Compression for Fast Read-only Access of a Large Database

Paul D. Filipski Jonathan J. Hull
Sargur N. Srihari
Center of Excellence for Document Analysis and Recognition
State University of New York at Buffalo
226 Bell Hall
Buffalo NY 14260
filipski@cs.buffalo.edu

August 10, 1992

Abstract

An approach for the compression and high speed access of a large read-only database is described. This method uses a set of pre-compiled search trees to access compressed source data. A technique known as *attribute coding* uses local characteristics of the source data to determine an optimal compression strategy. This paper describes the overall approach, including the design of the search trees and the attribute coding. An application of this method using the ZIP+4 file of the United States Postal Service is also discussed. This application achieves an effective compression ratio of 25:1. Also, efficient access to the data for a real-time address recognition system is achievable.

1 Introduction

Artificial intelligence systems often require high-speed access to very large databases that store the knowledge available to the system. While these systems are processing, they consult their store of knowledge frequently, using the stored data to check hypotheses, verify conclusions, etc. The data needed by these system is usually very structured, for example, a natural language processing system requires a lexicon providing words, parts of speech, variant forms, and other information. This paper describes the implementation of a structured database to support an address interpretation system.

The database that we describe is not unlike a full-text retrieval system ([8] [11]) in that it permits flexible access to the stored source data. However, our method takes advantage of the structured nature of the source data to achieve better source compression and more efficient access to the data.

USPS ADVANCED TECHNOLOGY CONFERENCE, Nov. 30 - DEC. 2, 1992, WASHINGTON, D.C.
USPS Advanced Technology Conference, Washington, D.C., Nov. 30 - Dec. 2, 1992

Table 1: Example ZIP+4 records

Record Number	*ZIP Code*	*Street name*	*Suffix*	*Address range*	*Add-on range*
1	14225	FAIRHAVEN	DR	0 18 E	1806
2	14225	FAIRHAVEN	DR	1 19 O	1805
3	14225	FAIRHAVEN	DR	20 58 E	1859
4	14225	FAIRHAVEN	DR	21 59 O	1860
5	14225	PO BOX		1581 1586 B	8581 8586
6	14225	PO BOX		1588 1588 B	8588
7	14225	PO BOX		1590 1594 B	8590 8594
⋮			⋮		
257	14227	WILLIAM	ST	3017 3199 O	1805

The methods that will be described were applied to the USPS ZIP+4 File. The ZIP+4 File provides a comprehensive list of every ZIP+4 Code and its address. A ZIP+4 Code is made up of the traditional five-digit ZIP Code and an additional four digit add-on. The ZIP Code is used to identify an area of the country and the delivery office to which the mailpiece is directed. The add-on denotes a more specific destination, which might be the floor of an office building, the side of a street, or a group of post office boxes.

The national ZIP+4 directory is composed of about 30 million records, and contains around 2.2 GB of data. Each record in the directory gives address information for a particular ZIP+4 Code or range of ZIP+4 Codes. The information includes a street name and suffix, and a range of address numbers on the street, as well as the ZIP Code and add-on. Given the full address for any delivery point in the United States, it is possible to find a ZIP+4 Code for that delivery point in the ZIP+4 directory. Table 1 shows some sample ZIP+4 records.

Additionally, the ZIP+4 file can be thought of as providing constraint information amongst attributes of its records. These constraints tell whether or not given values for two or more attributes are compatible, i.e., whether they appear together in any record. For example, the street name "FAIRHAVEN" and suffix "DR" are compatible in the above table, while the street name "FAIRHAVEN" and street number 250 are not. This use of the data as constraints is essential for constraint satisfaction problem solvers ([7]).

The address recognition system ([2]) interprets an address by first segmenting a digital image of an address block into individual words that are then recognized. Since word recognition can provide imperfect results, a search process is conducted over the ZIP+4 file to find the record that best matches the given information. The system may need to consider many possible values for each word to determine a solution. In the course of searching for a solution, it is possible for the address recognition system to make hundreds of queries to check the compatability of pairs of words. Since each mail piece must be processed in less than 700 milliseconds, it is essential that the database system have a response time of less than a millisecond. Since it would be

very difficult for a disk-based system to support this access rate, the entire database must be stored in main memory.

Section 2 gives a general description of our implementation of the database. Sections 3 and 4 present the compression methods used for the ZIP+4 File and inverted indices, respectively. Section 5 summarizes the performance of the database system.

2 Proposed Solution

We have developed a two-level system that supports queries that specify values for any subset of the attributes of a record. To allow accesses on many different fields, we construct data structures for every field that allow efficient access to records having a specified value for that field. After construction, the data structures are compressed using techniques based on bit-vector compression ([5] [10]) that will be described later. The source data is also compressed, using the technique of attribute coding ([3] [9]).

Search trees are added to the compressed source data to permit fast access to source records. These trees allow the retrieval of a list of pointers to every record containing a specific value for a given attribute. One tree is constructued for each attribute; each tree contains one node for each possible value that the attribute may assume. Attached to each node is a list of pointers. This list contains one pointer to each record in the source data that has the given value for that attribute. To answer queries that specify values for more than one attribute, the appropriate search tree is consulted to retrieve a list of pointers for each value. These lists are then intersected to compute the final answer to the query.

Using these principles, we have constructed a read-only database that supports the address recognition system described in Section 1. The system contains a full implementation of attribute coding used to compress the ZIP+4 file. Also, search trees were constructed for each of the attributes in the ZIP+4. Figure 1 shows a small excerpt from the ZIP+4 file and portions of the associated data structures for ZIP Code, street name, and ZIP+4 add-on. This example shows two nodes in the street name tree, one for "FAIRHAVEN", with four associated pointers, and one for "PO" with three associated pointers. In a real implementation, these lists of pointers would actually be much larger, and reference records across the entire database.

3 Attribute Coding

Large structured databases typically contain a considerable amount of vertical redundancy. That is, information in consecutive records is often similar. For example (see Table 1), in the ZIP+4 file, all records for a particular street are grouped together in the file. Clearly, any successful compression technique must take advantage of this redundancy.

The method of attribute coding takes advantage of the relationships between records in a structured database by storing a record as a set of differences from the previous record. These differences are specified by functions and their arguments.

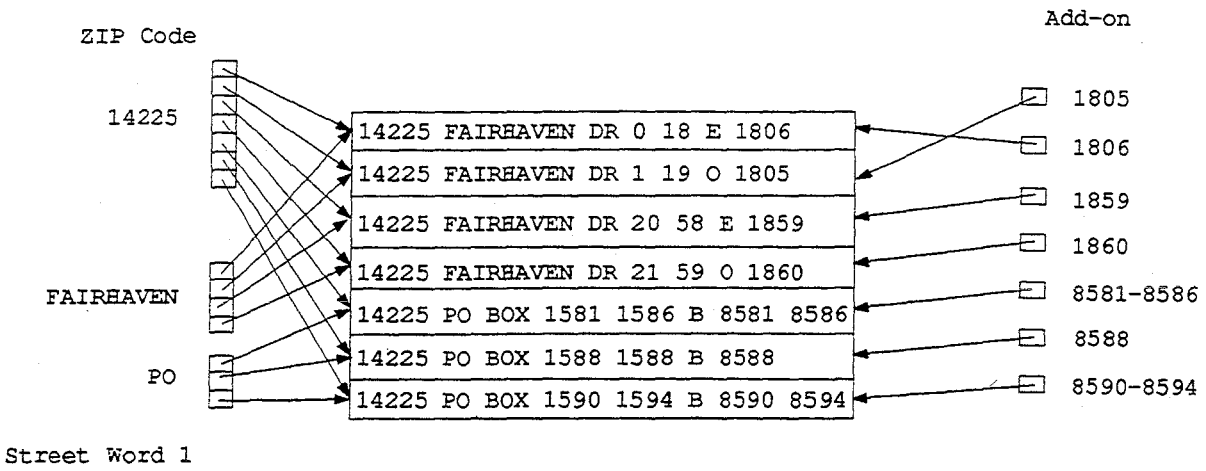

Figure 1: Sample ZIP+4 file and data structures

With the aid of a statistical analysis of the database, a set of functions is devised for each attribute. These functions have access to information in the previous record, as well as information that has already been computed in the current record. Also, a function may take an argument, and use it in addition to the other information to produce the new attribute value for the current record. Each function in the set is designed to encode in a small number of bits one statistically likely change in the attribute's value.

Attribute coding is based on the idea that knowledge of the values of earlier attributes in a record helps us to predict the values of later attributes. If B_i denotes attribute i of record B, attribute coding would express B_i as a function of all earlier attributes. That is, $B_i = f(B_1, B_2, \ldots, B_{i-1})$. Also, further information can be obtained from the vertical redundancy of the database. If A and B are consecutive records in the structured database, attribute coding would express each B_i as a function of not only the previous attributes in B, but also all the attributes of A. That is, $B_i = f(A, B_1, B_2, \ldots, B_{i-1})$.

In practice, there is not a strong statistical correlation between each attribute and all of the parameters that it theoretically depends on. Therefore each attribute is a function of only a few of these other attributes. For example, attribute B_i may depend on only the value A_i, for the same attribute in the previous record, and not on the value of any of the other attributes from records A or B.

There is another practical point to consider. Since random access to the data is needed, it is not feasible to have every record depend on the record before it. Then to recover record n, we would need to recover every record before n. This is clearly too time consuming when dealing with millions of records. Therefore, we compress these records in groups of N. This means that we will never need to decompress more

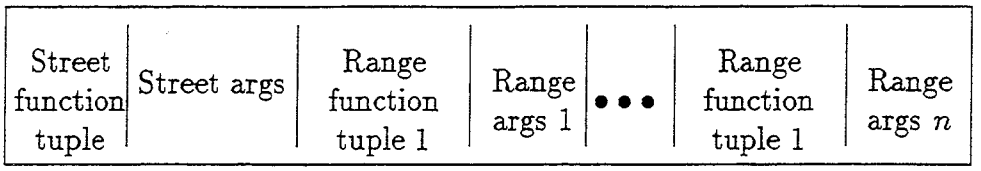

Figure 2: Compressed record format.

than N records to retrieve the one that we want. The value of N can be adjusted for a space/time tradeoff. As N increases, longer sequences of dependent records can be encoded, reducing the storage space needed. However, larger values of N mean that more records need to be recovered to retrieve one specific record, which requires more decompression time.

To perform the compression of a single record, we decide which functions will best compress each attribute. Identifiers for the functions used and their arguments are then stored.

Since the records in the source data are typically very redundant, some functions are much more likely to appear than others. To take advantage of this non-uniform distribution, the function identifiers are stored using Huffman codes ([6]). Huffman codes are used because they can be decompressed in constant time with a table lookup. If the maximum length of any Huffman code bit string is n bits, a table of 2^n entries is sufficient. Although an exponentially-sized table may be unacceptable for most applications, in large databases these tables make up an insignificant fraction of the total size. For example in the ZIP+4 database, the tables consume only one two-hundredth of the total storage.

Example Let us now consider a simplified example that will demonstrate how attribute coding works. This example will consider reduced ZIP+4 source records that contain only a ZIP Code, street name, address range and four digit add-on. The records that will be used are shown in Table 2. For the purpose of attribute coding, consecutive records in the source file that contain identical ZIP Code and street information are grouped into one larger record. This record has a hierarchical structure – one ZIP Code and street name that are applied to a number of address ranges, with each address range having its own ZIP+4 add-on. Using this method, it is necessary to store the coded version of the ZIP Code and street information once for the entire group of records, instead of once for each of the source records.

When compressing the ZIP+4 database, we write a tuple of the functions that were used to compress the street information (called the "street function tuple") and then the arguments for each of the functions. Similarly, for each address range, a tuple of range functions is stored, along with the arguments for each function. The format of the compressed records is shown in Figure 2.

To compress these records we will use a reduced set of functions for each of the attributes in our records. The functions that will be available are shown in Table 3. In this table, the notation arg_n refers to an argument of n bits. Also, an attribute

Table 2: Attribute coding example records.

ZIP Code	*Street name*	*Street number* *low*	*high*	*add-on*
14202	MAIN	100	198	3969
		101	199	3970
		200	399	3950
14202	MAPLE	100	199	6110
		124	124	9221

name followed by a prime (′) refers to the value of the same attribute in the previous record.

Table 3 presents the functions grouped by attribute. Along with each function is the probability that it will be used. This probability is determined experimentally through analysis of a portion of the source records, and is used in designing the Huffman code for the street and range function tuples.

The sets of attribute coding functions for the ZIP Code and street name attributes are relatively straightforward. Each set has two functions, one designed to be used when the attribute's value is the same in two consecutive records ($z5 = z5'$ and $st = st'$), and one that is to be used when the attribute's value changes ($z5 = \text{table}[\text{arg}_{16}]$ and $st = \text{table}[\text{arg}_{19}]$). For the ZIP Code attribute, "table" refers to a table of all legal ZIP Codes. There are about 45,000 legal ZIP Codes in the United States, so 16 bits are sufficient to reference an element in this table. Similarly, the table mentioned for the street name attribute is a list of all street names observed in the ZIP+4 file. This table contains around 400,000 entries, so 19 bits are needed to index into it.

There are two functions used to compress the number of ranges attribute (given as nr in Table 3). The first is $nr = 64$; this is for records with a full set of address ranges. The analysis of the source file shows that this function captures 15% of all cases. The other values taken on by the number of ranges attribute were fairly well-distributed in the range $1 \ldots 63$, so one additional function suffices.

An address range's street number low attribute is stored in two parts – the block difference (*snlbd*) and the remainder difference (*snlrd*). Here the block difference is defined to be the difference in all but the last two digits of an address number, and the remainder difference is the difference in just these last two digits. The block difference between two consecutive address ranges is much more stable than the remainder difference, as shown by the probability of the first function, $snlbd = 0$. Six functions are available to encode the remainder difference. Four of them are used to store the four differences observed to be most likely, and the remaining two can encode all other possible values of *snlrd*.

The street number high attribute of an address range is similarly divided into block difference (*snhbd*) and remainder difference (*snhrd*). Again, the block difference is found to possess much less entropy. Three functions are used to encode the values of *snhbd*, and six are used for *snhrd*.

Like both street number attributes, the add-on attribute is divided into block (*z4bd*) and remainder difference (*z4rd*). Three functions are used to encode the block

Table 3: Attribute coding example functions.

Function	*Probability*
ZIP Code	
$z5 = z5'$	0.97
$z5 = \text{table}[\text{arg}_{16}]$	0.03
Street Name	
$st = st'$	0.13
$st = \text{table}[\text{arg}_{19}]$	0.87
Number of Ranges	
$nr = 64$	0.15
$nr = 1 + \text{arg}_6$	0.85
Street number low	
$snlbd = 0$	0.84
$snlbd = 1$	0.05
$snlbd = 2 + \text{arg}_{20}$	0.11
$snlrd = 1$	0.33
$snlrd = 0$	0.27
$snlrd = 99$	0.15
$snlrd = 2$	0.02
$snlrd = 3 + \text{arg}_5$	0.15
$snlrd = 35 + \text{arg}_6$	0.08

Function	*Probability*
Street number high	
$snhbd = 0$	0.92
$snhbd = 1$	0.05
$snhbd = 2 + \text{arg}_{20}$	0.03
$snhrd = 98$	0.50
$snhrd = 0$	0.22
$snhrd = 1$	0.12
$snhrd = 99$	0.06
$snhrd = 2 + \text{arg}_5$	0.07
$snhrd = 34 + \text{arg}_6$	0.03
Add-on	
$z4bd = 0$	0.78
$z4bd = 1 + \text{arg}_7$	0.17
$z4bd = -1 - \text{arg}_7$	0.05
$z4rd = 1$	0.24
$z4rd = -1$	0.16
$z4rd = \text{table}[\text{arg}_3]$	0.17
$z4rd = 7 + \text{arg}_5$	0.15
$z4rd = 39 + \text{arg}_6$	0.12
$z4rd = -4 - \text{arg}_5$	0.09
$z4rd = -36 - \text{arg}_6$	0.07

difference – one to be used when the value hasn't changed, one for increases in the value, and one for decreases. The remainder difference has a very high entropy, and a total of seven functions are needed to compress it. The function $z4rd = \text{table}[\text{arg}_3]$ is used to allow one function to represent eight common values. The values stored in the table of this function are -3, -2, 0, 2, 3, 4, 5, 6.

Table 4 shows how the sample records in Table 2 are compressed using the functions in Table 3. The first line of this table repeats the first record that will be compressed. Beneath this, the functions that will be chosen to compress each attribute are shown. The first attribute that is encoded is ZIP Code. Here, there is no previous ZIP Code that we can relate the current ZIP Code to, so we are forced to use the more expensive function, $z5 = \text{table}[\text{arg}_{16}]$. (For the purposes of this example, the actual arguments aren't given. The lowercase letters over the arguments in Table 4 are used to refer to the values of these arguments in Figure 3.) The street name attribute must be similarly coded as a table lookup.

Recall that the low street address number will be stored as the difference from the previous low number. Since this is the first number to be compressed, we take the previous value to be zero. Then the difference to be stored is 100, which has a block difference of 1 and a remainder difference of 0. Both of these are constant functions available to compress the street number low attribute, so these functions are used.

Table 4: Attribute coding example compression.

ZIP Code	*Street name*	*Street address*		*Add-on*
		low	*high*	
14202	MAIN	100	198	3969
$z5 =$	$st =$	$snlbd = 1$	$snhbd = 0$	$z4bd = 1 + \arg_7^c$
$\text{table}[\arg_{16}^a]$	$\text{table}[\arg_{19}^b]$	$snlrd = 0$	$snhrd = 98$	$z4rd = 39 + \arg_6^d$
		101	199	3970
		$snlbd = 0$	$snhbd = 0$	$z4bd = 0$
		$snlrd = 1$	$snhrd = 98$	$z4rd = 1$
		200	399	3950
		$snlbd = 0$	$snhbd = 1$	$z4bd = 0$
		$snlrd = 99$	$snhrd = 99$	$z4rd = -4 - \arg_5^e$
14202	MAPLE	100	199	6110
$z5 = z5'$	$st =$	$snlbd = 1$	$snhbd = 0$	$z4bd = 1 + \arg_7^g$
	$\text{table}[\arg_{19}^f]$	$snlrd = 0$	$snhrd = 99$	$z4rd = 7 + \arg_5^h$
		124	124	9221
		$snlbd = 0$	$snhbd = 0$	$z4bd = 1 + \arg_7^i$
		$snlrd = 3 + \arg_5^j$	$snhrd = 0$	$z4rd = 7 + \arg_5^k$

In this record, the street number high is 98 greater than the low, so a block difference of 0 and a remainder difference of 98 must be stored. Again, these are both relatively frequent occurrences, so there are functions available for each.

Finally, the add-on is also stored as a block difference and a remainder difference from a previous value assumed to be 0. The function $z4bd = 1 + \arg_7$ is used to store the block difference of 39, and the function $z4rd = 39 + \arg_6$ is used to store the remainder difference of 69.

The second and third address ranges of this ZIP+4 record are compressed in a similar manner.

Since the second record to be compressed, has the same ZIP Code as the previous record, we are able to use the less expensive ZIP Code function, $z5 = z5'$. The compression functions used for the other fields are shown in Table 4.

Finally, once functions are chosen for each field, we must write the records out in the format shown in Figure 2. Assume that SFT_i denotes the Huffman-coded form of the street function tuple for the i^{th} source record. Similarly, let RFT_{ij} denote the Huffman-coded form of the range function tuple for the j^{th} address range in the i^{th} record. Then the bit stream that would be written out for the example records is shown in Figure 3. (The lower case letters refer to specific function arguments shown in Table 4).

4 Inverted Index Compression

As have other authors before us ([1, 8]), we have elected to view the lists of pointers stored in the inverted indices as bit vectors. Each list of pointers is transformed into

$SFT_1\ a\ b\ RFT_{11}\ c\ d\ RFT_{12}\ RFT_{13}\ e\ SFT_2\ f\ RFT_{21}\ g\ h\ RFT_{22}\ i\ j$

Figure 3: Output bits produced by compression of sample records.

a bit vector whose length is the number of records in the data base. A bit in the vector is set to 1 if the corresponding pointer is present in the list. Otherwise, the bit is set to 0. Treating the pointer lists as bit vectors allows us to directly apply effective bit vector compression techniques that have been developed previously.

Perhaps the best know bit vector compression method is that of run-length encoding, which was presented by Golomb in [5] in 1966. Golomb's method of run-length encoding compresses sparse bit vectors by encoding the number of 0-bits between consecutive 1-bits. To do this, run-length encoding requires a parameter m, which is chosen based on p, the probability that a given bit in the vector is 0. The run-length encoding method works by replacing m consecutive 0-bits on the input with a single 1-bit on the output. When fewer than m 0-bits remain until the next one bit, a 0-bit is written to the output as a separator, and the number of remaining 0-bits is written to the output using $log_2 m$ bits.

Golomb proves that this method of compressing bit vectors is optimal if p is such that m satisfying $p^m = \frac{1}{2}$ is an integer. Later, in [4], Galagher and Van Voorhis proved that run-length encoding is optimal for p and m satisfying

$$p^m + p^{m+1} \leq 1 < p^m + p^{m-1}. \tag{1}$$

In fact, for any p with $0 < p < 1$, $m = \lfloor -\log_p(1+p) \rfloor$ gives the unique value of m satisfying (1).

While the run-length encoding method can be proven optimal for all cases where the bit vector follows a geometric distribution, it is rare to find such a distribution in an actual application. This was noted before by Teuhola, in [10]. Where a modified version of run-length encoding, called exponential run-length encodings, was presented that tries to take advantage of the clustered nature of most bit vectors. It does this by having the number of 0-bits for which it is looking grow exponentially as more 0-bits are found. This allows the parameter to be small for brief runs of 0-bits, but as the runs get longer, the parameter grows, allowing the capture of more 0-bits with each 1-bit in the output.

Exponential run-length encoding also depends on a parameter, m, which Teuhola recommends choosing so that p^{2^m} is nearly $\frac{1}{2}$. The algorithm that exponential run-length encoding uses for storing a run-length of N consecutive 0-bits follows.

1. Determine $n \geq k-1$ such that

$$\sum_{i=k}^{n} 2^i \leq N < \sum_{i=k}^{n+1} 2^i.$$

2. Output $n-k+1$ 1-bits.

0100011111110010	(a)
0100000000000010	(b)
0000010000010000	(c)

Figure 4: Example of vector splitting. (a) original vector (b) isolated one bits (c) endpoints of sequences of ones

3. Output $N - \sum_{i=k}^{n} 2^i$ as a binary number using $n + 2$ bits.

In addition, we have found that in the ZIP+4 database, as well as other full text retrieval systems, it is likely for references for a specific value to cluster rather closely. For example, in the ZIP+4 database, all records for a given ZIP Code are adjacent. This is an even more extreme example of where the actual distribution of bits in the vector is not at all modelled well by a geometric distribution.

To take advantage of this extreme clustering, we have modified the run-length encoding and exponential run-length encoding by adding a "preprocessing" step to them. In this initial step, a single bit vector is split into two vectors, one that has bits set only if they were the endpoints of runs of consecutive one-bits in the original vector. The other new vector contains only those one-bits from the original vector that were isolated. Figure 4 gives an example of this process.

After this splitting is performed, each of the resultant vectors is compressed by one of the previous methods, and the compressed vectors are stored. To retrieve the original vector, the two compressed vectors are decompressed, and the two vectors are merged, with runs of one-bits being filled in from the vector containing only the endpoints.

We define C to be the compression performance of a given algorithm as

$$C = \frac{\text{Number of input bits}}{\text{Number of output bits}}.$$

For run-length encoding, C can be computed to be ([10])

$$C_{\text{rle}}(p) = \frac{1}{(1-p)(\log_2 m + \frac{1}{1-p^m})} \tag{2}$$

where $m = \lfloor -\log_p(1+p) \rfloor$.

To compute C for split run-length encoding, we reason in the following manner. If we let p be the probability that a bit in the vector is 0, and q be the conditional probability that a bit in the vector is 0, given that the previous bit was a one, we can also compute the theoretical compression gain of the split run-length encoding method. Start with a bit vector of length N having n ones. Consider this vector to be composed of runs of ones. The average length of one of these runs is $\frac{1}{q}$, since they obey a geometric distribution. Therefore, the entire vector is composed of $\frac{n}{1/q} = qn$ runs. We split this vector into two new vectors. The first is composed entirely of ones that appeared singly in the input. The probability that a given run of ones

Table 5: C_{srle} for some values of p and q.

$p = 0.8$		$p = 0.9$		$p = 0.99$	
q	$C_{\text{srle}}(p,q)$	q	$C_{\text{srle}}(p,q)$	q	$C_{\text{srle}}(p,q)$
0.2	4.84	0.2	8.23	0.2	55.06
0.3	3.64	0.3	6.12	0.3	40.20
0.4	3.01	0.4	5.02	0.4	32.66
0.5	2.63	0.5	4.38	0.5	28.23
0.6	2.40	0.6	3.97	0.6	25.47
0.7	2.26	0.7	3.73	0.7	23.78
0.8	2.21	0.8	3.63	0.8	22.92
0.9	2.29	0.9	3.71	0.9	22.94

in the vector consists of a single bit is q. Therefore, the first new vector will be of length N, and contain q^2n bits. The probability that a given bit in this vector is 0 is $1 - \frac{q^2n}{N} = 1 - q^2(1-p)$.

The second new vector contains the endpoints of runs of more than one one bit. Given a run in the original vector, the probability that its length is greater than one is $1 - q$. Therefore, this new vector, also of length N, contains $2(1-q)qn$ one bits. The probability that a given bit in this vector is 0 is $1 - \frac{2q(1-q)n}{N} = 1 - 2q(1-q)(1-p)$.

Finally, we can express the performance of the algorithm in terms of C_{rle} as follows:

$$C_{\text{srle}}(p,q) = \frac{2N}{N/C_{\text{rle}}(1-q^2(1-p)) + N/C_{\text{rle}}(1-2q(1-q)(1-p))} \quad (3)$$

$$= \frac{2}{1/C_{\text{rle}}(1-q^2(1-p)) + 1/C_{\text{rle}}(1-2q(1-q)(1-p))} \quad (4)$$

Table 5 shows the value of C_{srle} for values of q from 0.2 to 0.9 for $p = 0.8, 0.9$, and 0.99. Note that $C_{\text{rle}}(0.8) = 1.32$, $C_{\text{rle}}(0.9) = 2.12$, and $C_{\text{rle}}(0.99) = 12.33$.

5 Results

The database as described has been implemented for the national ZIP+4 file. It allows records to be retrieved by specifying values for any subset of their attributes. The system runs on Sun 4/630, and currently requires an average of 150ms to answer a query.

Attribute coding is able to reduce the size of the national ZIP+4 file from 2,200 MB to 89 MB, a compression ratio of nearly 25 to 1. A total of 86 different functions were designed to compress the 17 attributes of this file. Table 6 shows the storage needed for different types of attributes. Columns two and three of this table present the size in bits per source record for each of the attributes listed in the first column. Those attributes that are of size 0 in the uncompressed file are synthesized attributes used solely for more efficient compression.

The methods discussed in Section 4 were applied to a set of inverted indices used to access the national ZIP+4 File. The implementation contains one inverted index

Table 6: Results of attribute coding applied to the national ZIP+4 file.

Attribute	*Size (in bits)*	
	Before	*After*
ZIP Code	40	0.054
Street Information	240	2.84
Number of ranges	0	0.444
Street Function Tuple	0	0.385
Total for Street Information	280	3.723
Record Type	16	0.062
Address Ranges	272	5.324
Add-on	64	4.155
Range Function Tuple	0	11.172
Total for Range Information	352	20.713
Total	632	24.436

Table 7: Results of bit vector compression methods applied to inverted indices.

Attribute	*Size (in MB) using*				
	Before	*RLE*	*ERLE*	*SRLE*	*SERLE*
Street Information	204.7	8.7	8.6	17.1	8.1
Record Type	141.9	19.7	17.1	9.0	8.7
Address Ranges	145.3	56.3	54.9	51.6	50.5
Add-on	148.6	53.8	54.0	53.8	54.0
Total	640.5	138.5	134.6	131.5	121.3

for each attribute. Table 7 presents the performance of each algorithm on these indices. As this table shows, the extra step of splitting bit vectors can in some cases provide substantial savings (the record type index, for example). Also, it seems to be very sensitive to the method used to compress the newly generated bit vectors, as can be seen by the difference in performance between SRLE and SERLE on the street information indices.

The overall directory system compresses the ZIP+4 data using attribute coding and the access structures using SERLE compression. The compressed source data requires 89MB of storage, and the access structures require 121MB. Additionally, the software that runs the directory system needs a work space of 25MB. Therefore, the entire system occupies 235MB of memory.

6 Conclusion

This paper has presented methods for access to and compression of large, read-only, structured databases. The access methods are general enough to permit retrieval of records based on any or all of the attributes of the database.

Attribute coding was demonstrated as an effective method for compressing a struc-

tured database. It is easily capable of achieving large compression ratios with redundant data.

A variety of methods for bit vector compression were presented. A new approach involving some preprocessing of the uncompressed vector shows promise. More work is needed in this area.

7 Acknowledgments

We thank Chris Curtis and Chandra Bharathi of CEDAR for their assistance on this project. Thanks also go to Glenn Davis of Arthur D. Little, who provided valuable discussions of attribute coding and the database problem in general.

This work is supported by the Office of Advanced Technology of the United States Postal Service under task order number 104230-88D-2576.

References

[1] A. Bookstein and S. Klein. Flexible compression for bitmap sets. In *Proc. IEEE Data Compression Conference*, pages 402–410, Snowbird, Utah, April, 1991.

[2] P. B. Cullen, J. J. Hull, and S. N. Srihari. A constraint satisfaction approach to the resolution of uncertainty in image interpretation. In *Proc. of the Eighth Conference on Artificial Intelligence for Applications*, pages 127–133, March, 1992.

[3] G. Davis. Method and system for storing and retrieving compressed data. US Patent no. 4,868,570, September 1989.

[4] R. G. Gallager and D. C. V. Voorhis. Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory*, IT-21(2):228–230, March, 1975.

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

[6] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IERE 40*, pages 1098–1101, 1952.

[7] A. K. Mackworth. Constraint satisfaction. In *Shapiro S. C. (ed.), Encyclopedia of Artificial Intelligence*, volume 1, 2nd Edition, pages 205–211, 1990. John Wiley & Sons, New York.

[8] A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In *Proc. IEEE Data Compression Conference*, pages 72–81, Snowbird, Utah, 1992.

[9] Technology Resource Center, Arthur D. Little, Inc. Specification of directory modifications. Technical Report Task Order 90-08, Contract No. 104230-83-D-1902, United States Postal Service, December 1990.

[10] J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, 7(6):308–311, October 1978.

[11] I. Witten, T. Bell, and C. Nevill. Models for compression in full-text retrieval systems. In *Proc. IEEE Data Compression Conference*, pages 23–32, Snowbird, Utah, 1991.