Information Retrieval, 6, 5–19, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Compressing Inverted Files

ANDREW TROTMAN*
Department of Computer Science, University of Otago, PO Box 56, Dunedin, New Zealand

Received April 17, 2002; Revised October 2, 2002; Accepted October 16, 2002

Abstract. Research into inverted file compression has focused on compression ratio—how small the indexes can be. Compression ratio is important for fast interactive searching. It is taken as read, the smaller the index, the faster the search.

The premise "smaller is better" may not be true. To truly build faster indexes it is often necessary to forfeit compression. For inverted lists consisting of only 128 occurrences compression may only add overhead. Perhaps the inverted list could be stored in 128 bytes in place of 128 words, but it must still be stored on disk. If the minimum disk sector read size is 512 bytes and the word size is 4 bytes, then both the compressed and raw postings would require one disk seek and one disk sector read. A less efficient compression technique may increase the file size, but decrease load/decompress time, thereby increasing throughput.

Examined here are five compression techniques, Golomb, Elias gamma, Elias delta, Variable Byte Encoding and Binary Interpolative Coding. The effect on file size, file seek time, and file read time are all measured as is decompression time. A quantitative measure of throughput is developed and the performance of each method is determined.

Keywords: index compression, inverted files, document indexing, text searching

Introduction

An inverted file index of a document collection consists of two parts, the dictionary file and the postings file. For each unique term in the corpus, the dictionary stores that term, a position into, and the length of the inverted list in the postings file. Occasionally other data is included. The inverted list for any given term is a vector (document-number, occurrences) for each document in which the term is present.

The inverted list for a given term is often represented as $\{\langle d_1, f_1\rangle, \langle d_2, f_2\rangle, \ldots, \langle d_n, f_n\rangle\}$ where a pair $\langle d, f\rangle$ means the given term is present f times in document d. The inverted list is considered sorted sequentially by document ordinal number—something that occurs as a consequence of indexing sequentially and assigning each document a unique number.

It is common to store these lists as two, and that is the representation used herein. One list, the inverted document list, is the monotonic increasing sequence of document numbers. The other list, the inverted frequency list, is the list of frequencies for each document.

An additional list, the inverted word occurrence list, is required for adjacency searching. The vector $\{w_1, w_2, \ldots, w_n\}$ is stored where w is the ordinal number of the given instance of the given term within the document collection.

*Work conducted while at National Center for Biotechnology Information, Bethesda, MD, USA.

When the dictionary is stored as a B-tree with a high branching factor, it takes one disk seek and one disk read to retrieve the position/length pair for any given term (Witten et al. 1994). To retrieve an inverted list takes one more disk seek and one more disk read (Witten et al. 1994).

Zobel and Moffat (1995) and Williams and Zobel (1999) suggest that compression of inverted files will result in an increase in query throughput. The time to load and decompress a compressed inverted list is shorter than the time to load a never compressed inverted list, and therefore throughput is increased.

Many techniques have been proposed for compressing inverted lists (Williams and Zobel 1999, Moffat and Stuiver 2000, Antoshenkov 1994, Bookstein et al. 2000); the primary focus being on compression ratio. Compression ratio is measured as the mean number of bits required for storing the *d* pointers in the inverted lists of an entire document collection. The time to load and decompress are often forgotten or not reported.

Compression techniques tested

The inverted document and inverted word occurrence lists are strictly increasing monotonic sequences. Compressing these sequences is not as efficient as compressing the sequence of successive differences (Williams and Zobel 1999). For the sequence $\{4, 7, 9, 11, 15\}$, the differences are $\{4, 3, 2, 2, 4\}$. Frequent terms give rise to longer lists, which necessitates smaller differences, and therefore more efficient compression. For a term that occurs in every document the sequence would be $\{1, 1, 1, \ldots\}$ which can be compressed very efficiently. These difference lists can then be compressed using a suitable compression scheme.

Variable Byte Coding. Modern computers are designed for fast manipulation of bytes and sequences of bytes. A byte aligned encoding of integers can take advantage of optimized hardware operations when decompressing. Taking advantage of the hardware does, however, necessitate a trade off. It is necessary to forfeit size to take advantage of speed.

An integer x is represented as a sequence of 7-bit bytes; each being encoded in 8-bit bytes with the high bit a 1 for the last byte of the sequence and a 0 for all other bytes. The integer 135, for example, is converted into the sequence 00000001, 10000111.

Both compression and decompression using Variable Byte Coding are very fast as the smallest unit of manipulation is the whole byte. The compression ratio for an array of 32-bit integers is at best 25%, and at worst 125%. As integers become large the coded space also becomes large.

Elias gamma coding. Inverted lists can become very long, and information retrieval is disk bound (Williams and Zobel 1999, Zobel and Moffat 1995). Retrieving a long inverted list from disk can require more time than further manipulation of the very same list. To increase throughput from disk, a more efficient encoding of integers is required.

In the gamma code (Elias 1975) an integer x is represented in two parts, a header with $\lfloor \log_2 x \rfloor$ zeros, followed by the tail, a binary representation of x. This way the integer 9 is represented by the sequence 000, 1001 since $\log_2 x = 3$ the header is 000, and the tail is 1001.

Gamma coding is efficient for small integers where the length of the header remains small. When integers become large, the storage space also becomes large.

Elias delta coding. In the delta code (Elias 1975) an integer x is represented in two parts, $1 + \lfloor \log_2 x \rfloor$ using the gamma code, followed by the binary representation of x with the high bit turned off. This way the integer 9 is represented by the sequence 00, 100, 001.

The delta code requires more bits to store smaller integers, but large integers are more efficiently compressed.

Choice of gamma coding over delta coding would be made when, in a sequence of integers, the majority of the integers are known to be small. For information retrieval, as the number of occurrences of a term increases, so to does the density in the inverted lists. Examining just the inverted document list, as the density increases, the successive differences tend to the sequence $\{1, 1, 1, \ldots\}$, which has a more efficient coding using gamma codes.

Golomb coding. In the Golomb code (1966) an integer x is represented as two parts, a quotient and a remainder. The quotient is calculated as $q = \lfloor (x - 1)/k \rfloor$, and the remainder is calculated as $r = x - (q * k) - 1$ where k is the base to which the Golomb code is calculated.

If $r < p$ it can be stored in $\lfloor \log_2 k \rfloor$ bits, else it requires $\lceil \log_2 k \rceil$ bits, where p is the pivot point and is given by $p = 2^{\lfloor \log_2 k \rfloor + 1} - k$.

When $r < p$ the Golomb code is constructed with q zeros, a one, and r in binary. Otherwise it is represented with q zeros, a one, and $r + p$ in binary. This way the integer 9 encoded with $k = 3$ is represented as 00,1,11.

Choice of k is vital to the scheme. With a bad choice, encoded integers can become very large and thus take a long time to decompress. Witten et al. (1994) suggest assuming integers in an inverted list follow a Bernoulli model and using $k \approx 0.69 \times \text{mean}(a)$ for an array of integers, a.

The value of k chosen for these experiments was given using Witten's approximation. Each inverted list was compressed with its own k. The cost of storing k is not calculated and is not included in these results.

Williams and Zobel (1999) describe implementation optimizations for Golomb coding and suggest that in general Golomb encoding of integers is both faster to decode and more space-efficient than either Elias gamma or Elias delta codes, a finding confirmed by this investigation.

Binary Interpolative Coding. Binary Interpolative Coding (Moffat and Stuiver 1996, 2000) encodes a monotonic increasing sequence of integers using knowledge of its neighbors.

If, in a sequence of integers X_1, for a given integer x_i, the preceding integer x_{i-1} and following integer x_{i+1} are known, then through subtraction it is possible to know the maximum number of bits necessary to store x_i. Because x_i must be in the range $(x_{i-1} + 1, x_{i+1} - 1)$, the maximum number of bits required is $\log_2(x_{i+1} - x_{i-1} - 2)$. To decode requires prior knowledge of x_{i-1} and x_{i+1} so a list X_2 is drawn from the original list X_1 such that every second integer from list X_1 is in X_2 and the coding is then applied recursively.

The implementation tested here uses a centered minimal binary code (Howard and Vitter 1993). When compressing a sequence $1..n$, $\lceil \log_2 n \rceil$ bits are required, however $2^{\lceil \log_2 n \rceil} - n$ encodings are wasted. These wasted encodings can be shortened by one bit and used (so long as no short encoding is a prefix of a longer encoding). These minimal binary codes are then centered on the encoding range. Numbers at the extremes of the range requiring one bit more for storeage than those in the center.

As a further refinement, inverted centered minimal binary codes can be used. As the size of a range tends to small, the density is increasing. Moffat and Stuvier (2000) suggest taking advantage of this by allocating the minimal binary codes to the extremes in the range, rather then to the center. This refinement has not been examined.

As suggested by Moffat and Stuvier (2000), the tested implementation is recursive and minimizes the number of non $2^k - 1$ recursive calls by dividing the original list unevenly.

Compression of inverted document and inverted word occurrence lists is performed directly on the inverted lists, and without taking the differences. A monotonic increasing sequence is generated from the frequencies by calculating the cumulative sum at each document.

Compression techniques not tested

Chunking. Moffat and Zobel (1996) and later Vo and Moffat (1998) suggest chunking the compressed data into a series of blocks. For each block, a value/offset pair is stored. The value is the first value in the chunk, the offset is a pointer to that chunk of the compressed sequence. The value/offset pairs are stored at the beginning of the compressed sequence. If only part of the original sequence is needed, the appropriate chunk can be located and decompressed without decompressing the entire sequence.

Novel techniques. A number of novel index compression techniques have been proposed. In some instances decompression is not required, in others the compression is lossy, in others the technique does not scale.

Statistical compression techniques appear to produce good compression ratios, however there is still much undone research in the area. Bookstein et al. (1994) suggested a Markov approach to which they subsequently added by a Bayesian approach (Bookstein et al. 2000).

Numerous lossy techniques have been suggested. One commonly used technique is to drop frequently occurring words from the indexes (giving 100% compression for said terms). Koudas (2000) suggests merging into a single index terms that occur infrequently in the corpus, and also infrequently in queries. These are then compressed using a lossy compression technique to attain maximum compression.

The Byte-aligned Bitmap Codes (BBC) (Antoshenkov 1994) have received much interest in the relational database community (Stockinger 2001, Johnson 1999, Chan and Ioannidis 1999). A one-dimensional bitmap is divided on byte boundaries and encoded in chunks. Each chunk consists of two parts, a run and a tail. A run is series of bytes whose value is either all bit-zeros or all bit-ones. The tail is a series of bytes that breaks this rule. Chunks are written out with a header describing the chunk, then the run, and then the tail. Boolean operations can be performed directly on BBC codes without decompression and for operations on sparse bitmaps perform very well (Johnson 1999).

Using Hierarchical Bit-Vector Compression (Choueka et al. 1986), a bitmap is divided into a series of equal sized partitions, and a partition index created. The index is itself a bitmap, of length "number of partitions", where a one represents "set bits in partition" and a zero represents no set bits in partition. The compressed bitmap is then constructed by dropping the "all zero" partitions and appending the remaining partitions to the index. The process can then be re-applied recursively to form a tree. Again, decompression is not required for index manipulation.

Compression techniques using the inverted file dictionary as the compression dictionary have been proposed (Varadarajan and Chiuen 1997, Navarro et al. 2000). These techniques typically divide the text into a number of fixed-size blocks and index the blocks. If the block size is larger then the document size, there are fewer blocks to index than documents and therefore the index will be smaller. Once a hit has been identified the required block is retrieved and a linear search is performed to find the given document. Decompressing the block is unnecessary because the inverted file dictionary and the compression dictionary are shared allowing the compressed symbols to be scanned directly. If the block size is exactly document size, this technique degenerates to a document based inverted file search followed by a document linear search. If the block size is sub-document size the complexity of the block-based index search is greater than the complexity of a document-based search.

A Binary Decision Diagram Encoding of an inverted index (Lai and Chen 2001) represents the list as a binary tree of depth "bits in integer". Each integer is represented as a walk through the tree much as terms in a dictionary can be represented as a walk through a trie. Boolean operations can be performed directly on the tree. To determine, for example, if a given integer lies in the list, the tree is simply walked top-down.

Methods

Testing compression

Five compression techniques were examined, Variable Byte Compression, Elias gamma (1975), Elias delta (1975), Golomb (1966), and Binary Interpolative Coding (Moffat and Stuiver 1996), for suitability in compressing the TREC Wall Street Journal collection (Harman 1992–96). WSJ was chosen because of its ubiquity.

An inverted file index of WSJ was created and used to test each algorithm. Three inverted lists were created for each term, an inverted document list, and inverted frequency list, and an inverted word occurrence list. Each inverted list was read in turn and compressed to determine the number of bits required for storage. The compressed sequence was then decompressed and the decompression time was recorded. For each technique, against each inverted list, the triple is written to disk (integers in the sequence, compressed size in bits, decompression time).

The mean compression ratio was measured in bits per integer. Implementation used 32 bit integers, as that is the word size of the Pentium CPU.

Mean time to decompress was measured in clock cycles per integer using the Pentium RDTSC instruction (Intel Corporation 1997) (this instruction measures elapsed clock cycles). Experiments were carried out on a Windows 2000 Dell OptiPlex Gx 200, with a Pentium III running at 927 MHz.

Although each algorithm used here was implemented independently, Williams and Zobel have made the source code for Elias gamma, Elias delta, and Golomb encoding available for download (Williams 2002) on their web site.

Testing disk

To load an inverted list requires three operations, a disk seek, a disk read and decompression—therefore disk performance was also measured. Opening the computer revealed an IBM Deskstar DTLA-307015 ATA/IDE drive—this is a 15 GB drive (IBM Corporation 2000).

The mean disk seek time and mean disk read time were independently measured using the RDTSC instruction under Windows 2000. Disk throughput is measured in clock cycles to maintain a single unit of measure for all timings.

To determine how seek distance effects seek time a large file was created then defragmented. Seeks over random distances were timed then graphed. To determine the disk read time, reads of random numbers of sectors were performed from random locations in the file.

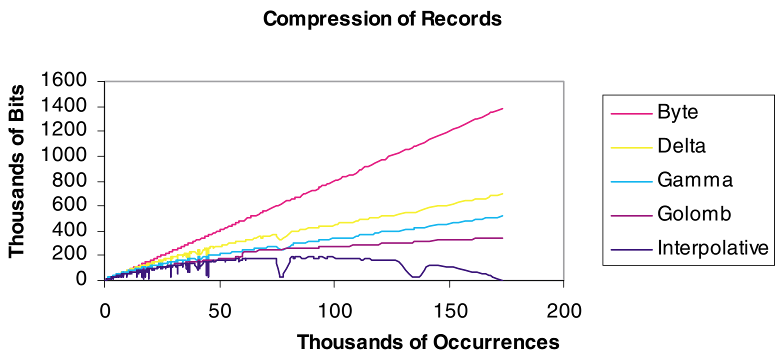
Once seek, read, and decompress times are known, it becomes possible to model each compression method's throughput. Graphing the performance against inverted list length resulted in the heuristics needed to choose a suitable algorithm.

Results

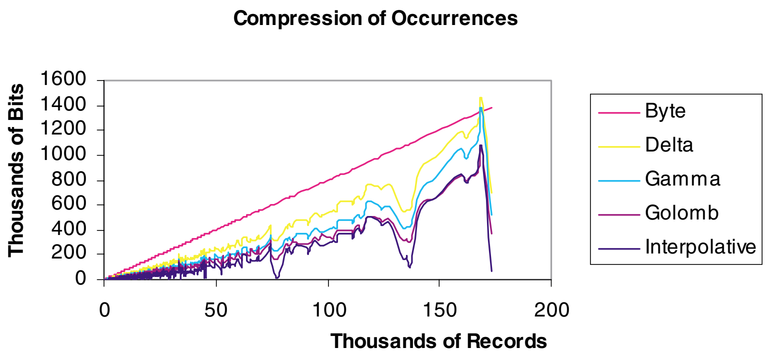
Compression ratio

Figure 1 displays the number of bits needed to store the compressed inverted list against the length of the inverted list (measured in integers). As reported elsewhere (Moffat and Stuiver 2000), compression using Binary Interpolative Coding results in the highest compression ratio for the inverted document list and inverted frequency list. For the inverted word occurrence list, Golomb slightly outperforms Binary Interpolative Coding.

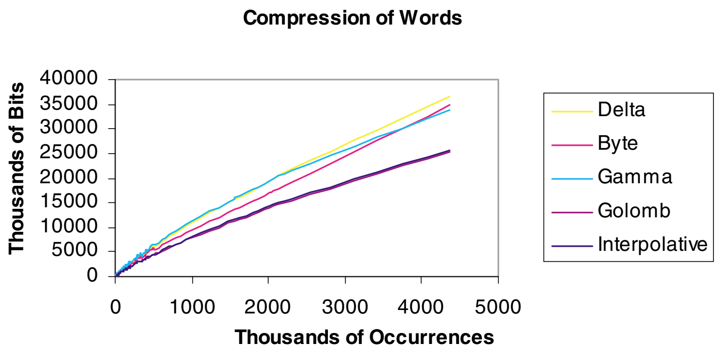
Binary Interpolative Coding performs very well when the data is clustered. If a term lies in a long sequence of adjacent documents, the compression becomes trivial and fewer bytes are required. As the frequency of the term increases, the number of documents in which it lies increases and longer sequences of adjacency are expected—so the compression will be more effective. When a term lies in all documents, the resulting compression can be stored in zero bytes! In figure 1(a) the graph for Binary Interpolative Coding demonstrates this clustering effect. As the frequency tends from 0.5 to 1.0, the compressed strings tend to zero length.



(a)



(b)

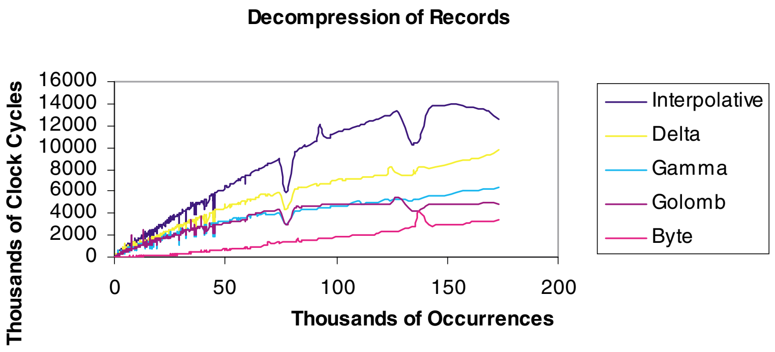


(c)

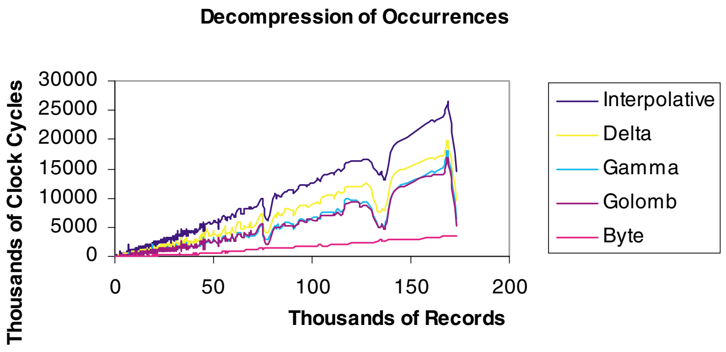
Figure 1. (a) Compression of inverted document lists, (b) Compression of inverted frequency lists, and (c) Compression of inverted word occurrence lists.

Decompression speeds

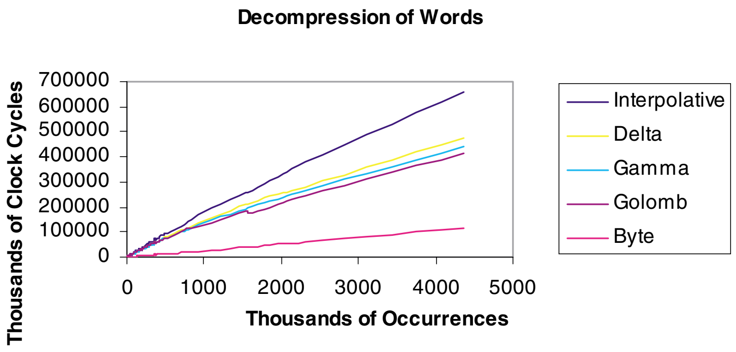
Figure 2 graphs decompression time in clock-cycles against the length of the uncompressed string (measured in integers). In all cases Variable Byte Coding outperforms the others, this



(a)



(b)



(c)

Figure 2. (a) Decompression of inverted document lists, (b) Decompression of inverted frequency lists, and (c) Decompression of inverted word occurrence lists.

is almost certainly due to the algorithm being byte oriented whereas the other algorithms are bit oriented. Decoding bytes into bits for Elias or Golomb decoding takes considerable time. From figure 1 and figure 2, it is clear Golomb outperforms Elias gamma and Elias delta in both compression size and decompression speed as previously reported by Williams and Zobel (1999).

Even though Binary Interpolative Coding is most efficient at compression, it is most inefficient at decompression. This is almost certainly due to the recursive nature of the algorithm—the other algorithms are implemented iteratively.

The disk

To measure the throughput of each compression method, it is necessary to measure the throughput of the disk. Both read time and seek time were measured.

How slow is the disk? There are two operations necessary to transfer data from the disk to memory. The first involves moving the disk head to the correct location (the seek). The second involves reading a sequence of sectors from disk and transferring to memory (the read).

Hard Drive manufacturers often provide exact details of the operating specifications of their equipment (IBM Corporation 2000), however, these details often do not take into account the environment in which the equipment is running. The seek and read times reported here are for an IBM Deskstar DTLA-307015 ATA/IDE drive running under Windows 2000.

How slow is a seek? To determine how disk seek time behaves, a 1.4 GB file was created, defragmented, and opened using non-buffered I/O.[1] 1024 sample seek points were chosen within the file. Each sample point was chosen to be a random location within the file and to lie on an exact sector boundary.

The time taken to seek from the current location to the sample point, and to read one sector, was taken using the Pentium RDTSC instruction. The results were collated in memory and written to disk at conclusion of the experiment.

Figure 3 graphs seek distance against time. Analysis of the samples suggests seek time increases with distance, and is highly variable. This is partly expected. The seek time should increase with distance as the head must seek further across the disk.

Seek time should be variable. Adjacent sectors in the file might not be adjacent on the disk. A seek forward by one sector could result in no disk head movement (if both sectors lie in the same track) or it could result in a head move (if they are not). Once the head arrives at the required location, some portion of a whole disk revolution is necessary before the read can occur. It is unlikely the head will be positioned above the correct sector at the exact moment it arrives at the desired track.

The data collected has a mean seek time of 7,954,054 clock cycles or 8.6 ms. This is inline with the manufacturer's specifications of 8.5 ms. The measured seek times are expected to take longer than the manufacturer's due to operating system overheads.

Given the mean seek time within the sample file, it is possible to calculate the mean seek time within a file of a different size. The general equation of the line that fits the samples in

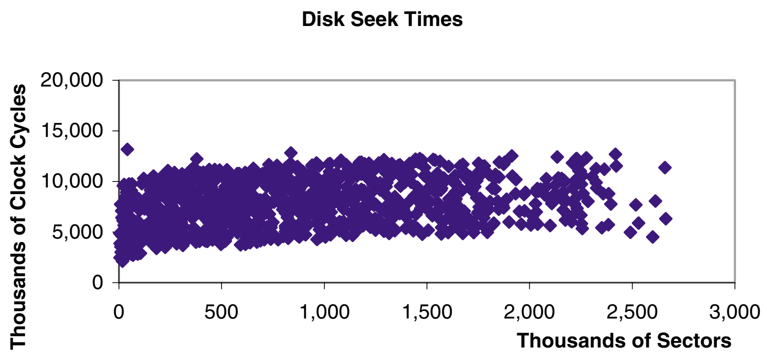


Figure 3. Time to seek and read as distance increases.

figure 3 is

$$T_1 = 1.02B + C \tag{1}$$

where T_1 is time in clock cycles and B is the number of sectors to seek. C is the constant 7,039,007, the inherent cost of seeking measured in clock cycles. The ratio of the size difference between the two files is

$$R = \frac{S_2}{S_1} \tag{2}$$

where S_1 is the size of the sampled file, and S_2 is the size of the new file.

The calculated new mean seek time is

$$T_2 = (T_1 - C)R + C \tag{3}$$

where T_2 is the calculated mean for the new file, and T_1 is the observed mean of the sampled file.

How slow is a read? To determine how disk read time behaves, a 1.4 GB file was created, defragmented, and opened using non-buffered I/O. 1024 samples were chosen to read a random number of sectors (up to 32 sectors) from a random sector-aligned location within the file. Times were measured in clock cycles using the Pentium RDTSC instruction. Results were collated in memory and written to disk once the experiment had concluded.

The read time in clock cycles per sector is graphed against the number of sectors read in figure 4. The results are as expected. When a small number of sectors is read the per-sector impact of seeking is high. As the length of the read increases the seek impact becomes increasingly insignificant. The graph tends to a mean read time of 18,327 cycles per sector, a data transfer rate of 24.69 MB/s. This is inline with the manufacturer's specifications that report a sustained data transfer rate of 37 MB/s. The observed data transfer rate is believed to be low due to operating system overheads.

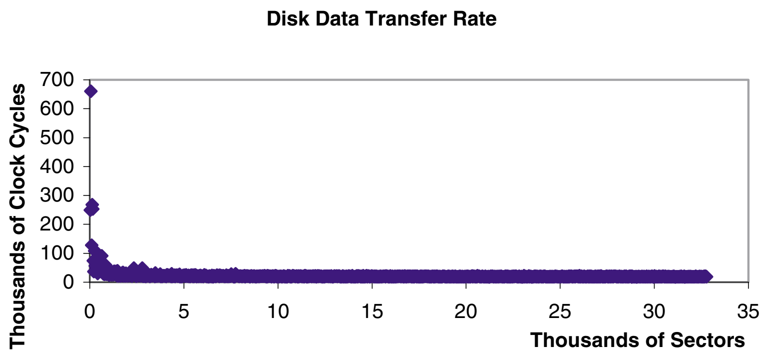


Figure 4. Time to read one sector from disk as read size increases.

For n integers the time t required to read a compressed inverted list from disk (post seek) is

$$t = \left\lceil \frac{En}{512 \times 8} \right\rceil \times 18327 \tag{4}$$

where E is the effectiveness of the compression technique measured in bits per integer. As the number of integers being read increases the effect of the ceiling diminishes so

$$t \approx EnA \tag{5}$$

where

$$A = \frac{18327}{512 \times 8} \approx 4.47 \tag{6}$$

The disk. Equation (3) describes the cost of randomly seeking into a file of any size. Equation (4) describes the cost of reading integers from a file. The disk's observed characteristics are inline with the manufacturer's specification, but in both cases are slower. The mean disk seek takes about 400 times the mean sector read time, suggesting the bottleneck in searching is not the read, but the seek.

When the same tests were carried out on small defragmented files the results were inline with those reported here. Seek times were in the range 2 million–10 million clock cycles and read times in the order of 18,500 clock cycles per block.

Putting it all together

Results from the disk experiments are combined with results from the compression experiments to give a true picture of how each algorithm performs.

Assuming a 1.4 GB file of inverted lists compressed with the most efficient scheme (from Table 1), the total cost of the load and decompressing is given by

$$U \approx T + EnA + dn \tag{7}$$

Table 1. Compression effectiveness, decompression speed, and relative performance.

	Byte	Delta	Gamma	Golomb	Interpolative	Unit
Mean						
Bits						
Records	9.35	8.67	8.48	6.15	5.94	bpo
Occurrence	8.00	4.52	3.43	2.69	1.57	bpr
Words	13.99	15.15	17.98	12.31	12.11	bpo
Cycles						
Records	11.47	112.55	101.09	99.09	149.62	cpo
Occurrences	9.24	64.52	44.91	42.52	105.71	cpr
Words	21.85	170.48	181.99	188.62	209.78	cpo
Relative mean						
Bits						
Records	1.57	1.46	1.43	1.04	1.00	
Occurrence	5.08	2.87	2.18	1.71	1.00	
Words	1.16	1.25	1.48	1.02	1.00	
Cycles						
Records	0.08	0.75	0.68	0.66	1.00	
Occurrences	0.09	0.61	0.42	0.40	1.00	
Words	0.10	0.81	0.87	0.90	1.00	

bpo—bits per occurrence, bpr—bits per record, cpo—cycles per occurrence, cpr—cycles per record.

where U is the time taken, T is the cost of the seek, EnA is the cost of the read, and dn is in-memory cost of decompression and n is the number of compressed integers being read.

As n tends to very large, the cost of the seek becomes insignificant and performance P of the decompression scheme is governed by the read and decompression time

$$P \approx EA + d \tag{8}$$

As n tends to zero, the performance is governed exclusively by the seek

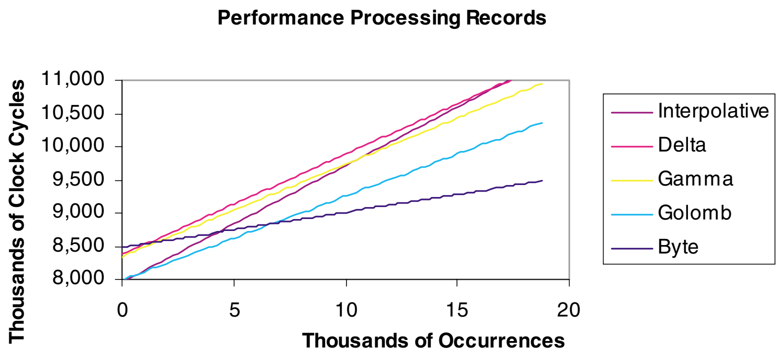
$$P \approx T \tag{9}$$

from (3)

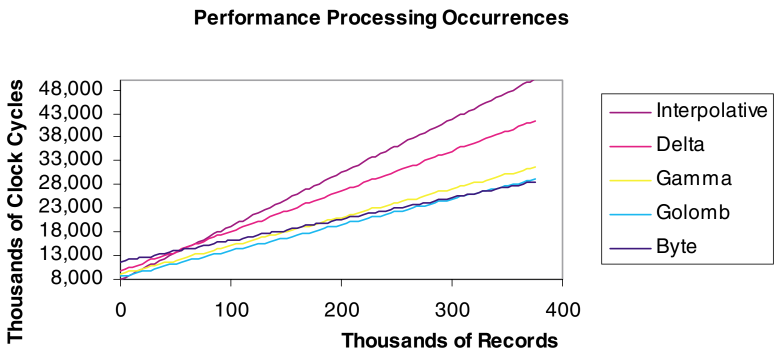
$$P \approx T = (M - C)R + C \tag{10}$$

where M is the measured mean seek time for the disk.

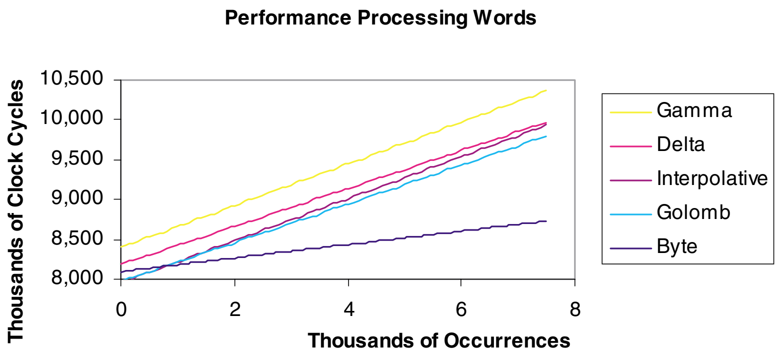
From figure 5, Binary Interpolative Coding performs best when the number of integers being coded is very small, however, very quickly Golomb coding outperforms the others, until finally when the number of occurrences becomes large Variable Byte Coding is most efficient.



(a)



(b)



(c)

Figure 5. (a) Projected performance loading and decompressing inverted document lists, (b) Projected performance loading and decompressing inverted frequency lists, and (c) Projected performance loading and decompressing inverted word occurrence lists.

These calculations are based on applying the same compression technique to every inverted list in an inverted index of a document corpus.

Conclusions

Mean compression ratio over a set of inverted indexes has been the exclusive measure of goodness of fit. Little or no research has been done on how the method performs as the size of the inverted lists increase. Since processing long inverted lists is a bottleneck in inverted file information retrieval systems, performance as inverted lists grow is a better measure.

Measured here is the performance under one set of conditions, the IBM Deskstar DTLA-307015 15 GB drive under Windows 2000 and using the TREC Wall St. Journal collection.

When the inverted lists are small, the throughput is determined by disk seek time, which is related to the length of the disk file. Minimizing the file size minimizes the length of the disk seek, so choice of compression based on minimizing file size is best (for example Golomb). As the length of the lists grows, however, the disk seek becomes less significant and throughout is related to compression ratio and decompression speed (so Variable Byte Encoding is better).

If disk space is a premium, Golomb compression is recommended. As reported by Williams and Zobel, use of Golomb results in both smaller files and faster decompression than Elias gamma or Elias delta (Williams and Zobel 1999).

Use of Golomb/gamma (Moffat and Zobel 1996) coding where the document numbers are encoded using Golomb and the frequencies using Elias gamma appears to be less efficient then using Golomb for both positions and frequencies.

Bounds can now be placed on the design of a "better" compression scheme. By knowing the mean compression ratio and the rate of decompression it is possible to determine quantitatively how the scheme will perform relative to existing schemes at various lengths of inverted list. Optimizations of the existing algorithms (perhaps an iterative Binary Interpolative Coding) can be quantitatively evaluated against their original algorithms for a measure of effectiveness of each optimization.

For optimal performance a mix of algorithms is recommended. Keep file size small by forfeiting clock cycles when an inverted list is small. Forfeit disk space when a list is long. Zipf's law suggests most terms occur infrequently so the file size overhead of using Variable Byte Encoding for long inverted lists is likely to be low.

Note

1. Windows 2000 has a flag FILE_FLAG_NO_BUFFERING to the file open method CreateFile( ) that "Instructs the system to open the file with no intermediate buffering or caching" (Microsoft Corporation 2000).

References

Antoshenkov G (1994) Byte aligned data compression. US Patent Number 5363098.

Bookstein A, Klein ST and Raita T (1994) Markov models for clusters in concordance compression. In: Proceedings of the 1994 IEEE Data Compression Conference DCC-94, pp. 116–125.

Bookstein A, Klein ST and Raita T (2000) Simple bayesian model for Bitmap compression. Information Retrieval, 1(4):315–328.

Chan CY and Ioannidis YE (1999) An efficient Bitmap encoding scheme for selection queries. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 215–226.

Choueka Y, Fraenkel AS and Klein ST (1988) Compression of concordances in full-text retrieval systems. In: Proceedings of the 11th ACM-SIGIR Conference on Information Retrieval, pp. 597–612.

Choueka Y, Fraenkel AS, Klein ST and Segal E (1986) Improved hierarchical bit-vector compression in document retrieval systems. In: Proceedings of the 9th ACM-SIGR Conference on Information Retrieval, pp. 88–97.

Elias P (1975) Universal codeword sets and the representation of the integers. IEEE Transactions on Information Theory, 21:194–203.

Golomb SW (1966) Run-length encodings. IEEE Transactions on Information Theory, 12(3):399–401.

Harman DKE (1992–96) Proceedings of the TREC Text Retrieval Conference. National Institute of Standards Special Publication.

Howard P and Vitter J (1993) Fast and efficient lossless image compression. In: Proceedings of the 1993 IEEE Data Compression Conference DCC-93, pp. 351–360.

IBM Corporation (2000) IBM Deskstar 75GXP and Deskstar 40GV hard disk drives. IBM TECHFAX #7011. Available at www.storage.ibm.com/hdd/desk/deskstar75gxp40gv.pdf (Viewed April 2002).

Intel Corporation (1997) Using the RDTSC instruction for performance monitoring. Available at cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf (Viewed April 2002).

Johnson T (1999) Performance measurements of compressed Bitmap indices. In: Proceedings of the 25th VLDB Conference, pp. 278–289.

Klein ST, Bookstein A and Deerwester S (1989) Storing text retrieval systems on CD-ROM: Compression and encryption considerations. ACM Transactions on Information Systems, 7:230–245.

Koudas N (2000) Space efficient Bitmap indexing. In: Proceedings of CIKM 2000, pp. 194–201.

Lai CH and Chen TF (2001) Compressing inverted files in scalable information systems by binary decision diagram encoding. Presented at SC2001, available at http://www.sc2001.org/papers/pap.pap338.pdf (visited April 2002).

Microsoft Corporation (2000) CreateFile. Available at msdn.microsoft.com/library/en-us/fileio/filesio_7wmd.asp (Viewed April 2002).

Moffat A and Stuiver L (1996) Exploiting clustering in inverted file compression. In: Proceedings of the 1996 IEEE Data Compression Conference DCC-96, pp. 82–91.

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

Moffat A and Zobel J (1992) Parameterized compression of sparse Bitmaps. In: Proceedings of the 15th ACM-SIGIR Conference on Information Retrieval, pp. 274–285.

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

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

Stockinger K (2001) Design and implementation of Bitmap indices for scientific data. In: Proceedings of International Data Engineering and Applications Symposium IDEAS-01, pp. 47–57.

Varadarajan S and Chiuen T (1997) SASE: Implementation of a compressed text search engine. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems.

Vo AN and Moffat A (1998) Compressed inverted files with reduced decoding overheads. In: Proceedings of the 21st ACM-SIGIR Conference on Information Retrieval, pp. 290–297.

Williams HE (2002) goanna.cs.rmit.edu.au/∼hugh/software/integer.coding.tar.gz (viewed April 2002).

Williams HE and Zobel J (1999) Compressing integers for fast file access. The Computer Journal, 42(3):193–201.

Witten IH, Moffat A and Bell TC (1994) Managing gigabytes. Van Nostrand Reinhold 1994.

Zobel J and Moffat A (1995) Adding compression to a full-text retrieval system. Software Practice and Experience, 25(8):891–903.