who mainly discuss the savings over a schema-unaware approach that can be achieved when additional information about the structure of the collection is available. We extend their findings and provide a more detailed discussion of the effect of a schema-independent index representation and its differences from the document-based approach.

d-Gap Distribution in Document-Level Indices

As mentioned in the previous section, when discussing Golomb codes, the assumption that all documents are independent of each other leads to the d-gap distribution

$$p(d = k) = \left(1 - \frac{n_T}{N}\right)^{k-1} \cdot \frac{n_T}{N} \tag{6}$$

in a document-level posting list (only containing document IDs) for a term T occurring in n_T documents. For the number of bits needed to encode a gap d as a binary number, denoted by $|d|$, we obtain:

$$p(|d| = k) = \left(1 - \frac{n_T}{N}\right)^{2^{k-1}-1} - \left(1 - \frac{n_T}{N}\right)^{2^k-1}. \tag{7}$$

This distribution has its analytical maximum at

$$k_{\max} = \log_2\left(\frac{-2 \cdot \log(2)}{\log(1 - n_T/N)}\right) \tag{8}$$

and then rapidly drops towards zero, as shown in Figure 2 for four terms arbitrarily selected from the text collections known as TREC disks 1-5. For the term “rattlesnake” in the TREC collection, for instance, with

$$\frac{n_T}{N} = \frac{181}{1544847} = 1.172 \cdot 10^{-4},$$

we can predict a maximum around

$$k_{\max} = \log_2\left(\frac{-2 \cdot \log(2)}{\log(0.9998828)}\right) \approx 13.5, \tag{9}$$

which is consistent with the peak at 13 bits in Figure 2(a). Minor deviations from the theoretical distribution are because documents in the chosen collection are not independent of each other, but are sorted according to publisher and date of publication, resulting in some local inter-dependencies.

The effectiveness of Golomb codes relies on the rapid drop after the peak, because everything on the right-hand side of the peak involves unary codewords. If the independence assumption is wrong, Golomb codes will not yield good compression rates.

d-Gap Distribution in Schema-Independent Indices

Although, in schema-independent indexing, nothing is assumed about the structure of the text collection, there are natural breaks in the collection, where the text on one side is independent of the text on the other. In traditional text collections, these breaks occur at document boundaries. Generalizing, these boundaries also occur between email messages, journal articles, and files.

Figure 3 shows the d-gap distribution for the same four terms already analyzed in Figure 2, this time under a schema-independent index representation. It can be seen that at least three of the four terms clearly have two local maxima in their d-gap distribution. This is because, whenever a term appears in a document, chances are it will appear a second time (or more), as indicated in Table 1 by

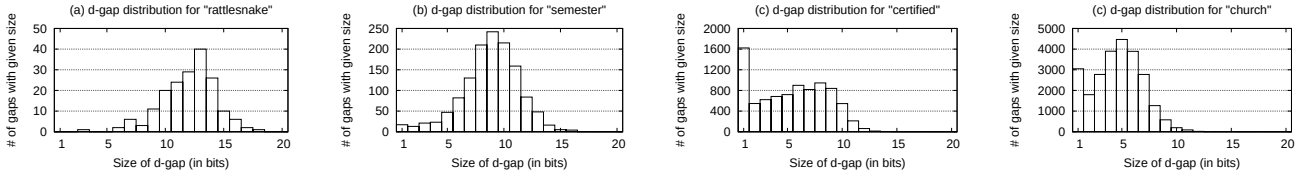

Figure 2: Gap distribution of the posting lists for "rattlesnake", "semester", "certified", and "church" in a document-level index with each posting corresponding to a document ID, extracted from TREC disks 1-5.

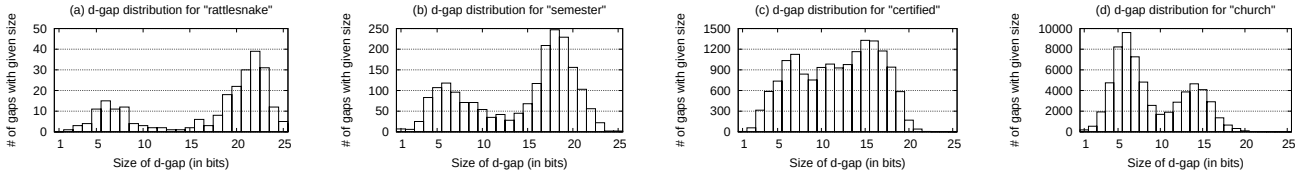

Figure 3: Gap distribution of the posting lists for "rattlesnake", "semester", "certified", and "church" under a schema-independent positional index representation, extracted from TREC disks 1-5.

the column labelled "Mean TF". Thus, the d-gap distribution is the sum of two distributions, an inter-document and an intra-document distribution. As the average document in the TREC collection contains 533 tokens, we can expect to see the first maximum in the lower range, below 9 bits, corresponding to multiple terms appearing in the same document, and the second maximum around $9 + k$ bits, where k is the position of the maximum in the corresponding distribution in a document-level index, as shown by Figure 2. For the term "semester", the maximum in a document-level index is at 9 bits; in the schema-independent index for the same text collection, it is at 18 bits.

This bimodal distribution, the effect of combining the intra-document with the inter-document distribution, means bad news for γ coding, as most d-gaps, due to the second maximum in the probability distribution, are no longer small, but require a larger number of bits, and also for Golomb coding, because it is no longer possible to find a suitable split between binary and unary encoding.

5. INDEX COMPRESSION FOR SCHEMA-INDEPENDENT TEXT RETRIEVAL

We now present an unaligned index compression scheme that explicitly takes into account the bimodal nature of the posting lists in a schema-independent index. We start by introducing *generalized unaligned binary coding* (GUBC), which is not a novel technique, but a rather obvious generalization of γ coding. We then explain *generalized unaligned binary coding with n components* (GUBC-n), which is a novel technique that cannot be found in the existing literature.

Generalized Unaligned Binary Coding

GUBC is a generalization of Elias' γ method. In a γ code, the size of the selector for every code word is equal to the size of the body; every bit in the body corresponds to one bit in the selector (possibly without the most significant bit of the body, which can be omitted, as discussed in section 3). GBUC generalizes this approach by associating each bit in the selector with σ bits in the body. If $\sigma = 5$, for instance, a 3-bit selector is followed by a 15-bit body, leading to a total code word size of 18 bits (as opposed to 29 or 30 bits if γ were used).

When a posting list needs to be compressed, the GUBC algorithm performs a brute-force search for the optimal value of σ, examining all values $1 \leq \sigma \leq 15$. The optimal value is selected and prepended to the compressed posting list (encoded as a 4-bit integer) to inform the decompressor of the σ value that was chosen. The brute-force search during the compression process seems expensive, but can in fact be realized by using histogram information gathered in a single traversal of the posting list, just like in the case of Huffman codes. This way, the search process is largely independent of the size of the list, and its cost is negligible, at least for long lists ($> 10{,}000$ postings).

For $\sigma > 1$, the ability of the γ method to implicitly encode the most-significant bit of the body inside the selector is lost. However, it is not completely lost, as the information that a d-gap is encoded using $k \cdot \sigma$ bits, for some integer k, still implies that it cannot be encoded in $(k-1) \cdot \sigma$ bits. This information can be used in a recursive fashion to save at least fractions of a bit, if not an entire bit, per posting.

Generalized Unaligned Binary Coding with n Components

GUBC can be generalized to the GUBC-n method. In GUBC-n, the compressor may choose a parameter tuple $(\sigma_1, \ldots, \sigma_n)$ instead of a single parameter σ. Under this encoding scheme, a code word selector of length k indicates a body containing s bits:

$$s = \begin{cases} \sum_{i=1}^{k} \sigma_i & \text{if } k \leq n \\ \sum_{i=1}^{n} \sigma_i + \sum_{i=n+1}^{k} \sigma_n & \text{if } k > n \end{cases}$$

The first bit in the selector corresponds to σ_1 bits in the body, the second bit in the selector to σ_2 bits in the body, and so forth. Of particular interest to us is the GUBC-3 method. This is motivated by the fact that d-gaps in a typical posting list follow a bimodal distribution, as shown in the previous section. Allowing the compressor to use 3 different chunk sizes accommodates for this: The first chunk can cover the local maximum that stems from the within-document distribution, the second chunk can cover the local maximum that stems from the inter-document distribution, and all further chunks can take care of the rest. Like in the case of GUBC, the optimal parameter settings

are determined by performing a brute-force search on a histogram data structure created from the posting list to be compressed, necessitating an additional pass over the list.

If we apply GUBC-3 to the example terms whose d-gaps distributions are shown in Figure 3, the brute-force search finds the following optimal parameter configurations:

- $(8, 12, 1)$ for "rattlesnake";
- $(9, 8, 1)$ for "semester";
- $(8, 5, 1)$ for "certified";
- $(7, 5, 1)$ for "church".

This corresponds to the positions of the local maxima of the d-gap distributions shown in the figure.

Our experimental results show that, despite its simplicity, GUBC-3 achieves very good compression rates in this type of index, better than most other integer encoding techniques.

6. INCREASING THE PERFORMANCE OF UNALIGNED CODES

Although unaligned compression techniques, like Huffman, interpolative coding, and the GUBC schemes presented in the previous section, that do not conform to byte boundaries in most cases offer better compression effectiveness than byte-aligned or word-aligned methods, both for document-level and for schema-independent indices, they are usually not used in search engines, mainly because their decompression performance is too low.

For example, Trotman [16] and Anh and Moffat [2] unanimously report that the decoding routine of their implementation of interpolative coding is up to 10 times slower than that of the vByte method. In fact, the decoding overhead reported by Anh and Moffat – 100-400 ns per posting – is so immense that it seems more efficient to use an uncompressed index than an index compressed using the interpolative method. Reading an uncompressed 32-bit posting from disk only takes around 100 ns on average, assuming a read throughput of 40 MB per second.

As a rough guideline, index compression is always worthwhile if the disk I/O time saved by compressing a posting is greater than the time it takes to decode it during query processing. Suppose we can choose between two index compression methods A and B, and A can encode postings using 1 bit less than B on average. Assume further that the inverted file is stored on a hard drive that achieves an average read throughput of 40 MB/s. Then this means that the disk I/O time saved by encoding postings with A instead of B is 3.1 ns per posting. Hence, choosing A over B is only sensible if A's decoding routine is at most 3 ns per posting slower than B's. On a typical CPU, this is between 5 and 10 clock cycles — not a lot of time. A more thorough discussion of this trade-off between disk I/O and decoding performance is given by Trotman [16].

Since even the best compression techniques usually only save a few bits per posting, compared to vByte or word-aligned techniques, it is important that their decoding routines are highly optimized, as even a few extra clock cycles per posting can outweigh the savings created by better compression effectiveness and thus result in a lower overall query processing performance.

In this section, we show how to efficiently decode unaligned binary codes. The main objective is to eliminate all bit-by-bit data access patterns and replace them by operations that work on a larger number of bits, preferably entire bytes or even whole machine words. The two central techniques used to achieve this are *bit buffering* and *selector look-ahead*.

```
int compress7Bits(long postings[], int cnt, byte out[]) {
  int result = 0;
  long previous = -1;
  long bitBuffer = 0;
  int bitsInBuffer = 0;
  for (int i = 0; i < cnt; i++) {
    long delta = postings[i] - previous - 1;
    while (delta >= 64) {
      bitBuffer += (64 + (delta & 63)) << bitsInBuffer;
      bitsInBuffer += 7;
      delta = delta >> 6;
    }
    bitBuffer += delta << bitsInBuffer;
    bitsInBuffer += 7;
    while (bitsInBuffer >= 8) {
      out[result++] = (bitBuffer & 255);
      bitBuffer >>= 8;
      bitsInBuffer -= 8;
    }
    previous = postings[i];
  }
  if (bitsInBuffer > 0)
    out[result++] = (bitBuffer & 255);
  return result;
}
```

Figure 4: A C++ implementation of a compression algorithm similar to vByte (Figure 1). d-gaps are encoded as sequences of 7-bit integers instead of 8-bit integers. The difference to vByte is the introduction of the `bitBuffer` that allows the program to perform byte-aligned memory accesses.

Bit Buffering

When decoding unaligned binary codes from an input byte array to some output buffer, most code words will span across multiple elements of the input array. To decode a code word, all these array elements will need to be accessed individually, their contents need to be combined, and bit shift operations have to be performed in order to combine the data stored in the array into a single variable representing the current code word. This can be very time-consuming, as mentioned above. The solution to this problem is to use a single variable (probably a register if compiler optimizations are turned on) the size of a machine word that is used to buffer memory accesses and allows to access the input array in a more efficient fashion.

In the decoding procedure of an unaligned coding method, data are transferred from the input array into the bit buffer, 8 bits at a time, and the actual decoding operations are exclusively performed on the lower-most bits of the buffer, representing the current code word in the encoded stream of d-gaps. Bits are added to and removed from the bit buffer by using shift operations, which can be carried out very efficiently. In the encoding procedure bits emitted by the encode are transferred into the bit buffer and only copied into the output array in chunks of 8 bits each, allowing for efficient byte-aligned memory access.

Figure 4 shows how bit buffering can be employed to improve the performance of a compression technique that is similar to vByte, but encodes d-gaps in chunks of 7 bits instead of 8. Without bit buffering, this would lead to a

```
int selector = 1;
do {
  int posOfFirstZeroBit = firstZeroBit[bitBuffer & 255];
  selector += posOfFirstZeroBit;
  bitsInBuffer -= posOfFirstZeroBit;
  bitBuffer >>= posOfFirstZeroBit;
} while (posOfFirstZeroBit >= 8);
```

Figure 5: Efficient selector look-ahead in the decoding routines of Elias γ and GUBC-n, using a precomputed table `firstZeroBit` containing values in the range 0..8.

complex unaligned memory access pattern where data from adjacent bytes in memory need to be combined via a costly sequence of bit shift and bit mask operations. The bit buffer solves this problem. Depending on the characteristics of the posting list to be processed, the 7-bit method is only between 40% and 90% slower than 8-bit method vByte, not 10 times slower, as indicated by Anh and Moffat [2].

The bit buffering technique can be used to speedup most unaligned, variable-bit encoding schemes. Additional performance improvements can be achieved by not transferring individual bytes, but 16-bit or even 32-bit chunks between the bit buffer and main memory. However, bit buffering requires that all d-gaps can be encoded in $w-7$ bit (or $w-15$ bits when transferring 16-bit chunks between bit buffer and main memory), where w is the size of a machine word on the chosen architecture. For 32-bit CPUs, this will lead to problems, as soon as the text collection for which an inverted file is being created consists of more than 2^{25} (33.6 million) tokens. Fortunately, with modern 64-bit CPUs, this is not an issue. We don't expect to see text collections containing more than 144 petabytes (2^{57} bytes) in the near future.

Selector Look-Ahead

Elias' γ, the Huffman method, and the GUBC-n codes presented in the previous section all have in common that their encoded representation of a d-gap consists of two parts: a selector, indicating the size of the code word, followed by the body, containing the d-gap as a binary number. This poses the problem of how to determine which part of the code word is the selector and which part is the body. For γ and GUBC-n, the selector is a sequence of "1" bits, followed by a "0" bit indicating the end of the selector. A straightforward implementation could scan the lower part of the bit buffer, one bit at a time, in order to find the first "0" bit in the buffer. However, this would re-introduce the bit-by-bit decoding operations that we were trying to eliminate.

The solution to this problem is to use a selector look-ahead table that can be used to determine the size of the selector by performing a small number of array lookups instead of a large number of bit operations. For γ and GUBC-n, the end of the selector is signalled by a "0" bit. We can use a precomputed table with 256 elements to quickly find out the position of the first "0" bit in any given sequence of 8 bits. This process is shown in Figure 5.

If the selectors are guaranteed to be at most 8 bits long, then this can be done even more efficiently. Moreover, the same general technique can be used for any prefix code, including Huffman codes, not only for "0" terminated selectors as in the case of GUBC-n.

Speeding up GUBC-n

The decoding process for GUBC-n (including the special case Elias γ) has two potential bottlenecks: Reading unaligned code words from memory and determining the length of the current code word by finding out the value of the zero-terminated selector component. Both bottlenecks can be eliminated by applying the techniques discussed above. Thus, an efficient implementation of a GUBC-n encoder is straightforward.

Speeding up Huffman

The decoding process for Huffman codes has two potential bottlenecks: Reading unaligned code words from memory and determining the value of the selector. Unaligned memory accesses can be eliminated by integrating a bit buffer into the decoding procedure. Quickly determining the value of the current selector is a bit more complicated than in the case GUBC-n. However, since Huffman codes are prefix-free, it is possible to construct a lookup table containing 2^k elements which can then be used to determine, for every sequence of k bits, the Huffman code word that the sequence starts with.

The lookup table needs to be constructed during the decoding process, since it depends on the Huffman tree, which slightly reduces decoding performance. Moreover, limiting the lookup table to 2^k means limiting the length of all Huffman code words to k bits, which potentially decreases the compression effectiveness of this method. However, for this specific application, the degradation is negligible. Milidiú et al. [9] provide an excellent discussion of the properties of length-restricted Huffman codes.

In our implementation of Huffman coding, we employ the BRCI [9] algorithm to impose a length limit of 10 bits on all selectors and consequently use a lookup table containing 2^{10} elements to quickly determine the value of the next selector, given the current content of the bit buffer.

Speeding up Interpolative Coding

The decoding procedure for interpolative coding has three potential bottlenecks: Reading unaligned code words from memory, recursive function calls, and determining $\lceil \log_2(a_k - a_1 - 1) \rceil$, the number of bits used to encode the posting $a_{\lfloor k/2 \rfloor}$. Unaligned memory accesses can be eliminated using a bit buffer. Recursive function calls, as suggested by the recursive definition of the method, can be replaced by an explicitly-maintained stack. The third problem, determining the number of bits used to encode a posting, can be realized efficiently by performing a local search, starting from the number of bits used to encode the posting a_k, $a_{\lfloor k/2 \rfloor}$'s parent in the recursive call stack. In most cases, the number of bits used for $a_{\lfloor k/2 \rfloor}$ is one or two less than the number of bits used for a_k, so a local search will be very efficient.

7. EXPERIMENTAL EVALUATION

The experimental evaluation of the techniques we have presented consists of three parts: Compression effectiveness, decoding performance, and impact on overall search engine response time. All methods are evaluated against vByte, the de-facto standard for index compression in search engines.

Experimental Setup

All performance experiments were conducted on a PC based on a 64-bit AMD Athlon64 3500+ CPU (2.2 GHz) with 2 GB of RAM, running a 64-bit Linux operating system. The inverted files were stored on a 7,200-rpm SATA hard drive.

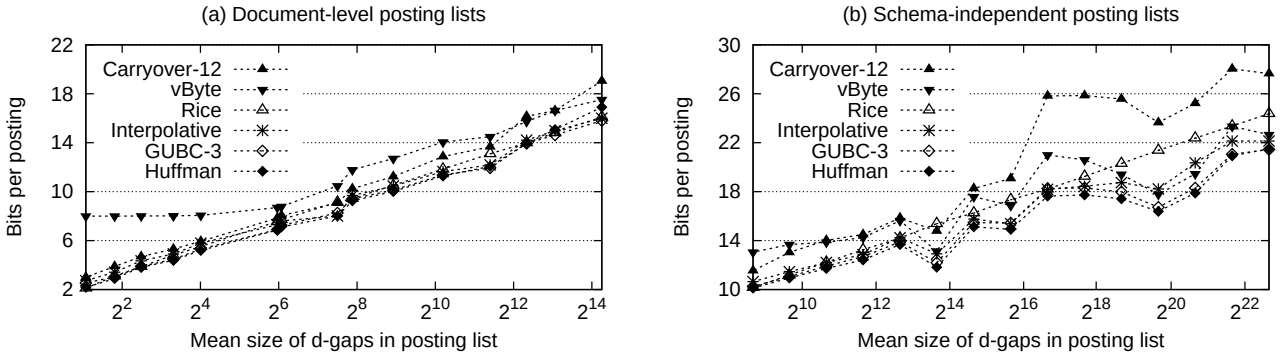

Figure 6: Compression effectiveness for document-level and schema-independent posting lists for the terms shown in Table 1. For document-level postings, all methods (except for vByte and Carryover-12) produce similar results. For schema-independent postings, this is not the case.

	vByte	Gamma	Rice	Interpol	Huffman	GUBC-3	RBUC-B	Carryover-12
landfilling	22.7 bits	32.1 (+9.4)	24.4 (+1.7)	**22.1 (-0.5)**	**21.5 (-1.2)**	**21.4 (-1.2)**	23.6 (+0.9)	27.7 (+5.0)
rattlesnake	23.3 bits	32.6 (+9.2)	23.4 (+0.1)	**22.1 (-1.2)**	**20.9 (-2.4)**	**21.1 (-2.3)**	**23.0 (-0.3)**	28.0 (+4.7)
corsica	19.5 bits	27.0 (+7.6)	22.4 (+2.9)	20.4 (+0.9)	**17.9 (-1.6)**	**18.3 (-1.2)**	20.5 (+1.0)	25.2 (+5.8)
ymca	17.8 bits	24.3 (+6.6)	21.4 (+3.6)	18.2 (+0.4)	**16.4 (-1.4)**	**16.7 (-1.1)**	18.5 (+0.7)	23.7 (+5.9)
semester	19.4 bits	26.5 (+7.0)	20.3 (+0.9)	**18.7 (-0.7)**	**17.4 (-2.0)**	**18.0 (-1.4)**	19.6 (+0.1)	25.6 (+6.1)
divorced	20.6 bits	27.7 (+7.1)	**19.3 (-1.3)**	**18.4 (-2.1)**	**17.7 (-2.9)**	**18.3 (-2.3)**	**19.1 (-1.5)**	25.9 (+5.3)
absent	21.0 bits	28.0 (+7.0)	**18.2 (-2.8)**	**18.2 (-2.8)**	**17.6 (-3.4)**	**18.3 (-2.7)**	**18.7 (-2.3)**	25.8 (+4.8)
certified	16.9 bits	21.6 (+4.8)	17.3 (+0.5)	**15.4 (-1.5)**	**14.9 (-1.9)**	**15.4 (-1.4)**	**16.0 (-0.8)**	19.1 (+2.2)
streets	17.6 bits	22.7 (+5.1)	**16.3 (-1.3)**	**15.8 (-1.8)**	**15.1 (-2.4)**	**15.5 (-2.1)**	**16.5 (-1.1)**	18.3 (+0.7)
church	13.1 bits	16.1 (+2.9)	15.4 (+2.2)	**12.9 (-0.2)**	**11.8 (-1.3)**	**12.2 (-0.9)**	13.3 (+0.1)	14.8 (+1.6)
agreed	15.6 bits	20.4 (+4.7)	**14.2 (-1.4)**	**14.2 (-1.4)**	**13.7 (-2.0)**	**13.9 (-1.7)**	**14.7 (-0.9)**	15.9 (+0.2)
director	14.3 bits	17.7 (+3.4)	**13.3 (-1.0)**	**12.9 (-1.4)**	**12.4 (-1.8)**	**12.7 (-1.6)**	**13.2 (-1.0)**	14.5 (+0.2)
high	13.9 bits	16.5 (+2.6)	**12.3 (-1.6)**	**12.2 (-1.6)**	**11.7 (-2.1)**	**11.9 (-1.9)**	**12.6 (-1.3)**	14.0 (+0.2)
there	13.7 bits	15.3 (+1.6)	**11.2 (-2.5)**	**11.5 (-2.2)**	**11.0 (-2.7)**	**11.1 (-2.6)**	**11.8 (-1.9)**	**13.0 (-0.7)**
which	13.1 bits	14.1 (+1.1)	**10.2 (-2.9)**	**10.7 (-2.4)**	**10.1 (-2.9)**	**10.3 (-2.8)**	**10.8 (-2.2)**	**11.5 (-1.5)**

Table 2: Compression effectiveness (in bits per posting) for schema-independent posting lists extracted from TREC disks 1-5. All numbers are relative to vByte.

All executables were compiled using gcc 3.4.2 (x86-64, -O1). As test data, we chose the following three text collections:

- The INEX-1.4 XML collection, containing 88 million tokens in 12,107 text articles;
- TREC disks 1-5, containing 823 million tokens in 1.5 million documents;
- TREC GOV2, containing 42 billion tokens in 25.2 million documents.

Most of our analysis is based on a certain set of terms and their posting lists in the *TREC disks 1-5* collection. A list of these terms, covering most of the whole range of frequent and infrequent terms, is given by Table 1.

Compression Effectiveness

For TREC disks 1-5, we constructed two indices: a document-level inverted file, in which every posting corresponds to a document identifier, without any frequency information or positional information, and a schema-independent inverted file with full positional information. We extracted the posting lists of the 15 terms listed in Table 1 and compressed them using various encoding methods.

The methods tested were vByte [15], Elias γ [7], Golomb-Rice codes [8], the interpolative method [11], Huffman, GUBC-3, the RBUC-B compression scheme recently proposed by Moffat and Anh [10] and the word-aligned Carryover-12 method presented by Anh and Moffat [2]. Numbers for RBUC-B and Carryover-12 were obtained by using the implementations made available on the authors' website[12]. The effectiveness results for all methods on TREC disks 1-5 are shown in Figure 6.

For the document-level index, it can be seen that, with the exception of vByte and Carryover-12, all methods perform reasonably well. When comparing Interpolative, Golomb-Rice, Huffman, GUBC-3, and RBUC-B coding, the differences in per-posting compression effectiveness are less than 1 bit for every term.

For the schema-independent index, the situation is completely different. Figure 6(b) shows that vByte, Carryover-12, and Golomb-Rice are much worse than the other codes. For the former two, this is not unexpected, as their being aligned codes somewhat restricts their flexibility and their ability to adjust to changes in the d-gap distribution. The reason why Rice code is outperformed by the other methods is that its underlying assumption – geometric d-gap distribution – is wrong for a schema-independent index. The Rice coder exhibits particularly poor performance for terms that have a high tendency to form clusters, such as "church", "corsica", and "ymca", as shown in Table 2. All three terms have an expected within-document term frequency of more than 1.5 — if they occur in a document, they usually occur more than once (cf. Table 1).

With the exception of Carryover-12, which only yields

9[1] http://www.cs.mu.oz.au/∼alistair/carry/

9[2] http://www.cs.mu.oz.au/∼alistair/rbuc/

	vByte	Gamma	Rice	Interpol	Huffman	GUBC-3	Gamma*	Interpol*
landfilling	9.6 ns	23.1 (+13.5)	11.3 (+1.7)	33.5 (+23.9)	79.3 (+69.7)	13.5 (+3.9)	89.4 (+79.8)	124.8 (+115.2)
rattlesnake	9.9 ns	23.4 (+13.5)	**9.1 (-0.7)**	32.6 (+22.8)	43.6 (+33.7)	12.3 (+2.4)	90.2 (+80.3)	126.2 (+116.3)
corsica	9.5 ns	20.6 (+11.1)	**8.0 (-1.5)**	33.8 (+24.3)	28.8 (+19.3)	12.1 (+2.6)	78.0 (+68.5)	118.7 (+109.2)
ymca	8.0 ns	19.8 (+11.8)	8.2 (+0.2)	33.4 (+25.4)	19.9 (+11.9)	11.4 (+3.4)	72.7 (+64.7)	109.5 (+101.5)
semester	7.9 ns	20.4 (+12.5)	8.9 (+1.0)	31.7 (+23.9)	15.1 (+7.3)	11.0 (+3.1)	77.3 (+69.4)	112.4 (+104.5)
divorced	7.3 ns	19.6 (+12.4)	8.2 (+0.9)	30.1 (+22.9)	12.5 (+5.3)	9.2 (+1.9)	79.6 (+72.4)	111.0 (+103.7)
absent	7.6 ns	19.3 (+11.7)	7.8 (+0.2)	29.5 (+22.0)	10.1 (+2.5)	9.0 (+1.4)	80.8 (+73.2)	110.5 (+102.9)
certified	7.9 ns	18.3 (+10.5)	**7.8 (-0.1)**	29.8 (+21.9)	11.0 (+3.2)	9.0 (+1.2)	67.1 (+59.3)	98.4 (+90.6)
streets	8.2 ns	18.0 (+9.8)	**7.5 (-0.8)**	28.1 (+19.8)	10.1 (+1.9)	8.8 (+0.5)	70.0 (+61.7)	100.0 (+91.8)
church	8.0 ns	16.6 (+8.5)	8.5 (+0.4)	34.2 (+26.1)	10.0 (+1.9)	9.2 (+1.1)	55.6 (+47.6)	91.1 (+83.0)
agreed	7.5 ns	17.8 (+10.3)	8.7 (+1.2)	33.4 (+25.9)	10.6 (+3.2)	9.2 (+1.7)	66.0 (+58.5)	100.1 (+92.6)
director	7.5 ns	17.2 (+9.7)	8.8 (+1.3)	34.1 (+26.6)	10.9 (+3.4)	9.2 (+1.7)	60.1 (+52.5)	94.2 (+86.7)
high	7.7 ns	17.1 (+9.4)	8.4 (+0.7)	35.2 (+27.5)	9.4 (+1.7)	9.0 (+1.4)	57.4 (+49.7)	91.7 (+84.0)
there	7.8 ns	16.4 (+8.6)	8.1 (+0.3)	34.7 (+26.9)	9.0 (+1.2)	8.6 (+0.8)	53.2 (+45.4)	88.5 (+80.7)
which	7.9 ns	14.5 (+6.7)	8.0 (+0.2)	33.3 (+25.4)	9.0 (+1.1)	8.5 (+0.5)	49.6 (+41.7)	85.0 (+77.2)

Table 3: Decoding performance (in nanoseconds per posting) for schema-independent posting lists extracted from TREC disks 1-5. All numbers are relative to vByte. Columns marked with a star represent our initial implementation of the respective method, without applying the techniques described in section 6.

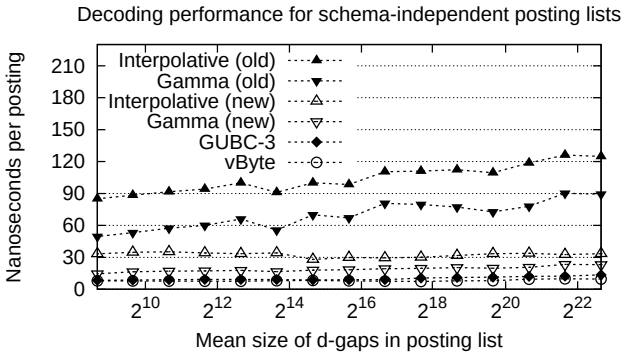

Figure 7: Decoding performance of several compression methods for schema-independent posting lists from TREC disks 1-5. With bit buffering and selector look-ahead, the decoding process for Elias γ is 3-4 times faster than without these techniques.

good compression rates for high-frequency terms (or for document-level indices), and Gamma, which is simply not competitive with the other methods, the other algorithms all produce decent results, usually better than vByte. However, the only methods that consistently achieve a better compression rate than vByte are Huffman and GUBC-3. Huffman saves between 2 and 3 bits per posting for most terms, GUBC-3 a little less.

Decoding Performance

To test the decoding efficiency of the various coding schemes, we took the posting list for each term, encoded it in each format, and repeatedly ran the decoding routine for the respective method. In total, for each posting list and each coding scheme, several billion postings were decompressed in order to obtain reliable performance figures.

We compared the decoding performance of vByte to that of γ and interpolative coding. For each of the latter two, we ran experiments with our original implementation, which performs bitwise decoding operations and does not make any use of the techniques described in section 6, and with our new implementation based on bit buffering. vByte's decoding routine needs between 7 and 10 nanoseconds per posting; the unoptimized versions of γ and interpolative coding between 50 and 120 nanoseconds – a factor-10 difference, which is in line with performance figures reported elsewhere [2] [16]. Our new implementations of these two methods, however, are much faster. The new γ decoder only needs between 14 and 24 nanoseconds per posting, the interpolative decoder between 28 and 36 nanoseconds – a slowdown between 100% and 300%, compared to vByte.

The decoders for Huffman and GUBC-3 are even more efficient and only require around 10 nanoseconds per posting. Compared to vByte, this is a per-posting slowdown between 1 and 3 nanoseconds. The optimized Rice decoder even outperforms vByte on some lists. The Huffman decoder, however, is only efficient for long posting lists, containing at least a few thousand postings. For shorter lists, the overhead associated with constructing the selector look-ahead table by far outweighs the cost of the actual decoding process.

Overall Impact on Search Engine Performance

The effect that the different compression algorithms have on the total size on the index for each text corpus is shown in Table 4. For comparison, the table includes the size of an uncompressed index, where each posting is stored as a simple 32-bit integer. Huffman and GUBC-3 achieve the best compression among all methods, around 85% the size of a vByte-compressed index (35% of an uncompressed index), with Huffman producing slightly better results than GUBC-3. The Golomb-Rice coder shrinks the index by 7% for TREC disks 1-5, but only achieves 2% for the INEX collection and increases the size of the index for GOV2. This is because the average document length for INEX (7,000 tokens) and GOV2 (1,700 tokens) is larger than for the TREC disks (550 tokens), leading to greater clustering due to within-document repetitions, a gross violation of the independence assumption that Golomb-Rice is based on.

We tested query processing performance using title-only queries extracted from TREC topics 1-500 (for TREC disks 1-5) and the first 1000 topics of the TREC 2005 Terabyte efficiency query stream (for GOV2). For each search query, we had our search engine report the 20 top-ranking documents according to their Okapi BM25 score. Each experiment was repeated multiple times in order to obtain reliable results. Average query times are shown in Table 5.

For TREC disks 1-5, vByte, Rice, Huffman, and GUBC-3 all achieve equivalent query processing performance. Only the interpolative method, due to the complex decoding operationgs, and the uncompressed index, due to the substan-

	INEX-1.4	TREC 1-5	GOV2
vByte	145.4 MB	1361.6 MB	61.3 GB
Golomb-Rice	141.9 MB	1261.1 MB	61.7 GB
Interpolative	132.2 MB	1216.2 MB	54.2 GB
Huffman	124.2 MB	1144.4 MB	50.6 GB
GUBC-3	126.8 MB	1170.7 MB	50.8 GB
None (32-bit)	360.4 MB	3218.5 MB	n/a

Table 4: Total size of compressed inverted file.

	TREC disks 1-5	GOV2
vByte	96.2 ms/query	1792.6 ms/query
Golomb-Rice	96.4 ms/query	1866.2 ms/query
Interpolative	110.7 ms/query	2445.0 ms/query
Huffman	96.1 ms/query	1783.6 ms/query
GUBC-3	96.0 ms/query	1781.6 ms/query
None (32-bit)	115.6 ms/query	n/a

Table 5: Impact on query processing performance.

tially increased disk I/O, are significantly slower than the other methods (15% and 20%, respectively). For the GOV2 text collection, the situation is roughly the same. However, the Rice coder, because it increases the size of the index compared to vByte, is about 4% slower than vByte. Huffman and GUBC-3, on the other hand, are able to generate a tiny improvement over vByte: Query processing is 0.5% (Huffman) and 0.6% (GUBC-3) faster than with vByte. For both text collections, the TREC disks and GOV2, the GUBC-3 compression method provides the highest query processing performance among all methods tested.

8. CONCLUSION

We have discussed the effectiveness of index compression techniques for schema-independent inverted files and proposed an unaligned encoding method, GUBC-3, that takes the special characteristics of such inverted files into account. In addition, we have presented performance-enhancing techniques for unaligned index compression schemes. As a result, we have obtained an unaligned compression algorithm that produces indices about 15% smaller than the well-known vByte method, while offering an equivalent, or even slightly increased, level of query processing performance. Our results contradict most earlier work on index compression algorithms for inverted files, which usually indicated that the query processing performance of aligned codes is much better than that of unaligned codes.

The evaluation presented in this paper is limited to schema-independent inverted files, but our methods can easily be applied to document-level indices. The techniques we use to obtain high-performance decoding routines for unaligned binary codes rely on the machine word size on the computer they are run on. While they produce very good results on a 64-bit architecture, they will be less efficient on 32-bit CPUs. However, we do not deem this a real limitation, as 32-bit CPUs are already beginning to disappear.

9. REFERENCES

[1] V. N. Anh and A. Moffat. Index Compression Using Fixed Binary Codewords. In *Proceedings of the Fifteenth Conference on Australasian Database*, pages 61–67, Dunedin, New Zealand, January 2004.

[2] V. N. Anh and A. Moffat. Inverted Index Compression using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, January 2005.

[3] S. Büttcher and C. L. A. Clarke. A Security Model for Full-Text File System Search in Multi-User Environments. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 2005)*, San Francisco, USA, December 2005.

[4] C. L. A. Clarke. Controlling Overlap in Content-Oriented XML Retrieval. In *Proceedings of the 28th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 314–321, Salvador, Brazil, August 2005.

[5] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An Algebra for Structured Text Search and a Framework for Its Implementation. Technical report, University of Waterloo, August 1994.

[6] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. Schema-Independent Retrieval from Heterogeneous Structured Text. Technical report, University of Waterloo, November 1994.

[7] P. Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[8] S. Golomb. Run-Length Encodings. *IEEE Trans. on Information Theory*, IT-12:399–401, July 1966.

[9] R. L. Milidiú and E. S. Laber. Improved Bounds on the Inefficiency of Length Restricted Codes. Technical report, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brazil, January 1997.

[10] A. Moffat and V. N. Anh. Binary Codes for Non-Uniform Sources. In *Proceedings of the 15th Data Compression Conference (DCC 2005)*, pages 133–142, Snowbird, USA, March 2005.

[11] A. Moffat and L. Stuiver. Exploiting Clustering in Inverted File Compression. In Storer and Cohn, editors, *Proceedings of the 1996 Data Compression Conference*, pages 82–91, 1996.

[12] A. Moffat and L. Stuiver. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval*, 3(1):25–47, 2000.

[13] A. Moffat and J. Zobel. Compression and Fast Indexing for Multi-Gigabyte Text Databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[14] A. Moffat and J. Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[15] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th ACM SIGIR Conference on Research and Development in Information Retrieval*, Tampere, Finland, August 2002.

[16] A. Trotman. Compressing Inverted Files. *Information Retrieval*, 6(1):5–19, January 2003.

[17] H. E. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, 1999.