Compressing Integers for Fast File Access

HUGH E. WILLIAMS AND JUSTIN ZOBEL

Department of Computer Science, RMIT University
GPO Box 2476V, Melbourne 3001, Australia
Email: {hugh,jz}@cs.rmit.edu.au

Fast access to files of integers is crucial for the efficient resolution of queries to databases. Integers are the basis of indexes used to resolve queries, for example, in large internet search systems and numeric data forms a large part of most databases. Disk access costs can be reduced by compression, if the cost of retrieving a compressed representation from disk and the CPU cost of decoding such a representation is less than that of retrieving uncompressed data. In this paper we show experimentally that, for large or small collections, storing integers in a compressed format reduces the time required for either sequential stream access or random access. We compare different approaches to compressing integers, including the Elias gamma and delta codes, Golomb coding, and a variable-byte integer scheme. As a conclusion, we recommend that, for fast access to integers, files be stored compressed.

Keywords: Integer compression, variable-bit coding, variable-byte coding, fast file access, scientific and numeric databases.

1. INTRODUCTION

Many data processing applications depend on efficient access to files of integers. For example, integer data is prevalent in scientific and financial databases, and the indexes to databases can consist of large sequences of integer record identifiers. Fast access to numeric data, whether in an index or as part of the data, is essential to the efficient resolution of queries.

For document databases, compression schemes can allow retrieval of stored text to be faster than when uncompressed, since the computational cost of decompression can be offset by reductions in disk seeking and transfer costs [1]. In this paper we explore whether similar gains are available for numeric data. We have implemented several integer coding schemes and evaluated them on collections derived from large indexes and scientific data sets. For comparison, we contrast our results with the theoretical space requirement, the entropy, and with retrieval of uncompressed data.

We describe several different methods of storing integers for fast file access on disk. We select and evaluate schemes that, when carefully coded, yield reasonable compression and fast retrieval of numeric data. In selecting schemes, we have chosen practical approaches that maximise speed, have static coding models, minimise disk space requirements, minimise main-memory use, and—because the model is not adaptive—allow order-independent record-based decompression. Our selection of techniques results in two possible classes: first, we consider the bit-wise Elias gamma and delta codes [2], and parameterised Golomb codes [3]; second, we evaluate byte-wise storage using standard four-byte integers and a variable-byte scheme.

We have found in our experiments that coding using a tailored integer compression scheme can allow retrieval to be up to twice as fast than with integers stored uncompressed. Indeed, for files of more than a few megabytes in size, simple variable-byte representations improve data throughput compared to storing data using standard four-byte integers. Our conclusion is that, for fast access to files containing integers, they should be stored in a compressed format.

2. BACKGROUND

Compression consists of two activities, modelling and coding [4]. A model for data to be compressed is a representation of the distinct symbols in the data and includes information such as frequency about each symbol. Coding is the process of producing a compressed representation of data, using the model to determine a code for each symbol. An efficient coding scheme assigns short codes to common symbols and long codes to rare symbols, optimising code length overall.

Adaptive schemes (where the model evolves as the data is processed) are currently favoured for general-purpose compression [5, 6], and are the basis of utilities such as *compress*. However, because databases are divided into small records that must be independently decompressible [1], adaptive techniques are generally not effective. Moreover, the requirement of atomic decompression precludes the application of vertical compression techniques—such as the READ compression commonly used in images [7]—that take advantage of differences between adjacent records. For text, for example, Huffman coding with a semi-static model is the method of choice because it is fast and allows order-independent decompression [7]. Similarly, arithmetic coding is in general a preferred coding technique; but it is slow for decompression and unsuitable as a candidate for fast file access [8].

In this paper, we investigate semi-static and static modelling schemes that are specific to and effective for integer data. Static modelling approaches have a model implicit in the compression algorithm and, therefore, do not require a model to be independently stored. We investigate using such integer coding approaches to provide faster file access than is available when integers are stored in an uncompressed 32-bit format.

The speed of file access is determined by two factors: first, the CPU requirements of decoding the compressed representation of the data and, second, the time required to seek for and retrieve the compressed data from disk. Without compression, the file access cost is only the time required to retrieve the uncompressed representation from disk. For a compression scheme to allow faster file access, the total retrieval time and CPU processing costs must be less than the retrieval time of the uncompressed representation. It is therefore important that a compression scheme be efficient in both decompression CPU costs and space requirements.

Space efficiency for a given data set can be measured by comparison to the information content of data, as represented by the *entropy* determined by Shannon's coding theorem [9]. Entropy is the ideal compression that is achievable for a given model. For a set S of symbols in which each symbol t has probability of occurrence p_t, the entropy is

$$E(S) = \sum_{t \in S} (-p_t \cdot \log_2 p_t)$$

bits per symbol. Implicit in this definition is the representation of the data as a set of symbol occurrences, that is, modeling of the data using simple tokens. In some domains, different choices of tokens give vastly varying entropy. There are several possible token choices for integer data, the most obvious of which is modeling each integer as a token. We report entropy in Section 4 using this integer token model.

In the remainder of this section, we describe variable-byte and variable-bit integer coding schemes that use static and semi-static models.

2.1. Variable-Byte Coding

With integers of varying magnitudes, a simple variable-byte integer scheme provides some compression. Variable-byte schemes are particularly suitable for storing short arrays of integers that may be interspersed with other data and, because of the byte-alignment, are fast to decode or retrieve from disk. In addition, variable-byte integers allow the storage of arbitrarily large integers.

We use a variable-byte representation in which seven bits in each byte is used to code an integer, with the least significant bit set to 0 in the last byte, or to 1 if further bytes follow. In this way, small integers are represented efficiently; for example, 135 is represented in two bytes, since it lies in the range $[2^7 \cdots 2^{14})$, as 00000011 00001110; this is read as 00000010000111 by removing the least significant bit from each byte and concatenating the remaining 14 bits. Variable-byte codes for selected integers in the range 1–30 are shown in Table 1. A typical application is the coding of index term and inverted list file offset pairs for an inverted index [10].

When storing large arrays of integers, variable-byte integers are generally not as space efficient as variable-bit schemes. However, when storing only a few integers, byte-aligned variable-byte schemes have almost the same space requirements as variable-bit schemes padded for byte-aligned storage. Variable-byte integers also work well with data sets where the structure of the data is unknown and different variable-bit schemes cannot be selectively applied. Moreover, variable-byte integers require few CPU operations to decode.

A pseudo-code description of the variable-byte decoding algorithm is shown in Figure 1. In this algorithm, integers are coded in an array of bytes A on disk and decoded into an integer v. We use the symbols $\triangleleft$ and $\triangleright$ throughout to denote left and right bit-wise shifts. The operation HEAD(A) returns one byte from the head of the array A and then consumes the byte. The coded integer v is retrieved by processing bytes of the array A, where each byte is stored in a single-byte integer i, while the least-significant bit is one. The operation

$$v \longleftarrow (v \triangleleft 7) + ((i \triangleright 1) \text{ BIT-AND } \texttt{0x7F})$$

concatenates the seven most-significant bits of i, which code the integer, to v; the bitwise AND of 0x7F hexadecimal with i right-shifted by one ensures bit that the most-significant bit of i is zero.

Note the return of $v + 1$ in the last step, so that this scheme can only be used for positive integers (as for all the coding schemes in this paper). There are obvious modifications to this scheme and the other schemes to allow representation of zero or negative integers.

2.2. Non-Parameterised Variable-Bit Coding

More efficient coding of larger arrays of integers is possible by using variable-bit schemes. Ideally, however, the structure of data sets to be coded is known and variable-bit schemes appropriate to the magnitude and clustering of the data can be applied selectively. There are two possible approaches to variable-bit coding: non-parameterised coding represents integers using a fixed

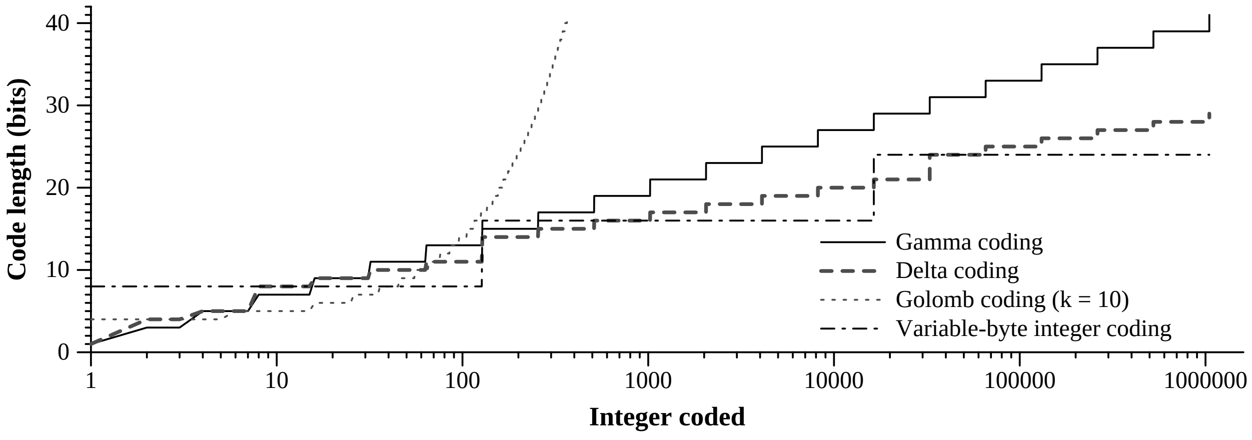

FIGURE 2. *Code lengths in bits of Elias gamma and delta codes, a Golomb code with $k = 10$, and variable-byte integer codes for integers in the range 1 to around 1 million.*

```
VARIABLEBYTEREAD(A)
int  $i = 0x1$
int  $v = 0$

1.  while (($i$ BIT-AND 0x1) = 0x1)
2.      $i \longleftarrow$ HEAD($A$)
3.      $v \longleftarrow (v \lhd 7) + ((i \rhd 1)$ BIT-AND 0x7F)
4.  return $v + 1$
```

FIGURE 1. *Pseudo-code for decompression of variable-byte integers. Each byte stores an integer as the seven most significant bits in each byte, with the least-significant bit indicating whether another byte follows. In this example, a 0 indicates that this is the last byte, while a 1 (`0x1`) indicates that another byte follows. Integers are coded in an array of bytes A on disk and decoded into an integer v. The coded integer is retrieved by removing the least significant bit from each byte and concatenating the remaining bits.*

coding scheme; while parameterised coding represents integers relative to a constant that is calculated or stored for decompression. We discuss in this section static non-parameterised schemes and in the next section semi-static parameterised methods.

Elias coding [2] is a non-parameterised method of coding integers that is, for example, used in large text database indexes [8] and specialist applications [10, 11]. Elias coding, like the other schemes described in this paper, allows unambiguous coding of integers and does not require separators between each integer of a stored array. There are two distinct Elias coding methods, gamma and delta coding.

In the Elias gamma code, a positive integer x is represented by $1 + \lfloor \log_2 x \rfloor$ in unary (that is, $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit), followed by the binary representation of x without its most significant bit. Thus 9 is represented by 0001001, since $1 + \lfloor \log_2 9 \rfloor = 4$, or 0001 in unary, and 9 is 001 in binary with the most significant bit removed. In this way, 1 is represented by 1, that is, is represented in one bit. Gamma coding is efficient for small integers but is not suited to large integers, for which parameterised codes (described in the next section) or the other Elias code, the delta code, are more suitable.

Elias delta codes are somewhat longer than gamma codes for small integers, but for larger integers, such as document numbers in a large index, the situation is reversed. For an integer x, a delta code stores the gamma code representation of $1+\log_2 x$, followed by the binary representation of x less the most significant bit.

Selected Elias codes for integers in the range 1–30 are shown in Table 1. A comparison of code lengths in bits for integers in the range 1 to around 1 million coded with gamma, delta, variable-byte coding, and a Golomb code (described below) is in Figure 2. Observe that for most numbers variable-byte codes are actually shorter than than the other codes, but nonetheless in practice they would be expected to be less efficient: in many applications the distribution of numbers is strongly skewed towards small values, the region where variable-byte codes are least efficient.

Pseudo-code for Elias gamma coding decompression is shown in Figure 3. Integers are shown coded in an array of bits B on disk and are decoded into an integer v. The function HEAD(B) returns one bit from the head of the bit array B and then consumes the bit. The bitwise coded integer v is decoded by processing bits from the array B: first, v is set to 1 to prepend a 1 to the returned value; second, a magnitude m is read in unary, being $1 + \log_2 v$; last, m bits are read, appended to v, and returned. For efficiency, an array of variable-bit codes is usually padded for byte-alignment for storage on disk and the array of variable-bit integers read into main-memory before decoding each integer.

Elias delta coding pseudo-code is shown in Figure 4. Integers are again shown coded in an array of bits B on disk and are decoded into an integer v. The integer v is decoded by first retrieving the gamma-coded value

TABLE 1. *Representations of selected integers in the range 1–30 as uncompressed eight-bit integers, Elias gamma codes, Elias delta codes, Golomb codes with $k = 3$ and $k = 10$, and variable-byte integers. Spacing in the Elias, Golomb, and variable-byte codes separates the prefix of the code from the suffix.*

Decimal	Uncompressed	Elias Gamma	Elias Delta	Golomb ($k = 3$)	Golomb ($k = 10$)	Variable-byte
1	00000001	1	1	1 10	1 001	0000001 0
2	00000010	0 10	0 100	1 11	1 010	0000010 0
3	00000011	0 11	0 101	01 0	1 011	0000011 0
4	00000100	00 100	0 1100	01 10	1 100	0000100 0
5	00000101	00 101	0 1101	01 11	1 101	0000101 0
6	00000110	00 110	0 1110	001 0	1 1100	0000110 0
7	00000111	00 111	0 1111	001 10	1 1101	0000111 0
8	00001000	000 1000	00 100000	001 11	1 1110	0001000 0
9	00001001	000 1001	00 100001	0001 0	1 1111	0001001 0
10	00001010	000 1010	00 100010	0001 10	01 000	0001010 0
11	00001011	000 1011	00 100011	0001 11	01 001	0001011 0
12	00001100	000 1100	00 100100	00001 0	01 010	0001100 0
13	00001101	000 1101	00 100101	00001 10	01 011	0001101 0
14	00001110	000 1110	00 100110	00001 11	01 100	0001110 0
15	00001111	000 1111	00 100111	000001 0	01 101	0001111 0
16	00010000	0000 10000	00 1010000	000001 10	01 1100	0010000 0
20	00010100	0000 10100	00 1010100	0000001 11	001 000	0010100 0
25	00011010	0000 11001	00 1011001	000000001 10	001 101	0011010 0
30	00011110	0000 11110	00 1011110	00000000001 0	0001 000	0011110 0

```
GammaRead(B)
int  $m = 0$
int  $i,j$
int  $v = $ 0x1

1.  $i \longleftarrow$ HEAD($B$)
2.  while ($i =$ 0x0)
3.        $m \longleftarrow m + 1$
4.        $i \longleftarrow$ HEAD($B$)
5.  for $j = 1$ to $m$
6.        $i \longleftarrow$ HEAD($B$)
7.        $v \longleftarrow (v \lhd 1)$ BIT-OR $i$
8.  return $v$
```

FIGURE 3. *Pseudo-code for the decompression of variable-bit Elias gamma codes. Elias gamma codes store an integer v as $1 + \lfloor \log_2 v \rfloor$ in unary followed by the binary representation of v without its most significant bit. In this pseudo-code, integers are coded in an array of bits B on disk and decoded into an integer v; for efficiency in all variable-bit coding, the array is typically padded for byte-alignment.*

of $1 + \log_s v$ as the magnitude m and then retrieving the binary representation of v less the most significant bit. As in gamma coding, a 1 is prepended to v by initialising v as 1, and v returned.

We have described above unary coding as part of the implementation of Elias gamma coding. Unary coding is, however, a scheme that can be used independently as a bit-wise coding scheme for very small integers. Applications of unary coding are, however, limited, as the length of the code for an integer v is v and only small integers can be coded efficiently. Because of this limitation of unary coding, we do not experiment with it as an independent scheme in our experiments.

For applications where integers are sorted, non-parameterised coding can be used to code the differences between the second and successive integers, rather than the integers. For example, consider an array of integers

$$725, 788, 1045, 6418, \ldots$$

that can, after taking differences between integers, be coded as

$$725, 63, 257, 5373, \ldots$$

While the savings are not immediately obvious, if this short array were delta coded without taking differences, 68 bits would be required to store the integers, but by taking differences only 60 bits are required.

2.3. Parameterised Variable-Bit Coding

As we show later, Elias codes yield acceptable compression and fast decoding, however better performance in both respects is possible with parameterised techniques, such as Golomb codes [3, 7]. The restriction, however, is that a parameter k must be calculated (and, in many cases, stored) with each array of coded integers. Indeed, the choice of k has significant impact on the compression using a parameterised model. We discuss Golomb coding below and briefly discuss choice of k later.

```
DeltaRead(B)
int  $i,j$
int  $m$
int  $v$ =0x1

1.  $m \longleftarrow$ GammaRead(B)
2.  for $j = 1$ to $m$
3.      $i \longleftarrow$ head$(B)$
4.      $v \longleftarrow (v \lhd 1)$ bit-or $i$
5.  return $v$
```

FIGURE 4. *Pseudo-code for the decompression of variable-bit Elias delta codes. Elias delta codes store an integer v as the Elias gamma code of $1 + \log_2 v$, followed by the binary representation of v without its most significant bit. Integers are coded in an array of bits B on disk and decoded into an integer v.*

Using Golomb coding, a positive integer v is represented in two parts: the first is a unary representation of the quotient $q = \lfloor (v-1)/k \rfloor + 1$; the second is a binary representation of the remainder $r = v - q \times k - 1$. Using this approach, d different remainders can be stored using $i = \lfloor \log k \rfloor$ bits, where $d = 2^{i+1} - k$, and all other remainders require $i + 1$ bits. That is, the binary representation of the remainder r may require $\lfloor \log k \rfloor$ or $\lceil \log k \rceil$ bits.

A pseudo-code description of the Golomb decoding algorithm is shown in Figure 5. In this description, an integer v is retrieved from a bit array B using a constant parameter k. As previously, head(B) returns and consumes one bit from the head of B. Steps 1 and 2 initialise constants; we discuss these later. Steps 3–6 retrieve from the bit array B the unary coded quotient q, where $q = \lfloor (v-1)/k \rfloor + 1$. Steps 7–9 retrieve the shortest possible bit-string representing the remainder r, that is, the first i bits calculated in step 1 where $i = \lfloor \log_2 k \rfloor$. The constant d from step 2 is used to determine the "toggle point" where an extra bit is required to code the remainder r. Step 10 tests whether the remainder r retrieved so far is greater than the "toggle point" d and, if so, an additional bit is retrieved in steps 11 and 12. The value of v is calculated by multiplying the quotient q by the constant k and adding the remainder $r + 1$ in step 13. Step 14 returns v.

Several optimisations in this Golomb decoding algorithm are possible. First, calculating the values of $i = \log_2 k$ and $d = (1 \lhd (i+1)) - k$ for each integer in an array of coded integers is unnecessary, since each integer uses the same k value; accordingly, an array of k, i, and d values can be stored and each constant calculated once only. Second, bits can be retrieved in blocks: $\log_2 k + 1$ bits may be retrieved for the remainder (which, in many cases, is one bit more than needed, requiring that the "extra bit" be stored in memory and passed as a parameter to decode the next Golomb-coded integer). Last, larger blocks of bits may also be retrieved from an in-

```
GolombRead($B,k$)
int  $q = 0$
int  $r$ = 0x0
int  $v,d,i,j,l$

1.  $i \longleftarrow \log_2 k$
2.  $d \longleftarrow (1 \lhd (i+1)) - k$
3.  $l \longleftarrow$ head$(B)$
4.  while ($l$ = 0x0)
5.      $q \longleftarrow q + 1$
6.      $l \longleftarrow$ head$(B)$
7.  for $j = 1$ to $i$
8.      $l \longleftarrow$ head$(B)$
9.      $r \longleftarrow (r \lhd 1)$ bit-or $l$
10. if $(r > d)$
11.     $l \longleftarrow$ head$(B)$
12.     $r \longleftarrow ((r \lhd 1)$ bit-or $l) - d$
13. $v \longleftarrow q \times k + r + 1$
14. return $v$
```

FIGURE 5. *Pseudo-code for the decompression of parameterised variable-bit Golomb codes. Integers are coded in an array of bits B using a constant k and are decoded in to an integer v. Golomb codes store the integer v as a quotient q and a remainder r. The quotient q is stored in unary, while the remainder is stored in binary requiring $\lfloor \log k \rfloor$ or $\lceil \log k \rceil$ bits (either i or $i+1$ bits). Steps 7–9 retrieve the first i bits of the remainder, while the pre-calculated value of d and the* **if** *test (steps 10–12) are used to assess whether a further bit is required and to retrieve that bit.*

memory bit array encompassing both the quotient and the remainder, where bits that are not processed in decoding an integer may be subsequently restored to the bit array. Such efficiencies are important in decoding arrays of Golomb coded integers to ensure that, while they are generally more space efficient that Elias codes, that they are also faster to decode.

Selection of the parameter k is also important for both the space and time efficiency of Golomb coding. Witten et al. [7] report that, for cases where the probability of any particular value occurring is small, an approximate calculation of k can be used. Where there is a wide range of values to be coded and each occurs with reasonable frequency, a practical global approximation of the Golomb parameter k is

$$k \approx 0.69 \times \text{mean}(v)$$

This model for selection of k is referred to as a Bernoulli model, since each coded integer is assumed to have an independent probability of occurrence and the occurrences of different values have a geometric distribution.

Skewed Bernoulli models, where a simple mean difference is not used, typically result in better compression than simple global models [12]. We have experimented with global Bernoulli models, but not with skewed models because of the increased complexity in calculating and storing appropriate k values. However, we would expect further small space savings through using a skewed model. Golomb codes for integers in the range 1–30, with two parameters of $k = 3$ and $k = 10$, are shown in Table 1. Further compression is also possible in specific applications by using semi-static models for different probability distributions, such as in inverted index posting lists where integers are sorted in increasing order and integers are often clustered [13].

3. TEST DATA

To evaluate the performance of integer coding techniques for fast file access, we use collections derived from scientific data and inverted indexes. In this selection we have focused on collecting large and small data sets that consist primarily of numeric data. We have removed from each data set any textual record markers and stored the numeric data as uncompressed 32-bit integers. These uncompressed integer collections are used as the baseline for comparison to the compressed variable-bit and variable-byte coded collections.

The scientific data sets used are derived from weather station measurements, temperatures, map contours, satellite greyscale image data, and prime numbers. The weather data (WEATHER) contains 20,175 records collected from each of 5 weather stations, where each station record contains 4 sets of 22 measurements (such as temperatures, elevations, rainfall, and humidity); in total WEATHER is 38.2 Mb. A much smaller temperature data set (TEMPS) was collected at ten minute intervals over a period of around 40 days from the Australian Bureau of Meteorology. The TEMPS data set contains measurements of temperatures in Melbourne, Australia and surrounding districts, where each temperature measurement is prefixed by a system clock value, giving a total of 2.8 Mb. A complex land contour map (MAP) was exported from a proprietary mapping application and stored as a file of integer elevations for all map points, giving 754.5 Mb. Our land satellite data (LANDSAT) contains layered satellite images of the Gippsland lakes (Victoria, Australia) used in water quality experiments, where each layer is collected at a different frequency spectrum, giving a total of 156.4 Mb. A prime number collection of the first one million prime numbers (PRIME) is 3.8 Mb.

Integer compression has been shown previously to offer space efficient representation of indexes [8, 13, 14]. Inverted file indexes contain large sorted arrays of integers or *postings* representing occurrences of terms in a collection. Each postings array contains a sorted list of document identifiers and, in many indexes, interspersed between each document identifier is a sorted array of one or more word positions of that term in the document. We experiment with an uncompressed index postings list of 630.5 Mb (VECTOR) extracted from a much larger carefully compressed 3,596 Mb postings file used in the CAFE [10] indexed genomic retrieval system.[1] This postings list contains both document identifiers and interleaved word positions.

In several experiments we compress integers without considering structure inherent in the data. That is, we apply the same compression scheme to each collection, treating the collection as a stream of integers; in the case of Golomb coding, we use one calculated global coding constant k for each file. This is a worst-case for compression schemes, since structure can be used to achieve better compression by choosing coding methods appropriate to the magnitude, ordering, and clustering of integers. For example, consider an index postings list: document identifiers may be represented by taking differences between adjacent identifiers and coded with a local parameterised Golomb code; counts of word positions for each document may be stored using a gamma code; and word positions may be coded as differences using a separate local parameter to a Golomb code. To illustrate the fast file access possible by using known structure of a file type, we also show experiments with TEMPS, PRIME, and VECTOR where such selected variable-bit coding has been applied.

4. RESULTS

We investigate both sequential stream access and random access to our test collections. Random access to compressed data is a requirement in indexing and a likely method of access to many scientific data sets. For example, individual layered images may be retrieved from a set of satellite images, only selected weather station data may be required from a large database, or individual spatial map components may be rendered in a mapping system. Zobel and Moffat [1] explored access speeds to uncompressed and compressed text, and showed that compressed text gives the relative best performance during random-access, when reductions in seek times are significant; in contrast, with sequential access only transfer rates and CPU time are involved. Compared to sequential access, therefore, we would expect random access to compressed data to be relatively faster than random access to uncompressed data.

To allow random access, we store a separate file of offsets for each collection, where each offset represents a file position in the collection that is the beginning of a block of 1,000 integers; for VECTOR, we do not block the data in this way, but instead—in the same way as the CAFE genomic retrieval system [10]—we treat each postings list as a record, where the average record length is 23,866 integers. We chose 1,000 integers per block to simulate record structure, as this was around the size of a typical record in WEATHER and an image

[1]We were unable to experiment with the full uncompressed index, as the integer representation was impractical to store as a single file (because of operating system file size limits). An uncompressed integer representation of the full postings array would require almost 13 Gb.

TABLE 2. *Compression performance of integer coding schemes, in bits per integer. The first line shows the size of each data set. For* temps, prime, *and* vector *the selected compression scheme is an approach that uses a combination of different coding schemes that rely on known structure in the data; for other collections, the selected compression is the best-performing approach of*

Scheme	temps	prime	weather	landsat	map	vector
Integers ($\times 10^6$)	0.72	1.00	10.00	41.01	197.80	165.29
Entropy	12.57	19.93	2.91	6.02	6.50	17.40
Elias gamma coding	33.50	44.65	16.57	8.42	11.02	11.42
Elias delta coding	23.80	30.84	12.82	8.09	10.19	9.78
Golomb coding	26.54	24.36	13.64	6.60	7.50	13.47
Variable-byte coding	22.11	30.74	12.59	8.00	8.63	11.97
gzip	10.21	10.91	3.00	4.53	0.24	11.82
Selected compression	7.14	5.52	12.59	6.60	7.50	7.87

TABLE 3. *Sequential stream retrieval performance of integer coding schemes, in megabytes per second. In each case data is retrieved from disk and, in all but the first case, decompressed.*

Scheme	temps	prime	weather	landsat	map	vector
Uncompressed 32-bit integers	2.34	2.31	2.19	2.39	2.48	1.98
Elias gamma coding	1.05	1.03	1.96	3.08	2.49	2.24
Elias delta coding	1.40	1.42	2.29	2.86	2.46	2.47
Golomb coding	1.77	1.85	2.31	3.25	3.13	2.30
Variable-byte coding	2.12	1.42	3.67	4.45	5.41	2.69
gzip	3.83	4.14	12.72	9.25	25.68	4.50
Selected compression	2.42	2.72	3.67	3.25	3.13	2.78

line in landsat. For our random access experiments, we report the speed of randomly seeking to 10% of the offsets in each collection and retrieving blocks of 1,000 integers at each offset; in the case of vector, each seek retrieves a mean of 23,866 integers.

For random-access experiments with variable-bit coding, each block of 1,000 integers is byte-aligned on disk, wasting on average 4 bits per 1,000 integers stored. Each random seek in the collection is then to a byte-aligned offset. As an example, the vector file requires an additional 0.2 Mb for byte-aligned storage, an increase of around 0.1% over an approach that only supports stream access.

All experiments are carried out on an Intel Pentium II dual processor machine running the Linux operating system under light load. Before each experiment the system cache and main memory were flushed to ensure that retrieval is directly from disk.

We show the results for compression and sequential retrieval of data from the six integer collections in Tables 2 and 3. The first block of Table 2 shows parameters of the collections: file sizes are shown in millions of integers and the entropy or theoretical minimum space requirement is shown as a baseline comparison for compression performance. The entropy is based on the integer-token model described earlier. Entropies are much less than 32 bits per integer for weather, landsat, and map, as these three collections contain relatively few distinct integers: weather stations only take 22 measurements that often do not vary significantly and are often substituted with 9999 to represent "no data collected", the landsat data contains only 7-bit integers, and map contains contour data that is clustered about the mean. In contrast, the entropies of the other collections are more than 10 bits per integer: the prime collection contains no duplicates, half of the integers in temps are distinct clock values, and vector stores over 16 million different document identifiers and offsets.

Table 2 shows the compression achievable with the different coding schemes. The variable-byte scheme was more effective than we had expected, always outperforming gamma, and in some cases delta and Golomb coding (although these schemes can be used to achieve better overall compression by mixing codes for special cases, as we discuss below). This good result for variable-byte coding is why it allows fast retrieval: it yields similar disk traffic to the other schemes but has lower CPU costs. For comparison we include results for the popular utility gzip, to illustrate the space savings possible by applying an efficient adaptive compression scheme. In all cases, gzip compression is excellent and, except for vector, better than with the generic static and semi-static compression schemes. With regard to speed, gzip provides a further illustration that the cost of decompression can be more than offset by savings in

TABLE 4. *Random-access retrieval performance of integer coding schemes, in megabytes per second. In each case data is retrieved from disk and, in all but the first case, decompressed.*

Scheme	TEMPS	PRIME	WEATHER	LANDSAT	MAP	VECTOR
Uncompressed 32-bit integers	0.31	0.49	0.39	0.33	0.34	0.70
Elias gamma coding	0.23	0.33	0.33	0.61	0.58	0.67
Elias delta coding	0.32	0.45	0.33	0.50	0.48	1.00
Golomb coding	0.34	0.49	0.46	0.68	0.54	0.83
Variable-byte coding	0.35	0.58	0.58	0.51	0.49	0.75
Selected compression	0.92	0.83	0.58	0.68	0.54	1.29

disk transfer costs. However, because GZIP is adaptive it cannot be used for random access to data and, therefore, is not a candidate for fast file access to databases of integers.

The final block of Table 2 shows how more efficient representation is possible by selectively applying variable-bit codes to the VECTOR, TEMPS, and PRIME collections. In the case of VECTOR we use separate local Golomb parameters for each list of document identifiers and word positions, and gamma codes for storing counts of identifiers in each list. For TEMPS we use two different Golomb parameters: the first for storing differences between the large system clock values and the second for storing the smaller temperatures. The values shown in the final block of Table 2 for the WEATHER, LANDSAT, and MAP collections are the values of the most space-efficient scheme from those listed in the rows above.[2]

Table 3 shows sequential stream retrieval speed for the uncompressed 32-bit data and the remaining blocks show the performance of the variable-bit and variable-byte schemes. All timing measurements are in megabytes per second, and include the time taken to retrieve data from disk and decompress it. In the case of parameterised Golomb coding, the parameter k is fixed for each collection and uses the global approximation described earlier. Note that the timings for VECTOR are not directly comparable to the other results, as they were determined under CAFE; the other timings are based on a suite of special-purpose code. Also, note that the timings with GZIP are not directly comparable to other figures and are included for illustration only: GZIP performs well because of the small compressed sizes and because, in contrast to our schemes, output is buffered into large blocks.

In our sequential retrieval experiments, one or more compression schemes for integers improves both the space requirements and the retrieval speed for each collection compared to storing uncompressed integers. By selecting an appropriate compression scheme for a file—as shown in the "selected compression" experiments in Tables 2 and 3—speed is 4%–68% better than with uncompressed integers. Importantly, for the LANDSAT, MAP, and VECTOR experiments, where there is substantially more data to be processed than in the other smaller collections, speed with the selected compression technique is 26%–40% better than with uncompressed integers, while the space requirement averages 21%–25% of the cost of storing uncompressed integers.

As expected, non-parameterised Elias coding is generally not as space-efficient or as fast as parameterised Golomb coding for sequential retrieval. Gamma coding does not work well as a fast retrieval technique on most of our test collections, since the average magnitude of the integers is higher than that for which gamma coding is suitable; the exception is for the 7-bit integers stored in the LANDSAT file. Delta coding works well for two larger collections, affording a 20% improvement in speed for LANDSAT and a 25% improvement for VECTOR, but is marginally slower for the MAP collection. In general, when a parameterised scheme cannot be easily used, delta coding saves space compared to uncompressed storage and can often improve retrieval performance.

Golomb coding with a single parameter k works well as a fast retrieval technique in all cases other than the small TEMPS file. Moreover, the selected compression techniques for PRIME and VECTOR illustrate that excellent performance is possible by careful selection of parameters to Golomb coding and by taking differences between sorted integers to reduce the range values stored. Additionally, Golomb coding works reasonably well for TEMPS when two parameters are used as in the selected compression scheme.

Variable-byte integers work well as a fast retrieval technique for all collections where large integers are stored. For the WEATHER, MAP, and VECTOR collection, variable-byte integers consume 27%–39% of the space required by uncompressed integers and retrieval is 1.35 to 2.18 times as fast. Variable-byte integers are often faster than variable-bit integers, despite having higher storage costs, because fewer CPU operations are required to decode variable-byte integers and they are byte-aligned on disk. As an example, for the VECTOR collection the CPU cost of decoding variable-byte inte-

[2]By modifying the data in WEATHER to use a different value as a "no data collected" marker, compression of 11.10 bits per integer can be achieved. By extending this approach further and coding "no data collected" markers as a single bit followed by a count of the number of markers and storing each collected value as a single bit, a counter, and a Golomb-coded value, compression of 2.85 bits per integer is possible.

gers is around 20% less than decoding either delta codes or the selected compression scheme. However, because carefully selected variable-bit codes require less space than variable-byte codes, such reductions in CPU costs are generally insignificant compared to the increased disk retrieval costs.

Table 4 shows the speed of integer compression schemes for random access to our test collections. As described earlier, decompression speeds are the throughput in Mb per second of randomly seeking to 10% of the byte-aligned offsets in each collection and retrieving 1,000 integers at each offset; for VECTOR, we randomly seek to 10% of postings list offsets, but retrieve on average 23,866 integers per list (hence the greater throughput).

The performance of integer compression schemes for random retrieval is even better. As expected, because of the smaller compressed file sizes, in addition to improved sequential stream retrieval of each compressed record, disk seek costs are also substantially reduced. This improvement in performance is marked for the larger LANDSAT, MAP, and VECTOR collections where the selected compression scheme is between 1.58 to 2.06 times as fast as retrieval of uncompressed integers. Indeed, in all cases except the inefficient gamma coding of PRIME, random retrieval speed is improved by storing integers in a compressed format.

5. CONCLUSIONS

We have shown that storing integers in compressed form improves the speed of disk retrieval for both sequential stream access and random access to files. Disk retrieval performance can be improved by applying either a variable-bit or variable-byte integer compression scheme. Selection of a scheme that is specific to the magnitude, ordering, and clustering of integers, as well as the file size, offers the best performance. For sequential stream retrieval, applying a selected variable-bit compression scheme to files larger than 100 Mb improves access speed by more than 25% over storage as standard uncompressed 32-bit integers. For smaller files, variable-byte schemes offer similar performance increases. For random retrieval, speed improvements are more marked, since average disk seek costs are reduced because of the reduced file sizes: for larger data sets, a carefully selected variable-bit compression scheme is at least 55% faster than an uncompressed scheme; and both variable-bit and variable-byte schemes work well for random-access to small collections.

Although the compression schemes described here are extremely simple, disk retrieval costs are reduced by compression since the cost of retrieving a compressed representation from disk and the CPU cost of decoding such a representation is usually less than that of retrieving an uncompressed representation. Improving retrieval performance in this way can yield significant performance gains for data processing applications such as databases: indexes contain long sequences of integers, most databases contain integer components, and many scientific repositories, such as weather, satellite, and financial databases, contain integers only. For fast access to files of integers, given that addition of compression to read or write utilities requires only a few lines of code, we recommend that they should always be stored compressed.

ACKNOWLEDGEMENTS

We thank Agnes Apostolou, Neil Christie, Neil Sharman, and Lachlan Wainwright for making available their satellite, contour, and temperature data. We also thank the referees for their comments. This work was supported by the Australian Research Council and the Multimedia Database Systems group at RMIT.

REFERENCES

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

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

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

[4] J. Rissanen and G.G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, IT-27(1):12–23, January 1981.

[5] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[6] A. Moffat, R. Neal, and I. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 1998. (To appear).

[7] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, 1994.

[8] T.C. Bell, A. Moffat, C.G. Nevill-Manning, I.H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9):508–531, October 1993.

[9] C.E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:55, 1951.

[10] H. Williams and J. Zobel. Indexing nucleotide databases for fast query evaluation. In *Proc. International Conference on Advances in Database Technology (EDBT)*, pages 275–288, Avignon, France, March 1996. Springer-Verlag. Lecture Notes in Computer Science 1057.

[11] H. Williams and J. Zobel. Compression of nucleotide databases for fast searching. *Computer Applications in the Biosciences*, 13(5):549–554, 1997.

[12] A. Moffat, J. Zobel, and S. T. Klein. Improved inverted file processing for large text databases. In *Proc. Australasian Database Conference*, pages 162–171, Adelaide, Australia, January 1995.

[13] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. In *Proc. IEEE Data Compression Conference*, pages 82–91, Snowbird, Utah, 1996.

[14] A. Bookstein, S.T. Klein, and D.A. Ziff. A systematic approach to compressing a full-text retrieval system. *Information Processing & Management*, 28(6):795–806, 1992.