Division of Research
Graduate School of Business Administration
The University of Michigan

March 1981

A PRACTITIONER'S GUIDE TO DATA COMPRESSION

Working Paper No. 255

Dennis G. Severance

The University of Michigan

FOR DISCUSSION PURPOSES ONLY

None of this material is to be quoted or
reproduced without the express permission
of the Division of Research.

A Practitioner's Guide to Data Compression

Dennis G. Severance
The University of Michigan
Ann Arbor, Michigan 48109

ABSTRACT

Data compression techniques can substantially reduce both secondary storage cost and data maintenance and retrieval time by decreasing the physical size of a database by as much as ninety percent. This paper is written to provide assistance to practitioners considering the use of data compression on commercial databases. It distills a large collection of literature on data compression developed over the past thirty years and extracts from it facts and guidelines to assist system designers in evaluating the costs and benefits of compression and in selecting techniques appropriate for their needs.

Key Words and Phrases: data compaction, file compression, data encoding, null suppression, differential encoding, comparison of alternatives

CR Categories: 3.7, 3.73, 4.33, 4.4, 5.6

OUTLINE

1. INTRODUCTION.................................... 1

2. COMMON FORMS OF REDUNDANCY AND BENEFITS OF COMPRESSION.................................... 2

3. FUNDAMENTAL CONCEPTS, TERMINOLOGY AND THEORY OF COMPRESSION.................................... 5

4. COMPARISON OF COMMON COMPRESSION METHODS...... 12

   4.1 Encoding Methods for Character Strings.................................... 12

   4.2 Null Suppression for Character Strings.................................... 25

   4.3 Differential Coding Applied to Character Strings.................................... 30

   4.4 Compression Enhancement with Formatted Data Records.............................. 31

5. SUMMARY.................................... 35

REFERENCES.................................... 38

1.0 INTRODUCTION

Compression techniques are consistently found to reduce secondary storage requirements in commercial databases by 30 to 70 percent [INFO75]. It is surprising, therefore, that these techniques are not more widely used. Three facts help to explain this phenomenon:

1) Data compression adds a layer of complexity to the design, implementation, and operation of an information system and designers are reluctant to accept additional complexity without clear and substantial benefits.

2) Designers typically underestimate the amount of data compression which is possible for a given database, and the implications of this compression are not fully appreciated.

3) Much of the published literature narrowly addresses individual compression techniques, surrounding them in a mathematical mystique. Designers understandably avoid areas in which they feel uncomfortable.

This paper distills a rich collection of literature on data compression [ARON77, DAVI76, VILL71] and extracts from it facts which will assist system designers in evaluating the costs and benefits of compression and in selecting a technique appropriate for their needs. Specifically, it defines the nature of data redundancy; explains the compression techniques designed to reduce it; considers advantages and disadvantages of each technique; and provides an index into literature relevant to a practitioner.

2.0 COMMON FORMS OF REDUNDANCY AND BENEFITS OF COMPRESSION

Data is stored in computer memories as patterns of binary digits. Redundancy and the associated opportunity for compression arise when portions of the stored patterns carry little or no "information" and are so predictable that they can be deleted or replaced by significantly shorter patterns. In general, redundancy exists in databases in one of three forms:

- Data values range over a domain much smaller than can be represented with their storage format.
- One or more data values occur with exceptionally high frequency.
- Significant correlation exists between successive data values.

Although some forms of redundancy such as parity bits and check digits [BERL74, PETE72] are valuable in preserving data integrity, most redundancy serves no useful purpose. Depending upon the amount of unneeded redundancy, compression can reduce the database storage requirements by as much as 98 percent.

Several forms of inefficient redundancy are obvious. Alphanumeric data stored in the form of popular 8-bit codes can be compressed by 37 percent by recoding with a 5-bit code [MART75]. Source programs stored in eighty-column card format can be compressed by 50 to 78 percent by removing blank characters [FAJM73, HAHN74, MULF71]. Successive telemetry readings generally vary slowly, and representing

them as a difference from previous readings can reduce transmission requirements by as much as 98 percent [MYER66].

Subtle forms of redundancy also offer a potential for substantial compression. The representation of commonly occurring database characters with short, variable-length codes can reduce space requirements by 47 percent [GILB58]. Coding of character combinations in textual data has reduced file size by 75 percent [HEAPS72, RUBI76]. Since the average "information" contained in a character of English text is estimated to be only a single bit [BELL53], the potential for compression of English words stored in the form of an 8-bit character code is nearly 90 percent!

Database compression yields several benefits. Obviously, the amount and cost of secondary storage are reduced. In addition, database access and transfer speeds can be increased while main memory buffer requirements are reduced. File scanning, merging, and sorting, as well as database backup and recovery operations, can be performed more rapidly, and the collection of applications which are feasible in a constrained storage environment is enlarged.

There are, as well, disadvantages. Additional processing time is required by compression and decompression operations. The most effective compression methods often produce variable-length records which are more difficult to store and retrieve. Many compression techniques require bit manipulation, which may be difficult or inefficient to accomplish when higher-level programming languages are used.

Added time for analyses, design, programming and testing are required during new system development. Finally, recurring maintenance of encoding and decoding tables is often required.

3.Ø FUNDAMENTAL CONCEPTS, TERMINOLOGY, AND THEORY OF COMPRESSION

It is important initially to distinguish three related terms.

Data Encoding is a process which maps from a collection of encoding units (i.e., one or more symbols in one data representation) to a collection of code values (i.e., one or more symbols in a second data representation). The relationship between encoding units and their corresponding code values is referred to as a code. If the code mapping is one-to-one then an inverse mapping exists and decoding refers to the reversing process.

Data Compaction is a form of data encoding which reduces data size while preserving all information considered relevant.

Data Compression is a reversible data compaction process.

While these terms are closely related, they are distinct. There are important nonreversible data encodings which yield effective database access keys in situations where compaction, if any, is incidental (e.g., entry/title keys for bibliographic files [NEWM71, LYNC73] or phonetic name keys for customer files [DOLB7Ø, WEID77]). Encryption [GUDE76, DIFF77] is a reversible data encoding technique (designed to obscure the meaning of sensitive data) which

does not generally result in compaction. Abbreviation [BOUR61] is a form of data compaction which is sometimes nonreversible. In this paper we are concerned exclusively with data compression techniques; that is, reversible encodings designed to reduce data storage requirements.

Classical information theory [SHAN48, HUFF52, ASH65] provides concepts and terminology fundamental to the discussion of data compression techniques. It is here that the <u>binary digit</u> or <u>bit</u> is defined as a basic measure of information. Given a database with N encoding units, $\{u_i\}$, the theory provides useful bounds on the amount of compression possible. Specifically, let ℓ_i denote the length in bits of the i-th encoding unit, while p_i denotes its occurrence probability. Since a string of n bits can take on 2^n distinct values, a <u>fixed-length code</u> of length $\hat{\ell}$ can always be constructed for $\hat{\ell}$ equal to the smallest integer greater than or equal to $\log_2 N$ (denoted $\hat{\ell} = \lceil \log_2 N \rceil$). Specifically, one can assign the binary numbers 1 to N to the encoding units and thereby form a code.

The <u>compression ratio</u> of a code is the relative amount of storage saved by encoding. Assuming our database had M total encoding unit occurrences, its original length would be $L = M \sum_{i=1}^{N} \ell_i p_i$. With the fixed-length code above the compressed database would have length $\hat{L} = M \hat{\ell} \sum_{i=1}^{N} p_i = M \lceil \log_2 N \rceil$; and the compression ratio would be

$$(\hat{L}-L)/L = \sum_{i=1}^{N} \left(1-\lceil \log_2 N \rceil / \ell_i \right) p_i .$$

Encoding and decoding with fixed-length codes is straightforward [KNUT73, MAXW73]. Encoding is performed as data are first entered into a database by matching values to be compressed against encoding units held in a main memory encoding table. During this process the replacement code value is either read directly from the code table after the match or calculated as a by-product of the search process (e.g., the resulting value of a "DO-loop" index or binary search trace vector). Decoding can be easily accomplished with an index operation into the original table, now using the code value to calculate the encoding unit's displacement in the table. While decoding requires approximately as much time as is saved through transmission of compressed data, it can often be avoided with fixed length codes by searching for data using encoded values and by designing codes which maintain order relationships for sorting operations.

For illustrative purposes, consider a database consisting of one million occurrences of the ten most popular words in the English language. Suppose that each is terminated by a blank and stored in variable-length, 8-bit character format. Table 1 exhibits an alternative fixed-length 4-bit code and illustrates calculations which show that the original database of 27,336,000 bits can be compressed by 85.4 percent.

While compression by 85.4 percent is extradordinary, further compression can be achieved in our example using a

i	ENCODING UNIT	LENGTH (l_i)	PROBABILITY (P_i)	CODE VALUE
1	THE_	32	.270	0000
2	OF_	24	.170	0001
3	AND_	32	.131	0010
4	TO_	24	.099	0011
5	A_	16	.088	0100
6	IN_	24	.074	0101
7	THAT_	40	.052	0110
8	IS_	24	.043	0111
9	IT_	24	.040	1000
10	ON_	24	.033	1001

$N = 10 \quad M = 1{,}000{,}000 \quad \hat{l} = \lceil \log_2 N \rceil = 4$

$$\text{ORIGINAL LENGTH} = L = M \sum_{i=1}^{N} l_i \, P_i = 27{,}336{,}000 \text{ BITS}$$

$$\text{COMPRESSED LENGTH} = \hat{L} = M \sum_{i=1}^{N} \hat{l} \, P_i = 4{,}000{,}000 \text{ BITS}$$

$$\text{COMPRESSION RATIO} = (L-\hat{L})/L = 85.4 \text{ PERCENT}$$

TABLE 1. A SMALL EXAMPLE OF A FIXED-LENGTH CODE

variable length code. As with international Morse Code [GILB69], the design objective is to assign short code values to frequently occurring encoding units and longer values to the infrequent ones. Many codes are possible and information theory provides assistance both in calculating the maximum compression which can be achieved and in selecting an efficient code. The theory establishes that for any code, the average code value length in a compressed database cannot be less than $I = \sum_{i=1}^{N} p_i \log_2 (1/p_i)$. I is referred to as the entropy of the database and takes on a maximum value of $\log_2 N$ when the occurrence possibility for all encoding units is equal to 1/N. (In this event a fixed length code of length I is in fact optimal.) For our examples I = 3.01 and therefore the best code we may devise will have a compression ratio no greater than (8-3.01)/8, or 89 percent. A variable length code will therefore increase compression here by at most 3.6 percent over the simple 4-bit code.

Whether or not it is useful in a real design situation, the most efficient variable length code is easily found. Specifically, Huffman codes [HUFF52, KNUT73, MART75] have been shown to yield minimum redundancy. One can construct a Huffman Code for any problem by building a binary tree, as follows [SCHW64]. Initially, encoding units are listed in order of their probability of occurrence. The two units with smallest probabilities are removed from the list; a 0-branch is assigned to one and a 1-branch to the other. Their

probabilities are added and assigned to a new combined unit which is merged back into the diminished list so that it is again in order. The procedure is repeated until a single unit remains as the root of the binary tree just constructed. The code value for any original encoding unit can now be read by traversing the path from this root to that encoding unit.

When the procedure is applied to our example, the binary tree shown in Figure 1 results. The Huffman code defined by this tree is shown in Table 2. Its expected code length of 3.05 is slightly greater than the limiting value I = 3.01.[1] Observe that for this code, as for Huffman codes in general, no code value is the prefix of another, and thus every encoding of a stream of encoding units is uniquely decipherable; that is, given an encoded database, one can always distinguish successive code values without resorting to delimiters or length codes. Note, however, that with a database so encoded, the loss of a single bit may affect the decoding of all subsequent code values, unless decoding is resynchronized in some way [WELL72].

On the basis of the concepts and definitions presented here, the next section will categorize a wide variety of alternative data compression techniques.

---

[1]Generally, one can achieve this limit only by encoding strings of encoding units with longer Huffman codes.

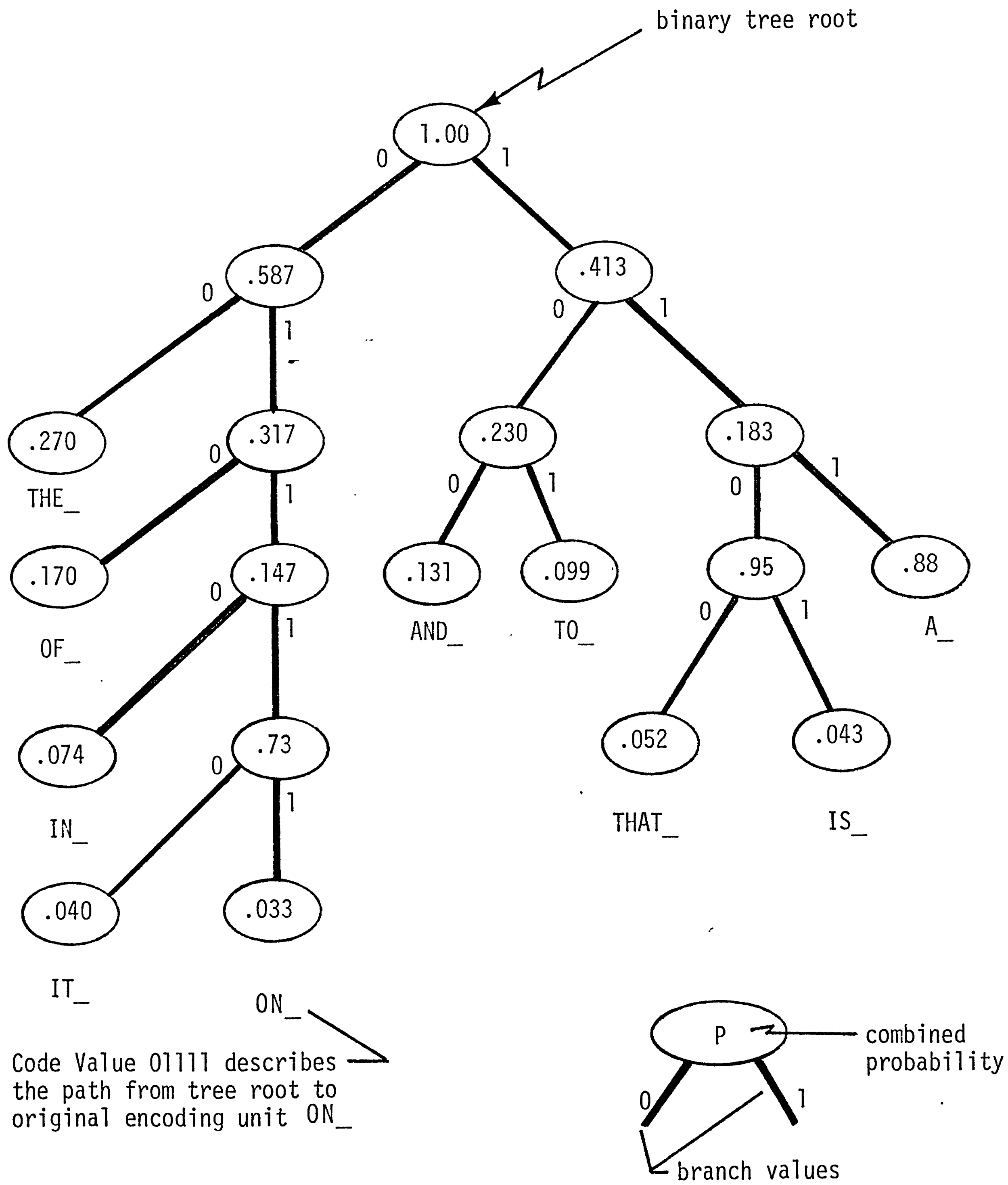


Figure 1. A Huffman Code Binary Tree For The Small Example

i	ENCODING UNIT	PROBABILITY (P_i)	CODE VALUE	LENGTH ($\hat{l}_i$)
1	THE_	.270	00	2
2	OF_	.170	010	3
3	AND_	.131	100	3
4	TO_	.099	101	3
5	A_	.088	111	3
6	IN_	.074	0110	4
7	THAT_	.052	1100	4
8	IS_	.043	1101	4
9	IT_	.040	01110	5
10	ON_	.033	01111	5

N = 10   M = 1,000,000

EXPECTED CODE LENGTH = $\sum_{i=1}^{N} P_i \hat{l}_i$ = 3.05 BITS

COMPRESSED DATABASE LENGTH = $\hat{L} = M \sum_{i=1}^{N} P_i \hat{l}_i$ = 3,053,000 BITS

ORIGINAL DATABASE LENGTH = L = 27,336,000 BITS

COMPRESSION RATIO = $(L-\hat{L})/L$ = 88.8 PERCENT

TABLE 2 A MINIMUM REDUNDACY HUFFMAN CODE FOR THE SMALL EXAMPLE

4.Ø COMPARISON OF COMMON COMPRESSION METHODS

It is easiest to grasp and compare the variety of compression methods which have been reported in published literature by dividing them into two basic categories. In the subsections which follow, we will first describe compression techniques which view a database as a single homogeneous sequence of characters and which reduce its size by either string encoding, null suppression or value differencing. We will then consider compression improvements which are possible with these same techniques when a database is analyzed more carefully as a collection of formatted records whose data items are related via common value domains.

4.1 Encoding Methods for Character Strings

Databases encoded as a stream of characters can be subdivided into substrings of arbitrary length. In general, encoding and decoding operations for short strings are faster and require less main memory for tables and algorithms, while codes for longer strings offer greater compression ratios. Since techniques which encode single characters are simplest, we analyze them first.

A fixed-length code of 6 bits is sufficient to represent a collection of up to sixty-four alphabetic, numeric, and special characters. Studying the popular 8-bit EBCDIC code shown in Figure 2, observe that all code values for uppercase, alphanumeric data begin with the bit

Bit positions 4, 5, 6 and 7 \ Bit positions 0 and 1	00				01				10				11			
Bit positions 2 and 3	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
0000					b̸	&	-					~	{	}	‡	0
0001							/		a	j	s		A	J		1
0010									b	k	t		B	K	S	2
0011									c	l	u		C	L	T	3
0100									d	m	v		D	M	U	4
0101									e	n	w		E	N	V	5
0110									f	o	x		F	O	W	6
0111									g	p	y		G	P	X	7
1000									h	q	z		H	Q	Y	8
1001									i	r			I	R	Z	9
1010					¢	!		:								
1011					.	$	,	#								
1100					<	*	%	@								
1101					(	)	_	'								
1110					+	;	>	=								
1111					|	¬	?	"								

Figure 2 The Extended Binary Coded Decimal Interchange Code - EDCDIC

combination "11." Removal of these bits from four successive characters permits their storage in 24 bits or three character positions. The shifting of bits required to accomplish such compression and decompression is easily programmed and inexpensively executed, and yields a storage reduction of 25 percent. If data are strictly alphabetic, then 5 bits are sufficient to represent a character; eight characters can be stored in five character positions; and 37 percent compression is achieved [MART75]. Numeric data affords an opportunity for 50 percent compression [CHEN75, SMIT75], and some computer systems provide hardware instructions to accomplish such encoding and decoding rapidly [IBM71b].

Maximum compression of character strings is achieved with variable-length codes. Heaps [HEAP72] describes a string compression method for characters in common English text which assigns 3-bit codes to the most frequent seven characters and 7-bit codes to the remainder. A prefix bit is appended to each value to distinguish code types. Since the seven most frequent characters account for 65 percent of all occurrences [REZA61], the expected code length for a textual database is 5.4 bits (= 4 X .65 + 8 X .35), 32.5 percent less than an 8-bit code.

This compression can be improved by packing more of the frequent characters into 4 bits in the following manner: let the code values 0 through 12 be the table displacements for the thirteen most frequently occurring characters while the

code values 13, 14, and 15 designate three tables for characters which are not members of the first thirteen. If the first 4 bits have the value 13, 14, or 15, the next 4 bits give a character displacement in three corresponding tables. Since the thirteen most frequently occurring characters comprise about 80 percent of all occurrences in English text, representing them with 4 bits reduces the average code length to 4.8 bits per character, for a 40 percent reduction.

Huffman coding of single characters with the occurrence frequencies of common English text produces the code shown in Figure 3 [GILB59]. Its expected code length of 4.12 bits yields a 48 percent compression. Martin [MART75] provides the Huffman code for a commercial database whose character frequency distribution is shown in Figure 4. Expected code length here is 2.91 bits, for compression of nearly 64 percent.

Gottlieb [GOTT75] reports the use of Huffman coding on a variety of large insurance files that already had some numeric data in compact binary form. The minimum compression that resulted was 50 percent. Ruth and Kreutzer [RUTH72] applied the Huffman coding to a 350-million-character Requisition Status File used at Navy Inventory Control Points and found its performance to be "unacceptable." However, by extending the set of character encoding units to include twelve commonly occurring character patterns, a 61 percent compression was achieved. This was the best

LETTER	PROBABILITY	CODE VALUE
	0.1859	000
A	0.0642	0100
B	0.0127	011111
C	0.0218	11111
D	0.0317	01011
E	0.1031	101
F	0.0208	001100
G	0.0152	011101
H	0.0467	1110
I	0.0575	1000
J	0.0008	0111001110
K	0.0049	01110010
L	0.0321	01010
M	0.0198	001101
N	0.0574	1001
O	0.0632	0110
P	0.0152	011110
Q	0.0008	0111001101
R	0.0484	1101
S	0.0514	1100
T	0.0796	0010
U	0.0228	11110
V	0.0083	0111000
W	0.0175	001110
X	0.0013	0111001100
Y	0.0164	001111
Z	0.0005	0111001111

Figure 3. A Huffman Code For Characters In Common English Text

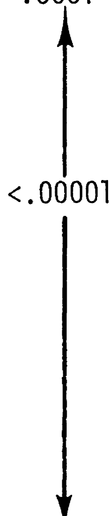
ENCODING UNTI	PROBABILITY	CODE VALUE
0	.555	0
1	.067	1000
2	.045	1100
8	.035	10010
3	.033	10100
A	.032	10101
5	.030	10110
6	.027	11100
4	.027	11101
9	.022	11110
7	.019	100110
F	.015	101110
B	.012	111110
Blank	.011	110110
D	.010	110100
E	.009	110101
Z	.007	1011110
P	.006	1111110
N	.005	1101110
U	.004	10011110
C	.004	10011100
H	.004	10011101
R	.003	10111110
M	.003	11111110
L	.003	11111111
S	.0025	11011110
I	.0020	100111110
T	.0015	110111110
K	.0015	110111111
Y	.0013	1001111110
X	.0012	1001111111
G	.0010	1011111100
J	.0010	1011111101
O	.0006	10111111100
Q	.0003	10111111101
V	.0003	10111111110
W	.0003	101111111110
.	.0001	1011111111110000
-		1011111111110001
?		1011111111110010
&		1011111111110011
/		1011111111110100
+		1011111111110101
<	 <.00001	1011111111110110
)		1011111111110111
(		1011111111111000
%		1011111111111001
=		1011111111111010
#		1011111111111011
?		1011111111111100
'		1011111111111101
@		1011111111111110
		1011111111111111

Figure 4. Huffman Code For A Specific Commercial Database

performance of twelve alternative codes evaluated. McCarthy [MCCA73] presents a systematic approach for selecting multicharacter encoding units for a Huffman code, and reports the compression achieved on a variety of files: 40 percent compression on an object-module file, 57 percent on an English text file, 69 percent on a name-and-address file, and 82 percent on a COBOL source file.

One sees from this data that the encoding of strings with multiple characters results in greater compression. The construction of codes for encoding units with N characters is referred to generically as N-gram encoding. Snyderman and Hunt [SNYD70], in an early application of digram encoding [BOOK76] (where N=2), take advantage of the fact that most of the data stored with an 8-bit code uses no more than 88 of the 256 possible code values. Thus, 168 code values are available to represent pairs of characters. The specific digrams selected experimentally by those authors, together with their code values (in a hexidecimal format), are shown in Table 3. Also shown are eight "master" characters and twenty-one "combining" characters from which the 168 (=8*21) digrams are derived, as follows:

> If the initial character in a string to be compressed is not a master character then it is passed unchanged; else if the next string character is not a "combining" character, then again the first character is passed unchanged; else both characters are passed with the encoded value formed by adding the code value component of the master character to that of the combining character. The process repeats with the remaining string. It is uniquely reversible.

Master Characters		Combining Characters		Noncombining Characters						Combined Pairs			
Symbol	Base Value	Symbol	Hex Code	Symbol	Hex Code	Symbol	Hex Code	Symbol	Hex Code	Symbol	Hex Code	Symbol	Hex Code
ƀ	58	ƀ	Ø0	J	15	q	2B	<	41	ƀb	58	Aƀ	6D
A	6D	A	Ø1	K	16	r	2C	(	42	ƀA	59	AA	6E
E	82	B	Ø2	Q	17	s	2D	+	43	ƀB	5A	AB	6F
I	97	C	Ø3	X	18	t	2E	&	44	ƀC	5B	AC	70
O	AC	D	Ø4	Y	19	u	2F	!	45	ƀD	5C	↓	↓
N	C1	E	Ø5	Z	1A	v	30	$	46	ƀE	5D	AW	81
T	D6	F	Ø6	a	1B	w	31	*	47	ƀF	5E	Eƀ	82
U	EB	G	Ø7	b	1C	x	32	)	48	ƀG	5F	EA	88
		H	Ø8	c	1D	y	33	;	49	ƀH	6Ø	↓	↓
		I	Ø9	d	1E	z	34	-	4A	ƀI	61	EW	96
		L	ØA	e	1F	Ø	35	/	4B	ƀL	62	Iƀ	97
		M	ØB	f	20	1	36	,	4C	ƀM	63	↓	↓
		N	ØC	g	21	2	37	%	4D	ƀN	64	Oƀ	AC
		O	ØD	h	22	3	38	_	4E	ƀO	65	↓	↓
		P	ØE	i	23	4	39	>	4F	ƀP	66	Nƀ	C1
		R	ØF	j	24	5	3A	?	50	ƀR	67	↓	↓
		S	10	k	25	6	3B	:	51	ƀS	68	Tƀ	D6
		T	11	l	26	7	3C	#	52	ƀT	69	↓	↓
		U	12	m	27	8	3D	@	53	ƀU	6A	Uƀ	EB
		V	13	n	28	9	3E	'	54	ƀV	6B	↓	↓
		W	14	o	29	¢	3F	=	55	ƀW	6C	UW	FF
				p	2A	.	40	"	56				
								<	57				

An Example of Compression

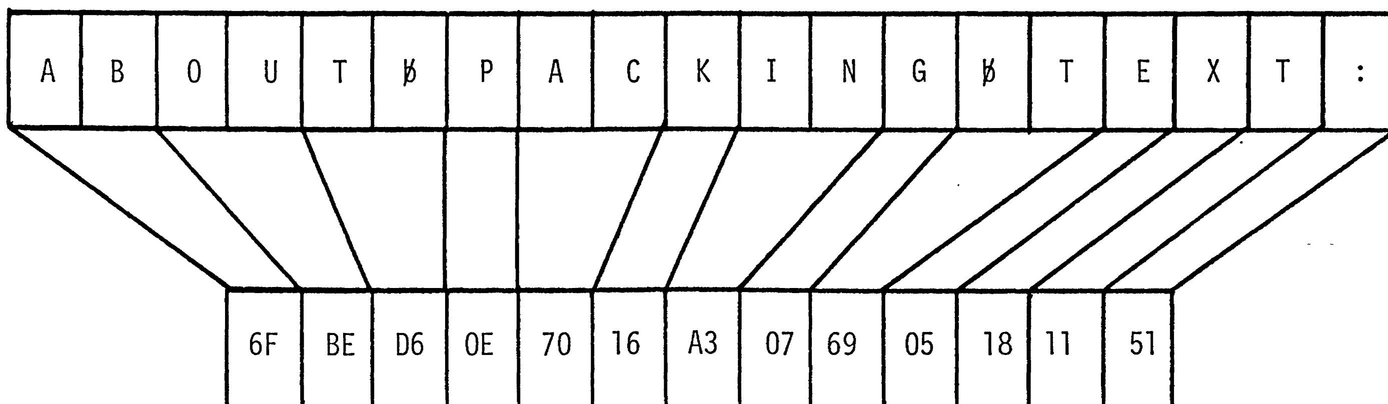


Table 3 DIGRAM ENCODING

Inherently, compression with digram encoding is limited to 5Ø percent, which is achieved only when every character is paired into a digram. Snyderman and Hunt achieved a 35 percent compression, reducing storage requirements of an online file by 6Ø million characters. Schieber and Thomas [SCHI71] used a more systematic method to establish an efficient set of digrams for a 21-million-character database; they achieved a 43.5 percent space reduction.

While these compression ratios are not dramatically better than the 25 percent offered by the simple fixed-length 6-bit code, the method is about as simple to implement. Digrams can be encoded and decoded rapidly via indexing operations, and the required tables are small enough to reside in main memory, in contrast to other encoding methods which may require time-consuming accesses to secondary storage. The method's use of character-length code values also eliminates synchronization problems associated with Huffman codes. Finally, 43 percent compression compares favorably to both Huffman coding of individual characters and compression ratios achieved with more complicated trigram and tetragram encodings [ARON77, SCHW63].

Our example of digram encoding maps two characters into 8 bits. A more general _numerical code [HAGA72]_ maps N characters into K bits. In a manner similar to digram encoding, code values are formed by reversibly combining numerical representations for two or more characters into a

single unique number. Consider, for example, the ten characters A through J. By establishing the correspondence (A=Ø, B=1,...,J=9) with the base-1Ø number system, the data sequence "CAB" can be represented by the number $2Ø1_{1Ø}$ ($=2*1Ø^2+Ø*1Ø^1+1*+1Ø^Ø$). Since the maximum code value in this example is $999_{1Ø}$, a machine with a 1Ø-bit word (with value range $2^{1Ø}=1Ø24$) could store any 3-character sequence in each word. In the same way, if all alphabetics were possible, then "CAB" would become $2Ø1_{26}(2*26^2+Ø*26^1+1*26^Ø)$, or $676_{1Ø}$. In this case, a 15-bit word would be required to hold three characters.

Numerical codes are generally designed to make maximum use of a given computer's word size. Given a computer with a K-bit word, code efficiency is affected by the encoding base, B, and the number M of characters combined. Clearly, as B increases, M decreases. For a machine with K=32, Hahn [HAHN74] evaluates values of B between 14 and 73 and shows that five to eight characters can combine in a single number, giving 2Ø to 5Ø percent compression over an 8-bit code. With B=37 (for alphanumeric data), compression is 2Ø percent and numerical coding is inappropriate, since the simpler 6-bit code offers 25 percent compression.

Hahn improved numerical encoding by incorporating into it a form of variable-length code. He maintains code values in several encoding/decoding tables of size D-1, where D is a value calculated to produce a good compression ratio for a specific machine. The D-1 most frequent characters are

placed in the first table, the next D-1 most frequent are placed in the second table, and so on. Encoding values are M-digit, base-D numbers and representing from 1 to M characters. M is now the maximum number of characters which might be encoded in a single word. Any character being encoded which is not one of the first D-1 symbols in the table is represented by the "escape character," Ø, followed by the character's position in the second table, or so on until the character is located. Obviously, maximum compression is achieved for the first D-1 characters. The more skewed the character occurrence distribution, the smaller the optimum value of D and the larger the average number of characters which can be packed into a single word. For English text on a 32-bit machine, Hahn found that D=21, M=7 was an optimum combination. He combines this code with a null suppression technique (described later) to obtain an average code value length of 4.7 bits per character. This 41 percent reduction again compares favorably with Huffman coding of individual characters.

Heaps [THIE72, HEAP74] has experimented extensively with a similar variable-length, fixed-increment code which attempts to combine the ease of managing fixed-length codes and the maximum compression achieved with variable-length codes. His technique is particularly effective for words or terms in textual data. Here the number of distinct terms is typically quite large and their usage frequency has a Zipfian distribution [ZIPF49, LESK7Ø, WEID77] which is quite

skewed. For this situation, fixed-length codes achieve relatively poor compression and their large encoding/decoding tables must be held in secondary memory where access times are several orders of magnitude greater than those in main memory.

Heaps suggests the following coding scheme to relieve both problems [HEAPS70, TIEL72]. For a vocabulary of N distinct terms, codes of lengths $N_1 < N_2 < \ldots < N_r$ are used to index terms in tables of size $2^{N_1-1}, 2^{N_2-2}, \ldots, 2^{N_r-r}$. The most frequent terms are assigned codes with length N_1. Codes for less frequent terms are built by appending code increments of length $N_2-N_1, \ldots, N_r-N_{r-1}$. The first bit in any increment indicates code continuation; if it is zero, another increment follows. Heaps experimented with different values of $N_1, N_2, \ldots, N_r$, and found that codes with lengths (3,6,9,12,...) and (4,8,12,16,...) perform effectively and were easily adaptable to machines with 6-bit and 8-bit characters, respectively. For large document databases, average code length has been found to be about one-fourth that of a fixed length code and within 8 percent of that of a Huffman code.

Information theory shows that to achieve maximum compression, an encoding unit with occurrence probability p should have length $\log_2(1/p)$. The procedure above attempts to match variable lengths to a given set of frequencies. Alternatively, if one is committed to the use of simple, more manageable fixed length codes, then maximum compression

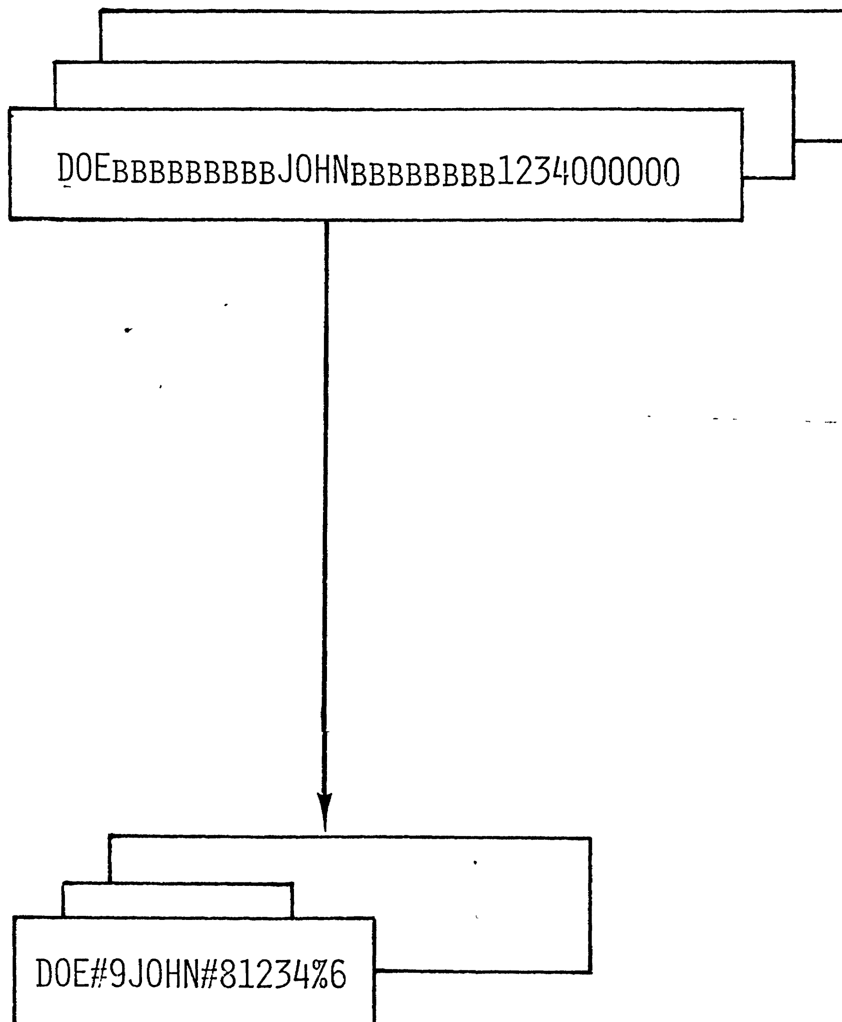
will be achieved only if the encoding units have equal occurrence probabilities. Recognizing this fact, Schuegraf and Heaps [SCHU74] propose a means of generating equifrequent word fragments [CLAR72] to be used as encoding units. A frequency count of all word fragments between two and eight characters in length is taken from the database to be encoded. All fragments having a frequency less than a selected threshold value are eliminated initially and left unchanged by the encoding process, since these infrequent fragments have least effect on total database compression and a substantial effect on the size of the encoding/decoding table required [HEAPS74]. Further elimination of encoding units is accomplished by subtracting the frequencies of longer fragments from the frequencies of shorter fragments contained within them and removing all which fall below the threshold. This favors the longer fragments which contribute most to compression. The number of fragments selected can be controlled with the threshold frequency and adjusted to the amount of main memory available for the encoding/decoding table. For any threshold, the final set of fragments have approximately equal frequencies, so that a fixed-length code is near optimal for the selected set of encoding units. Successful application of the procedure to the Library of Congress MARC tapes [LIBR70] is reported in [SCHU73].

4.2 Null Suppression For Character Strings

For a variety of reasons, commercial databases often contain large numbers of blanks, zeroes, and other filler characters used for the padding of vacant or variable length data fields. With modest processing costs these "null" characters can be compressed to dramatically reduce storage requirements.

A common method of null suppression is the *run length technique* [LYNC73] illustrated by Figure 5. A special character;is inserted to indicate the presence of a run of nulls and the run is replaced by a count indicating its length. Different types of runs can be distinguished with differing special characters. The actual choice depends upon the frequency of character occurrences in the existing code. For example, the EBCDIC code in Figure 2 has a number of characters which may be selected since they are unused in most applications. If no such unused character exists, an infrequently used character can be selected and its occurrence doubled in the encoded stream whenever it appears as a datum. Overbeek and DeMaine [DEMA71, OVER72] report such a use of null suppression to compress a variety of alphanumeric data files by 24 to 61 percent. Hill [HILL75] reports an even more dramatic compression of census files by 63 to 75 percent.

Martin [MART75] describes a run suppression technique for EBCDIC data with no lowercase characters. Here the second code bit is always "1" (see Figure 2) so that a



--- B
% --- 0

Figure 5. NULL SUPPRESSION - Run Length

setting of "Ø" can be used to designate the suppression of nulls. The remaining 7 bits can then represent both the type of null and the length of the run. If, as in Figure 6, for example, only blanks and zeroes are suppressed, then a single bit can distinguish between them, while 6 bits remain to designate run lengths as long as sixty-four characters. Code values here can be one half the length of those in the run length method above.

If only a single null type is suppressed, there is no need to identify it explicitly. In message transmission, for example, it is common to truncate leading and trailing blanks from each fixed-length record. Counts of the number of leading blanks and the number of significant characters in the record are placed at the start of the record, followed by the significant characters of the record [HAHN74]. The technique is especially effective when applied to files containing computer source programs, where leading and trailing blanks are common. Fajman and Bargelt have used the technique quite successfully in the WYLBUR text editor system [FAJM73]. Data stored by WYLBUR is broken into segments consisting of a count-byte split into a 4-bit count of blanks and a 4-bit count of nonblanks, followed by as many as fifteen nonblank characters. These segments can be set up very rapidly through the use of a hardware instruction to seek out runs of blanks and nonblanks. Decoding is accomplished very simply by jumping from segment to segment, inserting blanks where necessary. Compression of

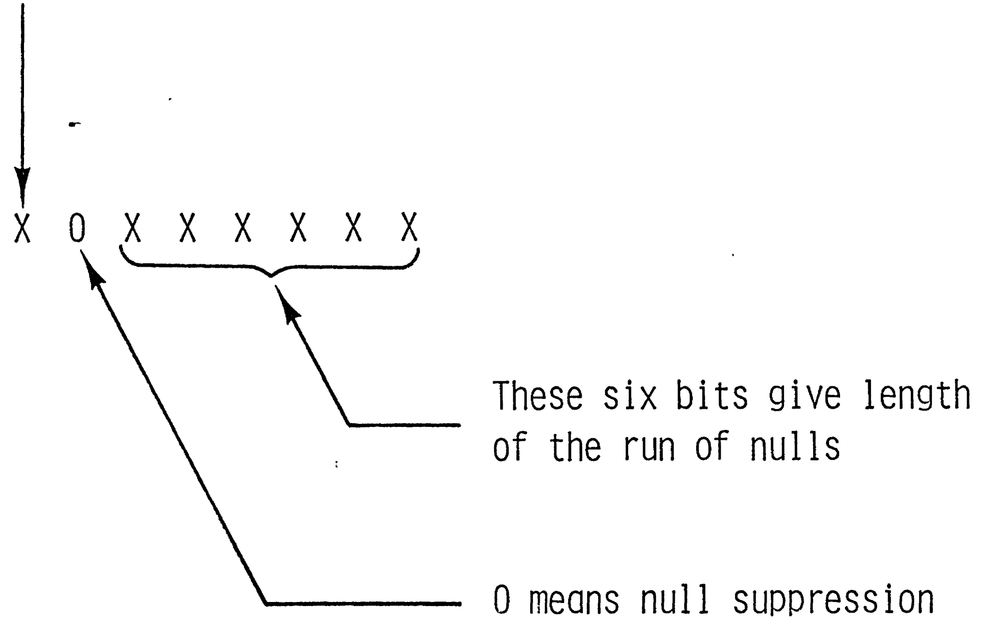


Figure 6. NULL SUPPRESSION WITH EBCDIC

50 to 70 percent has been achieved for many types of text.

Since null suppression techniques are relatively simple and inexpensive in comparison to the more efficient Huffman code, one naturally wonders how much space is actually saved by the Huffman code's additional complexity. For example, Huffman coding compressed the database in Figure 4 by 64 percent; however, the high frequency (55.5 percent) of zeroes suggests that as much as 50 percent compression might be achieved via null suppression alone. Martin [MART75] addressed this question by applying both methods to three commercial databases obtaining the results shown in Table 4. Here Huffman coding provided 12 to 28 percent additional compression which may or may not be significant enough in a particular application to justify its additional complexity.

Original File Size (Bytes)	Reduction Using Suppression of Repeated Characters (%)	Reduction Using Huffman Code (%)
300,000	54	82
3 million	34	46
19 million	64	83

Table 4. Compression Results for Three Commercial Databases

4.3 Differential Coding Applied to Character Strings

Differential coding involves the replacement of an encoding unit with a code value which defines a relationship to either a previous encoding unit or a selected pattern value [GOTT75]. Ruth and Kreutzer [RUTH72] and Villers and Ruth [VILL71] provide a ready index to literature on telemetry compression, which is the single most important application, of differential encoding. In telemetry applications a sensing device records measurements at a remote station for storage and analysis. Because successive readings are of uniform size and tend to vary relatively slowly, they are efficiently represented by their difference from the prior reading. Compression is applied prior to transmission and can reduce the total amount of data transmitted by more than 98 percent [MYER66].

Comparable relationships between successive data values rarely occur in business applications. Knuth [KNUT73] suggests a hypothetical application for which the prime numbers less than one million are required. Rather than storing the 78,498 different values directly, the successive difference between these primes are encoded. Since it can be shown that the difference between any two primes less than 1,357,201 does not exceed 63, these gaps can be encoded as fixed-length 6-bit values, reducing table size by 70 percent.

Date [DATE75] suggests another application in which sorted key values are stored and read sequentially. To

achieve compression, the keys are read in sequence and all leading characters of a key value common to the preceding value are replaced by their count. For example, the series of names JOHNSON, JONAH, JONES, JORGENSON would become (Ø) JOHNSON, (2) NAH, (3) ES, (2) RGENSON. Decoding demands sequential reading so that the preceding key value is once more available. The count of the key value to be decompressed provides the number of leading characters to be retained and the unencoded characters are then concatenated.

4.4 Compression Enhancement with Formatted Database Records

For reasons of process efficiency, commercial databases are typically composed of formatted data records [BENN67, LIU68]. Each data record is subdivided into data items (or fields) with specific boundaries, whose range of values and occurrence distributions often are known. Viewed narrowly, data item values are simply strings of characters and thus the compression methods described in the previous section can be applied directly. However, special opportunities for increased compression are presented since particular data items may take on a relatively narrow range of values over an entire database, and data items often exhibit a recurring intrarecord structure. The price paid to obtain this greater compression is an additional amount of main memory required for encoding and decoding tables now associated with individual data items rather than the entire database.

The amount of compression achieved with the string encoding methods of Section 4.1 is increased with a

formatted database because record data items naturally partition the database into character strings with like value characteristics. Values for a particular data item, for example, may be known a priori to be members of a specific set of alphabetic strings such as surnames, titles, cities, states, or possibly numeric quantities such as rates, dates, telephone numbers, or inventory levels. The specific value set and its occurrence distribution can be readily computed with the aid of a simple file analysis program [BRAY77, INFO75]. This knowledge substantially reduces the total number of value possibilities for character strings of a given length. As a result, the average code value length required to encode these strings can be significantly shorter. Martin [MART75] suggests, for example, that an 8-bit code is sufficient to distinguish 256 surnames, which account for more than 90 percent of all last names used in the United States. For all but the largest organizations, 16 bits (65,536 code values) are ample to distinguish such data as customers, suppliers, employees, or inventory items.

Date fields are frequently present in commercial databases and can be compressed substantially from the common 6-digit (MMDDYY) representation by using differential encoding. Selecting an appropriate date as a base point, compressed dates can be represented as a distance in days from that origin. January 1, 4713 B.C., for example, is the Julian date base point generally used to record astronomical

phenomena, and 22-bit code values can represent dates through the year 6700 A.D. A shorter 16-bit code and a more recent base point can represent dates ranging over a period of 175 years, which is sufficient for most business applications [ALSB75, MART75]. In records where a sequence of related dates is stored, each can be represent as its difference from the previous date with significantly fewer bits [GOTT75].

The compression achieved with null suppression techniques is also typically enhanced when compressing formatted data records. In fact, null suppression is so effective (sometimes achieving 60 to 70 percent compression) that it tends to be the only compression technique routinely supplied with commercial data management systems [CULL77, IBM74a, SOFT75]. A number of methods for compressing null data item values have been described by Olle [OLLE68] and analyzed by Maxwell [MAXW73]. As illustrated in Figure 7, the most common is a bit vector appended to the front of each record in which each bit is associated with a corresponding field. A bit value of 0 indicates presence of the null value which is dropped from the record during compression and reinserted during decompression. The same idea has been applied to groupings of data items in some data management systems (e.g. CICS [IBM74b], IMS [IBM71a]) which associate bits with record segments. Segments are collections of related data items which tend to occur either together or not at all (e.g., several items of data would be

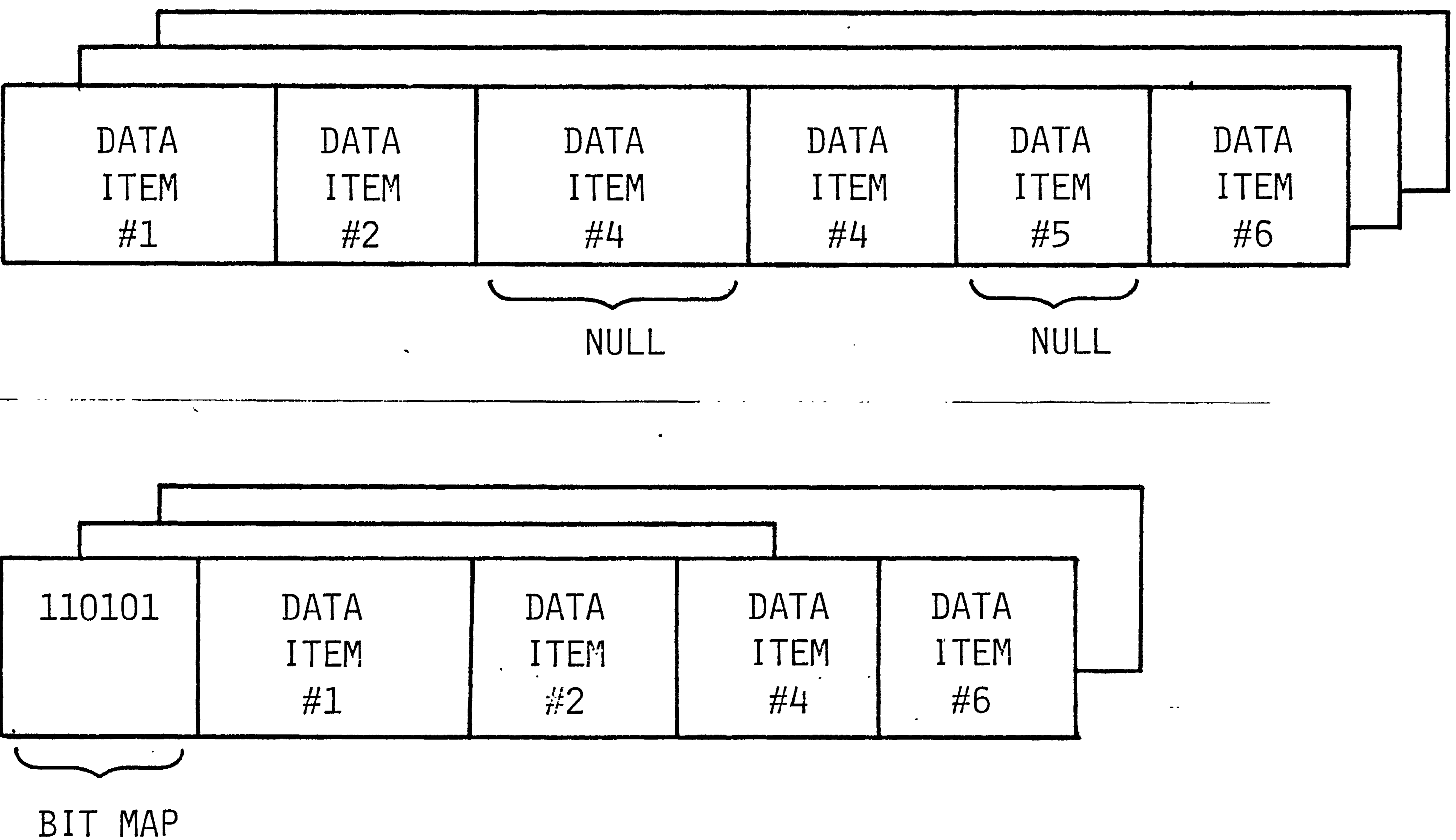


Figure 7. NULL SUPPRESSION - Bit Map

required to described each customer claim that is filed against a particular insurance policy). By suppressing entire record segments rather than their individual data items, both the amount of data compression and speed of decompression improve slightly.

SUMMARY

One should not conclude from this paper that data compression is desirable for all business applications. The benefit associated with compression of a particular database is affected by many variables: the size of the database, the amount and type of redundancy it contains, the nature and frequency of retrieval and update requests serviced, the availability and cost of memory for both code tables and data, the efficiency and complexity of the data compression technique considered, and finally the availability and cost of processor time for execution of these techniques. What is clear nevertheless is that typical commercial databases can in fact be compacted by 30 to 90 percent, and that this should be of more applied interest than current usage of compression techniques would indicate.

In practice one finds that the most common motivation for database compression is a storage constraint which otherwise precludes implementation of a particular application. Since a simple technique which affords the compression "necessary" for the application is often more highly prized than a more complex one which offers the

"best" compression, some rather pragmatic observations summarize this survey.

Four relatively simple compression techniques adequately address all problems of interest to most practitioners. Null suppression is easily implemented and often effective. Compression/decompression routines are commercially available or easily developed. They execute quickly, require neither code tables nor record formats, and achieve compression of 50 to 70 percent on a wide variety of data files. For files which do not compact under null suppression, digram encoding offers similiar advantages, requires a relatively small code table and reliably achieves compression of 30 to 40 percent.

Fixed-length codes are also simple and have the important advantage of providing fixed field and record boundaries. This facilitates direct record access, selective field compression/decompression, and data searching with compressed keys without need for data decompression. Code tables of even a few hundred entries may be held in main memory and searched rapidly. For databases whose encoding units have a uniform occurrence distribution, fixed-length codes achieve near-optimal compression.

In principle, Huffman codes do achieve optimal compression. However, their complexity and synchronization problems tend to make them unattractive in practice. In situations where a skewed encoding unit occurrence distribution makes a Huffman code substantially more

efficient than a fixed-length code, a variable-length fixed-increment code is an effective compromise. Specifically, these codes are relatively simple, self-synchronizing, and easily adapted to a machine of arbitary word length, while they provide a compression ratio which approximates that of a Huffman code.

ACKNOWLEDGEMENTS

I am endebted to Dean Wilder, Pat Lung and Jim McKeen, who have provided valuable assistance in the development and revision of this paper.

REFERENCES

[ALSB75] Alsberg, P.A., "Space and time Savings Through Large Data Base Compression and Dynamic Restructuring," *Proc. IEEE, 63,8 August 1975, pp. 1114-1122*.

[ARON77] Aronson, J., *Data Comparison - A Compression of Methods*, National Bureau of Standards, special publication 500-12, June 1977, 39 pp.

[ASH65] Ash, Robert, *Information Theory*, Interscience, 1965.

[BART74] Barton, I.J., Creasey, S.E., Lynch, M.F., and Snell, M.J., "An Information-theoretic Approach to Text Searching in Direct Access Systems," *Comm. ACM*, 17,6 June 1974, pp. 345-350.

[BELL53] Bello, F., "The Information Theory," *FORTUNE*, December 1953, pp. 136-158.

[BENN67] Benner, F.H., "On Designing Generalized File Records for Management Information Systems," *Proc. FJCC*, 1968, pp. 145-156.

[BERL74] Berlekamp, E.R. (ed.) *Key Papers in the Development of Coding Theory*, IEEE Press, Piscataway, N.J., 1974, 296 p.

[BOOK76] Bookstein, A. and Fouty, G., "A Mathematical Model For Estimating the Effectiveness of Bigram Coding," *Int. Proc. and Manag.* 12, 1976, pp. 111-116.

[BOUR61] Bourne, C.P. and Ford, D.F., "A Study of Methods for Systematically Abbreviating English Words and Names," *Jour. ACM*, 8,3, July 1961, pp. 538-552.

[BRAY77] Bray, O. and Severance, D.G. "Field Encoding Analysis Routine: User's Manual and System Documentation," *MISRC Technical Reports 77-20, 77-21*, GSBA, University of Minnesota, 1977.

[CHEN75] Chen, T.C. and Ho, I.T., "Storage-Efficient Representation of Decimal Data," *Comm. ACM*, 18,1, January 1975, pp. 49-52.

[CLAR72] Clare, A.G., Cook, G.M., and Lynch, M.F., "The Identification of Variable-length, Equi-frequency Character Strings in a Natural Language Data Base," *Computer J.* 15, 1972.

[CULL77] Cullinane Corporation IDMS Concepts and Facilities Order No. DØØ1, Wellesley Mass., 1977.

[DATE75] Date, C.J., An Introduction to Data Base Systems, Addison Wesley, 1975, Chapter 2.4, pp. 34-35.

[DAVI76] Davisson, L.D. and Gray, R.M. (eds.), Data Compression (benchmark papers in Electrical Engineering and Computer Science, Vol. 14), Dowden, Hutchinson, Ross, Inc., Stroudsburg, PA., 1976, 4Ø7 pp.

[DEMA71] DeMaine, P.A.D., "The Integral Family of Reversible Compressors," Journal of the IAG, (IFIPS, Amsterdam), 3, 1971, pp. 2Ø7-219.

[DIFF77] Diffie, W. and Helleiman, M.E., "Exhaustive Cryptanalysis of the NBS Data Encrytion Standard," Computer, IEEE, June 1977, pp. 74.

[DOLB7Ø] Dolby, James L., "An Algorithm for Variable-Length Proper Name Compression, Journal of Library Automation, December 197Ø, pp. 257.

[FAJM73] Fajman, Roger, and Borgelt, John, WYLBUR: An Interactive Text Editor and Remote Job Entry System, Comm. ACM, 16:5, May 1973, p. 314.

[GILB59] Gilbert, E.N., and E.F. Moore, "Variable-Length Binary Encodings," Bell System Tech J., July 1959, pp. 933.

[GOTT75] Gottlieb, D.S.A., Hagerth, P.G., Lehot, H. and H.S. Rubinowtiz, "A Classification of Compression Methods and their Usefulness for a Large Data Processing Center," Proceedings 1975 National Computing Conference, AFIPS, 44, (1955) pp. 453-458.

[GUDE76] Gudes, E., Koch, H.S. and Stahl, F.A., "The Application of Cryptography for Database Security," Proceedings 1976 National Computer Conference, 1976, pp. 97-1Ø7.

[HAGA72] Hagamen, W.D., Linden, D.J., Long, H.S., and Weber, J.C., "Encoding Verbal Information as Unique Numbers," IBM Systems Journal 11,4, 1972, p. 278.

[HAHN74] Hahn, Bruce, "A New Technique for Compression and Storage of Data," Comm. ACM, 17,8, August 1974, p. 434.

[HEAP70] Heaps, H.S. and Thiel, L.H., "Optimization Procedures for Economic Information Retrieval," Information Storage and Retrieval, 6, 1970, pp. 137-153.

[HEAP72] Heaps, H.S., "Storage Analysis of a Compression Coding for Document Data Base," INFOR, Vol. 10, No. 1, February 1972, p. 47-61.

[HEAP74] Heaps, H.S., "Compression of Databases for Information Retrieval and Management Information Systems," Working Paper, Computer Science Department, Sir George Williams University, Montreal.

[HILL75] Hill, G.L., "Maximizing Computer Access to Public Data Files," Proceeding 1975 Computer Science Conference ACM-SIGCSE, 1975.

[HUFF52] Huffman, D.A., "A Method for the Construction of Minimum-Redundancy Codes," Proc. I.R.E., 40,9, September 1952, pp. 1098-1101.

[IBM71a] IBM Corporation, IMS/360, System/Application Design Guide, Program Number 5734-XX6, February 1971.

[IBM71b] IBM Corporation, IBM System/370 Model 165 Functional Characteristics, Technical Newsletter, No. GN22-0401, July 1971.

[IBM74a] IBM Corporation, Utility Reducing Subroutines for Suystem/360/370, Program Number 5798 AZW, 1974.

[IBM74b] IBM Corporation, Customer Informaton Control System/Virtual Storage (CICS/VS) Application Programmer's Reference Manual, SH20-9003-0, January 1974, Palo Alto, California, pp. 402-403.

[INFO75] Informatics, Inc., SHRINK User Reference Manual, PMI Document No. 8616, October 1975.

[KNUT73] Knuth, Donald E., the Art of Computer Programming, Vol. 3, Chapter 6.1, Addison Wesley, 1973, 401 p.

[LESK70] Lesk, M.E., "Compressed Text Storage," Computing Science Technical Report #3, Bell Telephone Laboratories, 1970.

[LIBR70] Library of Congress, A MARC Format: Specifications of Magnetic Tapes Containing Monographic Catalog Records in MARC II Format, Information Systems Office, Washington, D.C. 1970.

[LIU68] Liu, H. "A File Management System for a Large Corporate Information System Data Bank," Proc. FJCC, 1968, pp. 145-156.

[LYNC73] Lynch, M.F., "Compression of Bibliographic Files Using an Adaption of Run-length Coding," Inf. Stor. Retr., 9, 1972, pp. 207-214.

[MART75] Martin, James, Computer Data-base Organization, Chapter 32, Prentice-Hall, 1975, 713 p.

[MAXW73] Maxwell, William L. and Severance, Dennis G., "Comparison of Alternatives for the Representation of Data Items Values in an Information System," Data Base, 5,2, SMIS Special Report No. 2, Winter 1973.

[MCCA73] McCarthy, J.P., "Automatic File Compression," International Computing Symposium 1973, North-Holland Publishing Company, pp. 511-516.

[MULF71] Mulford, J.F. and R.K. Ridall, "Data Compression Techniques for Economic Processing of Large Commercial Files," ACM Proc. Symp. Information Storage and Retrieval, 1971, pp. 207-215.

[MYER66] Myers, W., Townsend, M. and Townsend, T. "Data Compression by Hardware or Software," Datamation, April 1966, pp. 39-43.

[NEWM71] Newman, William L. and Buchinski, Edwin J., "Entry/Title Compression Code Access to Machine Readable Bibliographic Files," Journal of Library Automation, June 1971, p. 2.

[OLLE68] Olle, T.W., "Data Structures and Storage Structures for Generalized File Processing," Proceedings of the FILE 68 International Seminar on File Organization, Copenhagen, 1968, pp. 285-294.

[OVER72] Overbeek, R.A. and DeMaine, P.A.D., The Integral Family of Reversible Compressors, The SOLID System Report 2, Com. Sci. Dept, Pennsylvania State University, 1972, 121 p.

[PETE72] Peterson, W.W. and Weldon, E.J., Error Correcting Codes, MIT Press, 1972, 285 p.

[REZA61] Reza, F.M., An Introduction to Information Theory, Chapter 4, McGraw-Hill, New York, 1961.

[RUBI76] Rubin, Frank, "Experiments in Text File Compression," Comm. ACM, 19, 11, November 1976.

[RUTH72] Ruth, Stephen S. and Kreutzer, Paul J., "Data Compression for Large Business File," Datamation, September 1972, p. 62.

[SCHI71] Schieber, William S. and Thomas, George W., "An Algorithm for the Compaction of Alphanumeric Data," Journal of Library Automation, 4,4, December 1971, pp. 198-206.

[SCHU73] Schuegraf, E.J. and Heaps, H.S., "Selection of Equifrequent Word Fragments for Information Retrieval," Inform. Stor. Retr., Vol. 9, Pergamon Press 1973, pp. 697-711.

[SCHU74] Schuegraf, E.J. and Heaps, H.S., "A Comparison of Algorithms for Data Base Compression by the Use of Fragments as Language Elements," Infor. Stor. Retr., 10, 1974, pp. 309-319.

[SCHW63] Schwartz, E.S., "Dictionary for Minimum Redundancy Encoding," Jour. ACM, 10,4, October 1963, pp. 413-439.

[SCHW64] Schwartz, Eugene S. and Kallick, Bruce, "Generating A Canonical Prefix Encoding," Comm. ACM 7,3, March 1964, p. 166.

[SHAN48] Shannon, C.E., "A Mathematical Theory of Communication," Bell Syst. Tech. J., 1948, 27.

[SMIT75] Smith, A.J., "Comments on a paper by T.C. Chen and I.T. Ho," Comm. ACM 18,8, August 1975, pp. 463.

[SNYD70] Snyderman, M. and Hunt, B., "the Myriad Virtues of Text Compaction," Datamation, December 1970, p. 36.

[SOFT75] Software AG ADABAS Reference Manual, Software AG of North america, Reston, VA. 1975.

[THIE72] Thiel, L.H. and Heaps, H.S., "Program Design For Retrospective Searches on Large Data Base," Inform. Stor. Retr., 8, 1972, pp. 1-20.

[VILL71] Villers, J.J. and Ruth, S.R., Bibliography of Data Compaction and Data Compression Literature with Abstracts, 1971, Government Clearing House Study AD 723525.

[WAGN73] Wagner, Robert A., "Common Phrases and Minimum-Space Text Storage," Comm. ACM, 16, 3, March 1973, pp. 148-152.

[WEID77] Weiderhold, Gio, Database Design, McGraw-Hill, New York, N.Y. (1977) 658 p.

[WELL72] Wells, M., "File Compression Using Variable-Length Encodings," Computer Journal, 15, 4, November 1972, pp. 308-313.

[ZIPF49] Zipf, K.G., Human Behavior and the Principle of Least Effort, An Introduction to Human Ecology, Addison-Wesley, Reding, MA, 1949.