Compression of Inverted Indexes For Fast Query Evaluation

Falk Scholer Hugh E. Williams John Yiannis Justin Zobel
School of Computer Science and Information Technology
RMIT University, GPO Box 2476V
Melbourne, Australia, 3001.
{fscholer,hugh,jyiannis,jz}@cs.rmit.edu.au

ABSTRACT

Compression reduces both the size of indexes and the time needed to evaluate queries. In this paper, we revisit the compression of inverted lists of document postings that store the position and frequency of indexed terms, considering two approaches to improving retrieval efficiency: better implementation and better choice of integer compression schemes. First, we propose several simple optimisations to well-known integer compression schemes, and show experimentally that these lead to significant reductions in time. Second, we explore the impact of choice of compression scheme on retrieval efficiency.

In experiments on large collections of data, we show two surprising results: use of simple byte-aligned codes halves the query evaluation time compared to the most compact Golomb-Rice bitwise compression schemes; and, even when an index fits entirely in memory, byte-aligned codes result in faster query evaluation than does an uncompressed index, emphasising that the cost of transferring data from memory to the CPU cache is less for an appropriately compressed index than for an uncompressed index. Moreover, byte-aligned schemes have only a modest space overhead: the most compact schemes result in indexes that are around 10% of the size of the collection, while a byte-aligned scheme is around 13%. We conclude that fast byte-aligned codes should be used to store integers in inverted lists.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software Performance evaluation (efficiency and effectiveness); E.4 [**Data**]: Coding and Information Theory Data compaction and compression

General Terms

Performance, Algorithms, Design

SIGIR'02, August 11-15, 2002, Tampere, Finland.

Keywords

Inverted indexes, retrieval efficiency, index compression, integer coding

1. INTRODUCTION

Search engines have demanding performance requirements. Users expect fast answers to queries, many queries must be processed per second, and the quantity of data that must be searched in response to each query is staggering. The demands continue to grow: the Google search engine, for example, indexed around one billion documents a year ago and now manages more than double that figure[1]. Moreover, the increasing availability and affordability of large storage devices suggests that the amount of data stored online will continue to grow.

Inverted indexes are used to evaluate queries in all practical search engines [14]. Compression of these indexes has three major benefits for performance. First, a compressed index requires less storage space. Second, compressed data makes better use of the available communication bandwidth; more information can be transfered per second than when the data is uncompressed. For fast decompression schemes, the total time cost of transfering compressed data and subsequently decompressing is potentially much less than the cost of transferring uncompressed data. Third, compression increases the likelihood that the part of the index required to evaluate a query is already cached in memory, thus entirely avoiding a disk access. Thus index compression can reduce costs in retrieval systems.

We have found that an uncompressed inverted index that stores the location of the indexed words in web documents typically consumes more than 30% of the space required to store the uncompressed collection of documents. (Web documents often include a great deal of information that is not indexed, such as HTML tags; in the TREC web data, which we use in our experiments, on average around half of each document is indexable text.) When the index is compressed, the index size is reduced to between 10%–15% of that required to store the uncompressed collection; this size includes document numbers, in-document frequencies, and word positions within documents. If the index is too large to fit entirely within main memory, then querying the uncompressed index is slower: as we show later, it is up to twice as slow as the fastest compressed scheme.

In this paper, we revisit compression schemes for the in-

[1]See `http://www.google.com/`

verted list component of inverted indexes. We also propose a new method for decoding lists. There have been a great many reports of experiments on compression of indexes with bitwise compression schemes [6, 8, 12, 14, 15], which use an integral number of bits to represent each integer, usually with no restriction on the alignment of the integers to byte or machine-word boundaries. We consider several aspects of these schemes: how to decode bitwise representations of integers efficiently; how to minimise the operations required for the most compact scheme, Golomb coding; and the relative performance of Elias gamma coding, Elias delta coding, Golomb coding, and Rice coding for storing indexes.

We question whether bitwise compression schemes are the best choice for storing lists of integers. As an alternative, we consider bytewise integer compression schemes, which require that each integer is stored in an integral number of blocks, where each block is eight bits. The length of each stored integer can therefore be measured in an exact number of bytes. An additional restriction is to require that these eight-bit blocks must align to machine-word or byte boundaries. We propose and experimentally investigate several variations of bytewise schemes.

We investigate the performance of different index compression schemes through experiments on large query sets and collections of Web documents. We report two surprising results.

- For a 20 gigabyte collection, where the index is several times larger than main memory, optimised bytewise schemes more than halve the average decoding time compared to the fastest bitwise approach.
- For a much smaller collection, where the index fits in main memory, a bytewise compressed index can still be processed faster than an uncompressed index.

These results show that effective use of communication bandwidths is important for not only disk-to-memory transfers but also memory-to-cache transfers. The only disadvantage of bytewise compressed indexes is that they are up to 30% larger than bitwise compressed indexes; the smallest bitwise index is around 10% of the uncompressed collection size, while the bytewise index is around 13%.

2. INVERTED INDEXES

An inverted index consists of two major components: the *vocabulary* of terms—for example the words—from the collection, and *inverted lists*, which are vectors that contain information about the occurrence of the terms [14].

In a basic implementation, for each term t there is an inverted list that contains *postings* $< f_{d,t}, d >$ where $f_{d,t}$ is the frequency f of term t in the ordinal document d. One posting is stored in the list for each document that contains the term t. Inverted lists of this form—along with additional statistics such as the document length l_d, and f_t, the number of documents that contain the term t—are sufficient to support ranked and Boolean query modes.

To support phrase querying or proximity querying, additional information must be kept in the inverted lists. Thus inverted list postings should be of the form

$$< f_{d,t}, d, [o_{0,d,t} \ldots o_{f_{d,t},d,t}] >$$

The additional information is the list of offsets o; one offset is stored for each of the $f_{d,t}$ occurrences of term t in document d. Postings in inverted lists are usually ordered by increasing d, and the offsets likewise ordered within the postings by increasing o. This has the benefit that differences between values—rather than the raw values—can be stored, improving the compressibility of the lists.

Other arrangements of the postings in lists are useful when lists are not necessarily completely processed in response to a query. For example, in frequency-sorted indexes [9, 10] postings are ordered by $f_{d,t}$, and in impact-ordered indexes the postings are ordered by quantised weights [1]. These approaches also rely on compression to help achieve efficiency gains, and the improvements to compression performance we describe in this paper are as applicable to these methods as they are to the simple index representations we use as a testbed for our compression methods.

Consider an example inverted list with offsets for the term "Matthew":

$$< 3, 7, [6, 51, 117] >< 1, 44, [12] >< 2, 117, [14, 1077] >$$

In this index, the terms are words, the offsets are word positions within the documents, and the lists are ordered by d. This inverted list states that the term "Matthew" occurs 3 times in document 7, at offsets 6, 51, and 117. It also occurs once in document 44 at offset 12, and twice in document 117, at offsets 14 and 1077.

Ranked queries can be answered using the inverted index as follows. First, the terms in the user's query are located in the inverted index vocabulary. Second, the corresponding inverted lists for each term are retrieved from disk, and then processed by decreasing f_t. Third, for each posting in each inverted list, an accumulator weight A_d is increased; the magnitude of the increase is dependent on the similarity measure used, and can consider the weight $w_{q,t}$ of term t in the query q, the weight $w_{d,t}$ of the term t in the document d, and other factors. Fourth, after processing part [1, 6] or all of the lists, the accumulator scores are partially sorted to identify the most similar documents. Last, for a typical search engine, document summaries of the top ten documents are generated or retrieved and shown to the user. The offsets stored in each inverted list posting are not used in ranked query processing.

Phrase queries require offsets and that a given sequence of words be contiguous in a matching document. For example, consider a combined ranked and phrase query:

"Matthew Richardson" Richmond

To evaluate such a query, the same first two steps as for ranked querying are applied. Then, instead of accumulating weights, it is necessary to construct a temporary inverted list for the phrase, by fetching the inverted list of each of the individual terms and combining them. If the inverted list for "Matthew" is as above and the inverted list for "Richardson" is

$$< 1, 7, [52] > < 2, 12, [1, 4] > < 1, 44, [83] >$$

then both words occur in document 7 and as an ordered pair. Only the word "Richardson" is in document 12, both words occur in document 44 but not as a pair, and only "Matthew" occurs in document 117. The list for "Matthew Richardson" is therefore

$$< 1, 7, [51] >$$

After this, the ranking process is continued from the third step, where the list for the term "Richmond" and the newly created list are used to adjust accumulator weights. Phrase queries can involve more than two words.

3. COMPRESSING INVERTED INDEXES

Special-purpose integer compression schemes offer both fast decoding and compact storage of inverted lists [13, 14]. In this section, we consider how inverted lists are compressed and stored on disk. We limit our discussions here to the special-purpose integer compression techniques that have previously been shown to be suitable for index compression, and focus on their use in increasing the speed of retrieval systems.

Without compression, the time cost of retrieving inverted lists is the sum of the time taken to seek for and then retrieve the inverted lists from disk into memory, and the time taken to transfer the lists from memory into the CPU cache before they are processed. The speed of access to compressed inverted lists is determined by two factors: first, the computational requirements for decoding the compressed data and, second, the time required to seek for and retrieve the compressed data from disk and to transfer it to the CPU cache before it is decoded. For a compression scheme to allow faster access to inverted lists, the total retrieval time and CPU processing costs should be less than the retrieval time of the uncompressed representation. However, a third factor makes compression attractive even if CPU processing costs exceed the saving in disk transfer time: compressing inverted lists increases the number of lists that can be cached in memory between queries, so that in the context of a stream of queries use of compression reduces the number of disk accesses. It is therefore important that a compression scheme be efficient in both decompression CPU costs and space requirements.

There are two general classes of compression scheme that are appropriate for storing inverted lists. Variable-bit or *bitwise* schemes store integers in an integral number of bits. Well-known bitwise schemes include Elias gamma and delta coding [3] and Golomb-Rice coding [4]. *Bytewise* schemes store an integer in an integral number of blocks, where a block is eight bits in size; we distinguish between blocks and bytes here, since there is no implied restriction that a block must align to a physical byte-boundary. A simple bytewise scheme is variable-byte coding [2, 13]; uncompressed integers are also stored in an integral number of blocks, but we do not define them as bytewise schemes since, on most architectures, an integer has a fixed-size representation of four bytes. In detail, these schemes are as follows.

Elias coding [3] is a non-parameterised bitwise method of coding integers. (Non-parameterised methods use static or fixed codes to store integers.) The Elias gamma code represents a positive integer k by $1 + \lfloor \log_2 k \rfloor$ stored as a unary code, followed by the binary representation of k without its most significant bit. Using Elias gamma coding, small integers are compactly represented; in particular, the integer 1 is represented as a single 1-bit. Gamma coding is relatively inefficient for storing integers larger than 15 [13].

Elias delta codes are suited to coding larger integers, but are inefficient for small values. For an integer k, a delta code stores the gamma code representation of $1 + \lfloor \log_2 k \rfloor$, and then the binary representation of k without its most significant bit.

Golomb-Rice bitwise coding [4] has been shown to offer more compact storage of integers and faster retrieval than the Elias codes [13]; indeed, it is bitwise optimal under the assumption that the set of documents with a given term is random. The codes are adapted to per-term likelihoods via a parameter that is used to determine the code emitted for an integer. In many cases, this parameter must be stored separately using, for example, an Elias code. For coding of inverted lists, a single parameter is used for all document numbers in a postings list, but each posting requires a parameter for its offsets. The parameters can be calculated as the lists are decoded using statistics stored in memory and in the lists, as we discuss later.

Coding of an integer k using Golomb codes with respect to a parameter b is as follows. The code that is emitted is in two parts: first, the unary code of a quotient q is emitted, where $q = \lfloor (k-1)/b \rfloor + 1$; second, a binary code is emitted for the remainder r, where $r = k - q \times b - 1$. The number of bits required to store the remainder r is either $\lceil \log_2 b \rceil$ or $\lfloor \log_2 b \rfloor$. To retrieve the remainder, the value of the "toggle point" $t = 1 \triangleleft ((\log_2 k) + 1)) - b$ is required, where $\triangleleft$ indicates a left-shift operation. After retrieving $\lfloor \log_2 b \rfloor$ bits of the remainder r, the remainder is compared to t. If $r > t$, then one additional bit of the remainder must be retrieved. It is generally thought that caching calculated values of $\log_2 b$ is necessary for fast decoding, with a main-memory penalty of having to store the values. However, as we show later, when the standard log library function is replaced with a fast bit-shifting version, caching is unnecessary.

Rice coding is a variant of Golomb coding where the value of b is restricted to be a power of 2. The advantage of this restriction is that there is no "toggle point" calculation required, that is, the remainder is always stored in exactly $\lceil \log_2 b \rceil$ bits. The disadvantage of this scheme is that the choice of value for b is restricted and, therefore, the compression is slightly less effective than that of Golomb coding.

For compression of inverted lists, a value of b is required. Witten et al. [14] report that for cases where the probability of any particular integer value occurring is small—which is the usual case for document numbers d and offsets o—then b can be calculated as:

$$b = 0.69 \times \text{mean}(k)$$

For each inverted list, the mean value of document numbers d can be approximated as $k = N/f_t$ where N is the number of documents in the collection and f_t is the number of postings in the inverted list for term t [14]. This approach can also be extended to offsets: the mean value of offsets o for an inverted list posting can be approximated as $k = l_d/f_{d,t}$ where l_d is the length of document d and $f_{d,t}$ is the number of offsets of term t within that document. As the statistics N, f_t, and l are often available in memory, or in a simple auxiliary structure on disk, storage of b values is not required for decoding; approximate values of l can be stored in memory for compactness [7], but use of approximate values has little effect on compression effectiveness as it leads to only small relative errors in computation of b.

In bytewise coding an integer is stored in an integral number of eight-bit blocks. For variable-byte codes, seven bits in each block are used to store a binary representation of the integer k. The remaining bit is used to indicate whether the current block is the final block for k, or whether an additional block follows. Consider an example of an integer k

in the range of $2^7 = 128$ to $2^{14} = 16,384$. Two blocks are required to represent this integer: the first block contains the seven least-significant bits of the integer and the eighth bit is used to flag that another block follows; the second block contains the remaining most-significant bits and the eighth bit flags that no further blocks follow. We use the convention that the flag bit is set to 1 in the final block and 0 otherwise.

Compressing an inverted index, then, involves choosing compression schemes for the three kinds of data that are stored in a posting: a document number d, an in-document frequency $f_{d,t}$, and a sequence of offsets o. A standard choice is to use Golomb codes for document numbers, gamma codes for frequencies, and delta codes for offsets [14]. (We explore the properties of this choice later.) In this paper, we describe such a choice as a GolD-GamF-DelO index.

3.1 Fast Decoding

We experiment with compression of inverted lists of postings that contain frequencies $f_{d,t}$, documents numbers d, and offsets o. For fast decompression of these postings, there are two important considerations: first, the choice of compression scheme for each component of the posting; and, second, modifications to each compression scheme so that it is both fast and compatible with the schemes used for the other components. In this section, we outline the optimisations we use for fast decompression. Our code is publically available and distributed under the GNU public licence.[2]

Bitwise Compression

We have experimented with a range of variations of bitwise decompression schemes. Williams and Zobel [13] reported results for several efficient schemes, where vectors that contain compressed integers are retrieved from disk and subsequently decoded.[3] In their approach, vector decoding uses bitwise shift operations, bit masks, multiplication, subtraction, and function calls to retrieve sequences of bits that span byte boundaries. In our experiments on Intel Pentium-based servers running the Linux operating system, we have found that bitwise shift operations are usually faster than bit masks, and that the function calls are slow. By optimising our code to use bitwise shifts and to remove nested function calls, we have found that the overall time to decode vectors—regardless of the compression scheme used—is on average around 60% of that using the code of Williams and Zobel.

Other optimisations that are specific to Golomb-Rice coding are also of value. Golomb-Rice decoding requires that $\log_2 b$ is calculated to determine the number of remainder bits to be retrieved. It is practicable to explicitly cache values of $\log_2 b$ in a hash table as they are calculated, or to pre-calculate all likely-to-be-used values as the retrieval query engine is initialised. This saves recalculation of logarithms when a value of b is reused in later processing, with the penalty of additional memory requirements for storing the lookup table.

We measured the performance of Golomb coding with and without caching. Timings are average elapsed query evaluation cost to process index information for 25,000 queries on a 9.75 gigabyte (Gb) collection of Web data [5] using our prototype retrieval engine on a GolD-GamF-GolO index (that is, Golomb document numbers, gamma frequencies, Golomb offsets); we discuss collection statistics and experimental design further in Section 4. The cache lookup table size is unrestricted.

We found that, without caching of $\log_2 b$ values, the average query evaluation time is 0.961 seconds. Caching of $\log_2 b$ values as they are calculated during query processing roughly halves the average query evaluation time, to 0.494 seconds. Pre-calculating and storing the values offers almost no benefit over caching during query processing, reducing the time to 0.491 seconds; this reflects that only limited b values are required during query evaluation. Caching of toggle points yields 0.492 seconds. As toggle points are calculated using bitwise shifts, addition, and subtraction, this is further evidence that bitwise shifts are inexpensive on our hardware.

An alternative approach to managing log computations is to replace the standard library log function with a loop that determines $\lfloor \log_2 b \rfloor$ using bitwise shifts and equality tests; the logarithm value can be determined by locating the position of the most-significant 1-bit in b. We found that this led to slight additional improvements in the speed of decoding Golomb codes, outperforming explicit caching. All Golomb-Rice coding results reported in this paper are computed in this way.

Bytewise Compression

We have experimented with improvements to variable-byte coding. Unlike in bitwise coding, we have found that masking and shifting are equally as fast because of the large number of shifts required. We use shifts in our experiments.

Perhaps the most obvious way to increase the speed of variable-byte decoding is to align the eight-bit blocks to byte boundaries. Alignment with byte boundaries limits the decoding to only one option: the flag bit indicating if this is the last byte in the integer is always the most significant bit, and the remaining seven bits contain the value. Without byte alignment, additional conditional tests and operations are required to extract the flag bit, and the seven-bit value can span byte boundaries. We would expect that byte alignment would improve the speed of decoding variable-byte integers.

Figure 1 shows the effect of byte alignment of variable-byte integers. In this experiment, variable-byte coding is used to store the offsets o in each inverted list posting. The optimised Golomb coding scheme described in the previous section is used to code document numbers d and Elias gamma coding is used to store the frequencies $f_{d,t}$. We refer to this as a GolD-GamF-VbyO index.

The graph at the left of Figure 1 shows total index size as a percentage of the uncompressed collection being indexed. The first bar shows that, without byte alignment, the GolD-GamF-VbyO index requires almost 13% of the space required by the collection. The second bar shows that padding to byte alignment after storing the Gamma-coded $f_{d,t}$ values increases the space requirement to just over 13.5% of the collection size. We discuss the other schemes in this figure later in this section.

The graph at the right of Figure 1 shows elapsed query evaluation times using different index designs. Timings are

[2]The search engine used in these experiments and our integer compression code is available from `http://www.seg.rmit.edu.au/`

[3]The code used by Williams and Zobel in their experiments is available from `http://www.cs.rmit.edu.au/~hugh/software/`

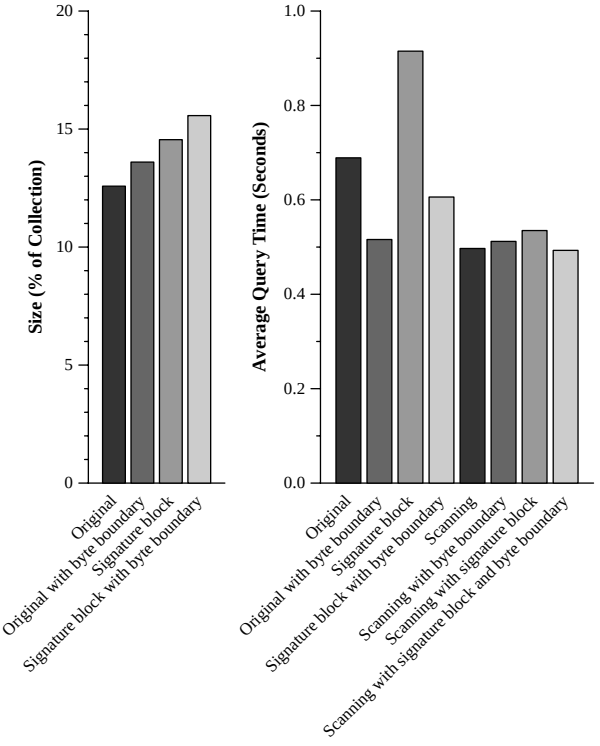

Figure 1: Variable-byte schemes for compressing offsets in inverted lists in a GolD-GamF-VbyO index. Four different compression schemes are shown and, for each, both original and scanning decoding are shown. Scanning decoding can be used when offsets are not needed for query resolution.

the average elapsed query evaluation cost to process the inverted lists for 25,000 queries on a 20 Gb collection of Web [5] data using our prototype retrieval engine. Queries are processed as conjunctive Boolean queries. The first bar shows that the average time is around 0.7 seconds for the GolD-GamF-VbyO index without byte alignment. The second bar shows that the effect of byte alignment is a 25% reduction in average query time. Therefore, despite the small additional space requirement, byte-alignment is beneficial when storing variable-byte integers.

A second optimisation to variable-byte coding is to consider the query mode when processing the index. For querying that does not use offsets—such as ranked and Boolean querying—decoding of the offsets in each posting is unnecessary. Rather, all that is required are the document numbers d and document frequencies $f_{d,t}$. An optimisation is therefore to only examine the flag bit of each block and to ignore the remaining seven bits that contain the value. The value of $f_{d,t}$ indicates the number of offsets o stored in the posting. By examining flag bits until $f_{d,t}$ 1-bits are processed, it is possible to bypass the offsets with minimal processing. We call this approach *scanning*.

Scanning can also be used in query modes that do require offset decoding. As we discussed earlier, phrase querying requires that all terms are present in a matching document. After processing the inverted list for the first term that is evaluated in a phrase query, a temporary inverted list of postings is created. This temporary list has a set D of documents that contain the first term. When processing the second term in the query, a second set of document numbers D' are processed. Offsets for the posting associated with document $d \in D'$ can be scanned, that is, passed over without decoding, if d is not a member of D. (At the same time, document numbers in D that are not in D' are discarded.)

We show the performance of scanning in Figure 1. The fifth and sixth bars show how scanning affects query evaluation time for variable-bytes that are either unaligned and aligned to byte boundaries in the GolD-GamF-VbyO index. Scanning removes the processing of seven-bit values. This reduces the cost of retrieving unaligned variable-bytes to less than that of the aligned variable-byte schemes; the small speed advantage is due to the retrieval of smaller lists in the unaligned version. Scanning has little effect on byte-aligned variable bytes, reflecting that the processing of seven-bit values using shift operations has a low cost. Overall, however, byte-alignment is preferred since the decoding cost of offsets is expensive in an unaligned scheme.

A third optimisation is an approach we call *signature blocks*, which are a variant of *skipping*. Skipping is the approach of storing additional integers in inverted lists that indicate how much data can be skipped without any processing [14]. Skipping has the disadvantage of an additional storage space requirement, but has been shown to offer substantial speed improvements [14]. A signature block is an eight-bit block that stores the flag bits of up to eight blocks that follow. For example, a signature block with the bit-string 11100101 represents that five integers are stored in the eight following eight-bit blocks: the string 111 represents that the first three blocks store one integer each; the string 001 represents that the fourth integer is stored over three blocks; and, the string 01 represents that the final integer is stored over two blocks. As all flag bits are stored in the signature block, the following blocks use all eight bits to store values, rather the seven-bit scheme in the standard variable-byte integer representation.

The primary use of signature blocks is skipping. To skip offsets, $f_{d,t}$ offset values must be retrieved but not processed. By counting the number of 1-bits in a signature block, the number of integers stored in the next eight blocks can be determined. If the value of $f_{d,t}$ exceeds this, then a second or subsequent signature block is processed until $f_{d,t}$ offsets have been skipped. The last signature block is, on average, half full. We have found that bitwise shifts are faster than a lookup table for processing of signature blocks.

The speed and space requirements are also shown in Figure 1. Not surprisingly, the signature block scheme requires more space than the previous variable-byte schemes. This space requirement is further increased if byte alignment of blocks is enforced. In terms of speed, the third and fourth bars in the right-hand histogram show that signature blocks are slower than the original variable-byte schemes when offsets are processed in the GolD-GamF-VbyO index. These results are not surprising: signature blocks are slow to process when they are unaligned, and the byte-aligned version is slow because processing costs are no less than the original variable-byte schemes and longer disk reads are required.

As shown by the seventh bar, when offsets are skipped the unaligned signature block scheme is slower than the original

variable-byte scheme. The savings of skipping with signature blocks are negated by more complex processing when blocks are not byte-aligned. In contrast, the right-most bar shows that the byte-aligned signature block scheme with skipping is slightly faster on average than all other schemes. However, we conclude—given the compactness of the index and good overall performance—that the best all-round scheme is the original variable-byte scheme with byte alignment. Therefore, all variable-byte results reported in the Section 4 use the original byte-aligned variable-byte scheme with scanning.

Customised Compression

Combinations of bitwise and bytewise compression schemes are also possible. The aim of such approaches is to combine the fast decoding of bytewise schemes with the compact storage of bitwise schemes. For example, a simple and efficient custom scheme is to store a single bit that indicates which of two compression schemes is used, and then to store the integer using the designated compression scheme. We have experimented with several approaches for storing offsets. The simplest and most efficient approach we tested is as follows: when $f_{d,t} = 1$, we store a single bit indicating whether the following offset is stored as a bitwise Elias delta code or as a bytewise eight-bit binary representation. When storing values, we use Elias delta coding if the value is greater than 256 and the binary scheme otherwise. This scheme has the potential to reduce space because in the median posting $f_{d,t}$ is 1 and the average offset is around 200. Selective use of a fixed-width representation can save storage of the 6-bit prefix used to indicate magnitude in the corresponding delta code.

We report the results with this scheme, which we call *custom*, in the next section. This was the fastest custom scheme we tested. Other approaches we tried included switching between variable-byte and bitwise schemes, using the custom scheme when $f_{d,t}$ is either 1 or 2, and other simple variations. We omit results for these less successful approaches.

4. RESULTS

All experiments described in this paper are carried out on an Intel Pentium III based machine with 512 Mb of main-memory running the Linux operating system. Other processes and disk activity was minimised during timing experiments, that is, the machine was under light-load.

A theme throughout these experiments and greatly impacting on the results is the importance of caching. On a modern machine, caching takes place at two levels. One level is the caching of recently-accessed disk blocks in memory, a process that is managed by the operating system. When the size of the index significantly exceeds memory capacity, to make space to fetch a new inverted list, the blocks containing material that has not been accessed for a while must be discarded. One of the main benefits of compression is that a much greater volume of index information can be cached in memory. For this reason, we test our compression schemes with streams of 10,000 or 25,000 queries extracted from a query log [11], where the frequency distribution of query terms leads to beneficial use of caching. Again, queries are processed as conjunctive Boolean queries.

The other level at which caching takes place is the retention in the CPU cache of small blocks of data, typically of 128 bytes, recently accessed from memory. CPU caching

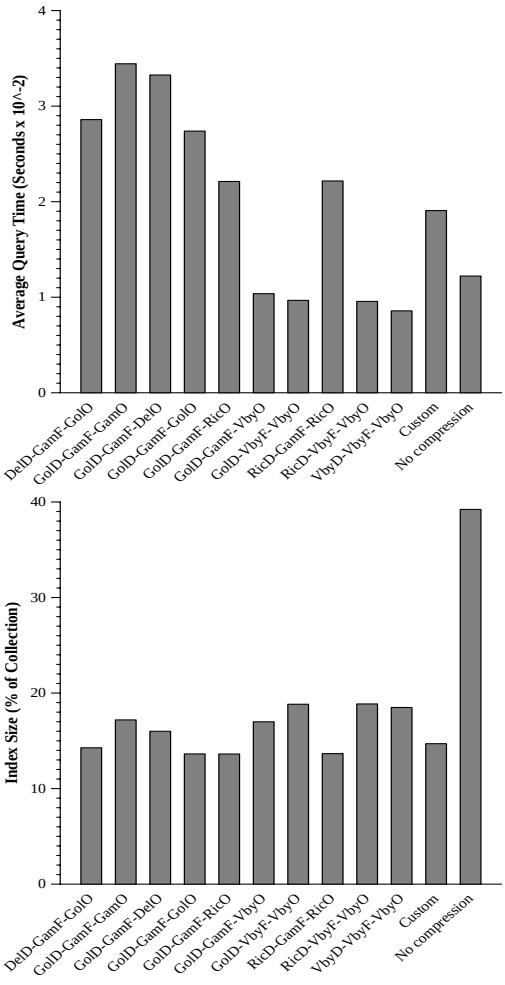

Figure 2: Performance of integer compression schemes for offsets in inverted lists, in an index with Golomb document numbers and gamma frequencies. In this experiment, the index fits in main memory. A 500 Mb collection is used, and results are averaged over 10,000 queries.

is managed in hardware. In current desktop computers, as many as 150 instruction cycles are required to fetch a single machine-word into the CPU. At a coarser level, compression of postings lists means that the number of fetches from memory to cache during decompression is halved.

Small collection

Figure 2 shows the relative performance of the integer compression schemes we have described for storing offsets, on a 500 Mb collection of 94,802 Web documents drawn from the TREC Web track data [5]; timing results are averaged over 10,000 queries drawn from an Excite search engine query log [11]. The index contains 703, 518 terms.

These results show the effect of varying the coding scheme used for document numbers d, frequencies $f_{d,t}$, and offsets o. In all cases where both bitwise and variable-byte codes are used, the bitwise codes are padded to a byte boundary before a variable-byte code is emitted; thus, for example, in a GolD-GamF-VbyO index, there is padding between the gamma

frequency and the sequence of variable-byte offsets. Not all code combinations are shown; for example, given that the speed advantage of using variable-byte document numbers is small, we have not reported results for index types such as VbyD-GamF-RicD, and due to the use of padding a choice such as VbyD-GamF-VbyD. Given the highly skew distribution of $f_{d,t}$ values, Golomb or Rice are not suitable coding methods, so these have not been tried.

In the "no compression" case, fixed-width fields are used to store postings. Document numbers are stored in 32 bits, frequencies in 16 bits, and offsets in 24 bits; these were the smallest multiples of bytes that would not overflow for reasonable assumptions about data properties.

The relative performance of Elias delta and gamma, Rice, and Golomb coding is as expected. The non-parameterised Elias coding schemes result in larger indexes than the paramterised Golomb-Rice schemes that, in turn, result in slower query evaluation. The average difference between offsets is greater than 15, making Elias delta coding more appropriate overall than gamma coding; the latter is both slower and less space-efficient.

On the lower graph in Figure 2, comparing the fourth and fifth columns and comparing the fifth and eighth columns, it can be seen that choice of Golomb or Rice codes for either offsets or document numbers has virtually no impact on index size. Comparing the fifth and eighth columns on the upper graph, the schemes yield similar decoding times for document numbers. However, Rice codes are markedly faster for decoding offsets, because no toggle point calculation is required. Among the bitwise schemes, we conclude that Rice coding should be used in preference to other schemes for coding document numbers and offsets.

The most surprising result is the effect of using the optimised byte-boundary variable-byte scheme for coding offsets. Despite the variable-byte index being 26% larger than the corresponding Rice-coded index, the overall query evaluation time is 62% less. Further speed gains are given by coding all values in variable-byte codes. Indeed, variable-byte decoding is faster even than processing uncompressed lists. This result is remarkable: the cost of transfering variable-byte coded lists from memory to the CPU cache and then decoding the lists is less than the cost of transferring uncompressed lists. To our knowledge, this is the first practical illustration that compression improves the efficiency of an in-memory retrieval system. We conclude from this that variable-byte coding should be used to store offsets to reduce both disk retrieval and memory retrieval costs.

In experiments with integers, Williams and Zobel found that variable-byte coding is faster than the bitwise schemes for storing large integers of the magnitude stored in inverted lists [13]. Our result confirms this observation for retrieval systems, while also showing that the effect extends to fast retrieval from memory and that improvements to variable-byte coding can considerably increase decoding speed.

The custom scheme uses both Elias delta and a binary bytewise scheme, reducing query evaluation to around 58% of the time for the Elias delta scheme. However, the custom scheme is almost twice as slow as the variable-byte scheme and, therefore, has little benefit in practice.

Large collection

Figure 3 shows the results of a larger experiment with an index that does not fit within the main-memory of our machine. Exactly the same index types are tried as for the experiment above. A 20 Gb collection of 4,014,894 Web documents drawn from the TREC Web track data [5] is used and timing results are averaged over 25,000 Boolean queries drawn from an Excite search engine query log [11]. The index contains 9,574,703 terms. We include only selected schemes in our results.

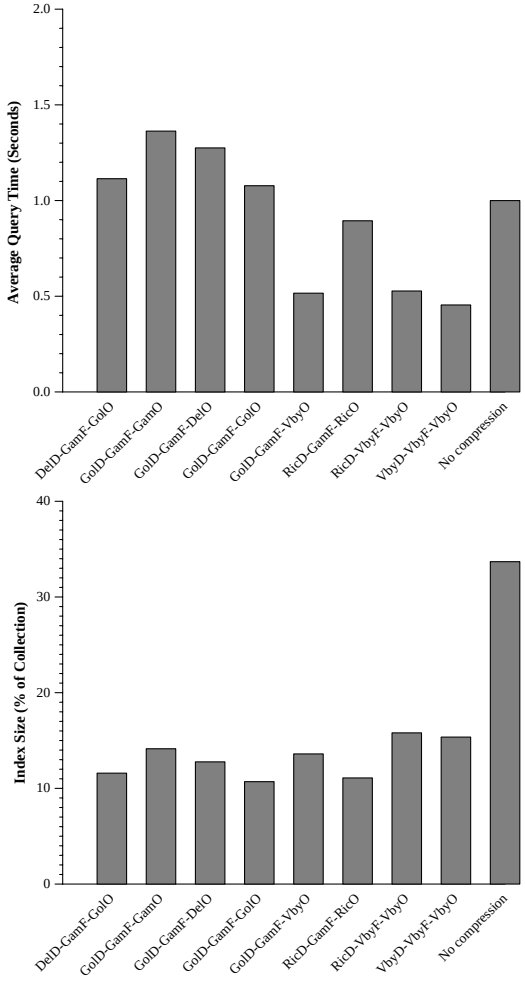

Figure 3: The performance of integer compression schemes for compressing offsets in inverted lists, with Golomb-coded document numbers and gamma-coded offsets. In this experiment, the index is several times larger than main memory. A 20 Gb collection is used, and results are averaged over 25,000 queries.

We again note that we have not used heuristics to reduce query evaluation costs such as frequency-ordering or early termination. Indeed, we have not even used stopping; with stopwords removed, query times are greatly impoved. Our aim in this research is to measure the impact on index decoding time of different choices of compression method, not to establish new benchmarks for query evaluation time. Our improvements to compression techniques could, however, be used in conjunction with the other heuristics, in all likelihood further reducing query evaluation time compared to the best times reported previously.

The relative speeds of the bitwise Golomb, Elias delta, and variable-byte coded offset schemes are similar to that of our experiments with the 500 Mb collection. Again, variable-byte coding results in the fastest query evaluation. Perhaps unsurprisingly given the results described above, an uncompressed index that does not fit in main-memory is relatively much slower than the variable-byte scheme; the disk transfer costs are a larger fraction of the overall query cost when the index does not fit in memory, and less use can be made of the memory cache. Indexes with variable-byte offsets are twice as fast as indexes with Golomb, delta, or gamma offsets, and one-and-a-half times as fast as indexes with Rice offsets. VbyD-VbyF-VbyO indexes are twice as fast as any index type with non-variable-byte offsets.

In separate experiments we have observed that the gains demonstrated by compression continue to increase with collection size, as the proportion of the index that can be held in memory declines. Despite the loss in compression with variable-byte coding, indexes are still less than one-seventh of the size of the indexed data, and the efficiency gains are huge.

5. CONCLUSIONS

Compression of inverted lists can significantly improve the performance of retrieval systems. We have shown that an efficiently implemented variable-byte bytewise scheme results in query evaluation that is twice as fast as more compact bitwise schemes. Moreover, we have demonstrated that the cost of transferring data from memory to the CPU cache can also be reduced by compression: when an index fits in main memory, the transfer of compressed data from memory to the cache and subsequent decoding is less than that of transferring uncompressed data. Using byte-aligned coding, we have shown that queries can be run more than twice as fast as with bitwise codes, at a small loss of compression efficiency. These are dramatic gains.

Modern computer architectures create opportunities for compression to yield performance advantages. Once, the main benefits of compression were to save scarce disk space and computer-to-computer transmission costs. An equally important benefit now is to make use of the fact that the CPU is largely idle. Fetching a single byte from memory involves a delay of 12 to 150 CPU cycles; a fetch from disk involves a delay of 10,000,000 cycles. Compression can greatly reduce the number of such accesses, while CPU time that would otherwise be unused can be spent on decoding. With fast decoding, overall costs are much reduced, greatly increasing query evaluation speed. In current computers such architecture considerations are increasingly important to development of new algorithms for query processing. Poor caching has been a crucial shortcoming of existing algorithms investigated in this research.

There are several possible extensions to this work. We plan to investigate nibble-coding, a variant of variable-byte coding where two flag bits are used in each variable-byte block. It is likely that this approach may improve the performance of signature blocks. We will also experiment with phrase querying in practice and to explore the average query evaluation speed when partial scanning is possible.

6. REFERENCES

[1] V. Anh, O. de Kretser, and A. Moffat. Vector-Space ranking with effective early termination. In W. Croft, D. Harper, D. Kraft, and J. Zobel, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 35–42, New York, Sept. 2001.

[2] E. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[3] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, Mar. 1975.

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

[5] D. Hawking, N. Creswell, and P. Thistlewaite. Overview of TREC-7 very large collection track. In E. Voorhees and D. Harman, editors, *Proc. Text Retrieval Conference (TREC)*, pages 91–104, Washington, 1999. National Institute of Standards and Technology Special Publication 500-242.

[6] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, Oct. 1996.

[7] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory-efficient ranking. *Information Processing & Management*, 30(6):733–744, 1994.

[8] G. Navarro, E. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[9] M. Persin. Document filtering for fast ranking. In W. Croft and C. van Rijsbergen, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 339–348, Dublin, Ireland, 1994.

[10] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[11] A. Spink, D. Wolfram, B. J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science*, 52(3):226–234, 2001.

[12] A. Vo and A. Moffat. Compressed inverted files with reduced decoding overheads. In R. Wilkinson, B. Croft, K. van Rijsbergen, A. Moffat, and J. Zobel, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 290–297, Melbourne, Australia, July 1998.

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

[14] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[15] N. Ziviani, E. de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, Nov. 2000.