Inform. Systems Vol. 8, No. 1, pp. 51–62, 1983
Printed in Great Britain.

0306-4379/83/010051-12$03.00/0
© 1983 Pergamon Press Ltd.

A PRACTITIONER'S GUIDE TO DATA BASE COMPRESSION

TUTORIAL

DENNIS G. SEVERANCE
The University of Michigan, Ann Arbor, MI 48109, U.S.A.

(*Received* 26 *February* 1982)

Abstract—Data compression techniques can improve information system performance by reducing the 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 for the storage of a commercial database. It reviews a wealth of literature on data compression and presents facts and guidelines which will assist system designers in evaluating the costs and benefits of compression and in selecting techniques appropriate for their needs.

1. INTRODUCTION

Since compression techniques consistently reduce secondary storage requirements in commercial databases by 30–90%[4, 19, 39, 71], it is surprising that they are not more widely used. Three facts help to explain this phenomenon:

(1) 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.

(2) 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.

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

This paper distills a rich collection of literature on data compression and extracts from it facts which will assist database 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. COMMON FORMS OF REDUNDANCY AND BENEFITS OF COMPRESSION

Data values are stored in a database as patterns of binary digits. Redundancy exists when portions of these patterns are predictable and therefore carry little or no "information." Redundancy typically exists in one of three forms:

—One or more data values occur with exceptionally high frequency.

—Significant correlation exists between successive data values.

—Data values range over a domain much smaller than can be represented with their storage format.

While forms of redundancy such as parity bits and check digits[8, 55] serve a planned and useful purpose, most redundancy is unplanned and unnecessary. With no loss of informations space requirements for textual files can be reduced by 47% by representing commonly occurring characters with short variable length codes[24]. Coding of character combinations can reduce file size by 75%[31, 59]. Computer programs stored in eighty-column card format can be compressed by 50–78% by removing blank characters[23, 29, 50]. Representation of successive telemetry readings as differences from previous readings can reduce transmission requirements by as much as 98%[51].

Database compression may yield many benefits. Storage costs and buffer requirements can be reduced while data access and transfer speeds are increased. Telecommunication charges can also be reduced, while effective data transmission speed is increased. Scanning, merging, and sorting, as well as database backup and recovery operations, can be performed more rapidly on smaller files; 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. Some effective compression methods produce variable-length records which are more difficult to store and retrieve. Some compression techniques require bit manipulation, which is often difficult or inefficient to accomplish with higher-level programming languages. Added time for analyses, design, programming and testing are required during new system development. And finally, ongoing maintenance of encoding and decoding tables is sometimes needed as new data values are stored in a database.

3. FUNDAMENTAL CONCEPTS, TERMINOLOGY, AND THEORY OF COMPRESSION

It is important to distinguish three related terms.

Data Encoding is a process which maps from a collection of *encoding units* (i.e. one or more symbols is one

Table 1. A small example of a fixed-length code.

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

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

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

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

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

data representation) to a collection of *code values* (i.e. one or more symbols is a second data representation). The relationship between encoding units and their corresponding codes 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 closely related, these concepts are distinct. Some important nonreversible encodings generate effective database access keys with only incidental compaction (e.g. entry/title keys for bibliographic files[52, 46] and phonetic name keys for customer files[22, 74]). Encryption[27, 21] is a reversible data encoding technique (designed to obscure the meaning of sensitive data) which generally yields no compaction. Abbreviation[11] is a compaction technique which is sometimes nonreversible. This paper is concerned exclusively with reversible encodings designed specifically to reduce data storage requirements.

A number of concepts fundamental to our discussion derive from the field of information theory[66, 34, 1, 5]. A *binary digit* or *bit* is defined as a basic measure of storage. Consider a database with M total occurrences of N different encoding units; let l_i denote the length in bits of the ith encoding unit, while p_i denotes its occurrence probability. As in the small example of Table 1, one can always assign the binary numbers 0 to $N-1$ to the encoding units and thereby construct a *fixed-length code* of length $\hat{l}$, for $\hat{l}$ equal to the smallest integer greater than or equal to $\log_2 N$ (denoted $\hat{l} = \lceil \log_2 N \rceil$). The *compression ratio* of a code is the relative amount of storage saved by encoding. Since the original database has length $\hat{L} = M \sum_{i=1}^{N} \hat{l} p_i$ and the encoded database has length $L = M \sum_{i=1}^{N} l_i p_i$; the compression ratio becomes $(L - \hat{L})/L = \sum_{i=1}^{N} (\lceil 1 - \log_2 N \rceil / l_i) p_i$. Table 1 shows that a 85.4% compression can be achieved for a database consisting of the ten most popular words in the English language (assuming that each word is originally terminated by a blank stored in 8-bit character format).

Encoding and decoding with fixed-length codes is straightforward[42, 48]. 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[76]. 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 is accomplished by using the code value to calculate the encoding unit's displacement in the code table. By either designing codes which maintain order relationships for sorting operations[9, 26] or by searching for data using encoded values, decoding operations can sometimes be avoided during data processing operations.

While the 85.4% compression in our example is extraordinary, a *variable length code*, which assigns short code values to frequently occurring encoding units and longer values to the infrequent ones will achieve even greater compression[24]. Information theory establishes the fact that no code can have an average code value length 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 $p_i = 1/N$ for all i. (In this event a fixed length code of length I is the fact optimal.) For our examples $I = 3.01$ and therefore the best code we can devise may have a compression ratio no greater than 89%. A variable length code will therefore increase compression here by at most 3.6% over the simple 4-bit code.

Whether or not it would be worth the effort in a real design situation, the most efficient variable length code is easily found. Specifically, *Huffman codes*[34, 42, 47] have been shown to yield minimum redundancy for a given collection encoding units with known and unchanging occurrence probabilities. One can construct a Huffman Code for any problem by building a binary tree, as follows[65]. 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 Fig. 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

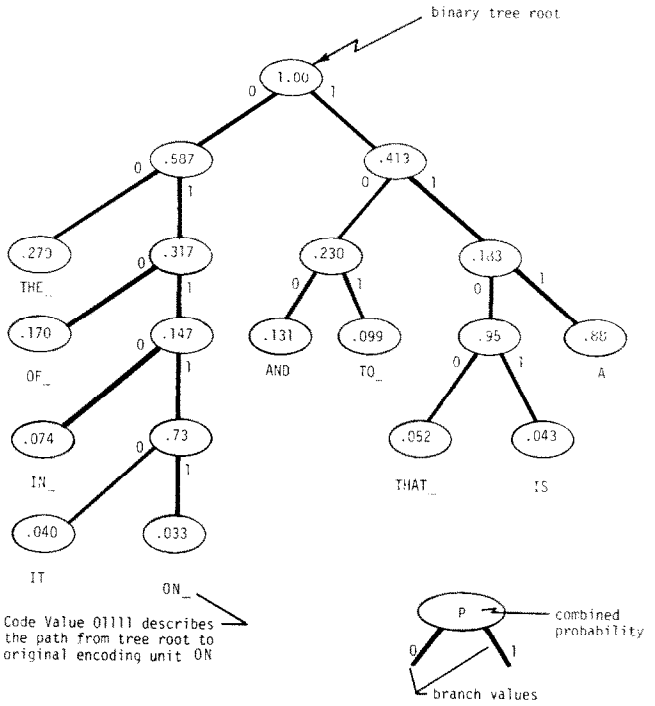


Fig. 1. A Huffman code binary free for the small example.

value $I = 3.01$.† 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 [75, 26].

Table 2. A minimum redundacy Huffman code for the small example

i	ENCODING UNIT	PROBABILITY (P_i)	CODE VALUE	LENGTH (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

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

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

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

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

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

The Huffman coding technique required knowledge of the encoding unit occurrence probabilities, p_i, which for a commercial database can be readily estimated in a variety of ways [12, 39]. When these probabilities are either unknown or change over time, as is the case for data messages arriving for transmission over communication equipment, the data compression problem becomes more complicated. A class of *universal coding schemes* [78, 18] have been devised to compensate for this lack of knowledge. Basically, the techniques employ a memory buffer of some size to capture recent history of the data stream and encode a current data segment to be transmitted in terms of the buffer location and length of an equivalent prior segment. Ziv and Lempel [79, 80] provide an informative discussion of specific universal algorithms and show that they may achieve compression comparable to optimal codes with full a priori knowledge.

4. COMPARISON OF COMMON COMPRESSION METHODS

With the basic concepts presented above, we now compare a wide variety of compression methods by dividing them into two categories. We first discuss compression techniques which view a database as a homogeneous character string and reduce its size through substring encoding, null suppression or value differenc-

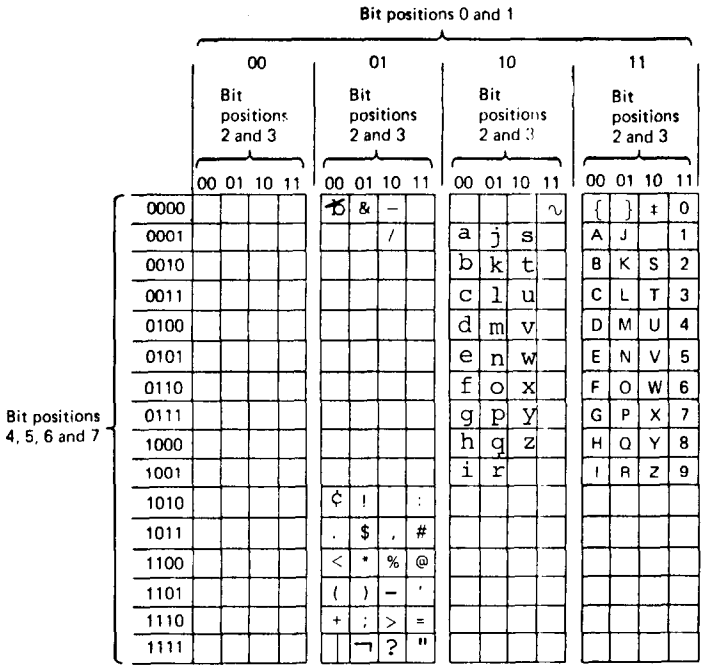


Fig. 2. The extended binary coded decimal interchange code—*EDCDIC*.

ing. We later consider the compression improvements which are possible with these same techniques when the 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.

An analysis of the popular 8-bit fixed length *EBCDIC* code shown in Fig. 2, reveals that code values for all numerics and uppercase alphabetics begin with the bit combination "11." Removal of these bits from four successive characters permit compression to three character positions. The shifting of bits required to accomplish this compression (and decompression) is easily programmed, inexpensively executed and yields a storage reduction of 25%. 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% compression is achieved[47]. Numeric data affords an opportunity for 50% compression[13, 67] and can be accomplished with hardware instructions on some machines[35, 3].

Maximum compression of character strings is achieved with variable-length codes. Heaps[31] 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% of all occurrences[57], the expected code length for a textual database is 5.4 bits ($= 4 \times 0.65 + 8 \times 0.35$), 32.5% 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 0–12 be the table displacements for the thirteen most frequently occurring characters while the code values 13–15 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% 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% reduction.

Huffman coding of single characters with the occurrence frequencies of common English text produces the code shown in Fig. 3[24]. Its expected code length of 4.12 bits yields a 48% compression. Martin[47] provides the Huffman code for a commercial database whose character frequency distribution is shown in Fig. 4. The expected code length of 2.91 bits provides compression of nearly 64%. Gottlieb[25] reports compression results of 50% or more on a variety of large insurance files that already had some numeric data in compact binary form.

Ruth and Kreutzer[60] applied the Huffman coding to a 350-million-character Inventory Requisition File with known occurrence probabilities and found its performance to be "unacceptable." However, by extending the set of character encoding units to include twelve commonly occurring *multicharacter patterns*, a 61% compression was achieved. This was the best performance of twelve alternative codes evaluated. McCarthy[49]

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	01111000
W	0.0175	001110
X	0.0013	01111001100
Y	0.0164	001111
Z	0.0005	01111001111

Fig. 3. A Huffman code for characters in common english text.

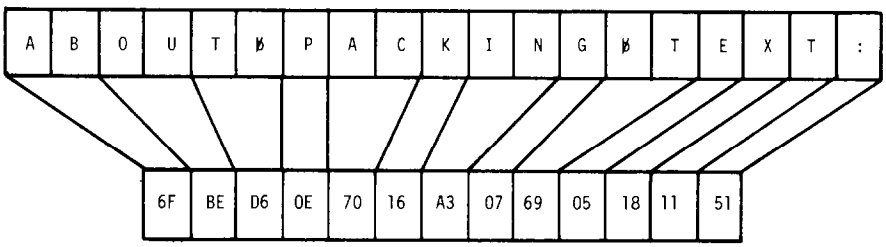
ENCODING UNIT	PROBABILITY	CODE VALUE
0	.555	0
1	.067	1000
2	.045	1100
8	.035	10010
3	.033	10100
A	.032	10101
5	.030	10110
6	.027	11100
4	.027	11101
9	.022	11110
7	.019	100110
F	.015	101110
B	.012	111110
Blank	.011	110110
D	.010	110100
E	.009	110101
Z	.007	1011110
P	.006	1111110
N	.005	1101110
U	.004	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

Fig. 4. Huffman code for a specific commercial database.

Table 3. Digram encoding

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

An Example of Compression



presents a systematic approach for selecting multicharacter encoding units for a Huffman code, and reports the compression achieved on a variety of files: 40% compression on an object-module file, 57% on an English text file, 69% on a name-and-address file, and 82% on a COBOL source file. Guazzo[26] provides an insightful discussion of operational and complexity limitations of Huffman codes and describes more sophisticated algorithms with preferred characteristics for compression of strings of encoding units with correlated occurrence probabilities.

These results suggest that one can increase compression by encoding strings with multiple characters. Encoding units with N characters are referred to as N-*grams*. In an early application of *digram encoding*[10] (where $N = 2$), Synderman and Hunt[68], exploit the fact that data stored with an 8-bit code generally uses no more than 88 of the 256 possible code values leaving 168 code values to represent pairs of characters. Table 3 presents the 168 diagrams and code values (in a hexidecimal format) selected by those authors for compression. Also shown are eight "master" characters and twenty-one "combining" characters from which the 168 (= 8*21) diagrams 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.

Inherently, compression with digram encoding is limited to 50%, which is achieved only when every character is paired into a diagram. Snyderman and Hunt achieved a 35% compression, reducing storage requirements of an online file by 60 million characters. Schieber and Thomas[61] used a more systematic method to establish an efficient set of digrams for a 21-million-character database; they achieved a 43.5% space reduction.

While these compression ratios are not dramatically better than the 25% offered by the simple fixed-length 6-bit code, the method is about as simple to implement. Since digrams can be encoded and decoded rapidly via indexing operations on small tables stored in main memory. Character-length code values eliminate the

synchronization problems associated with Huffman codes, and 43% compression compares favorably not only to that achieved with Huffman coding of individual characters but also to that of more complicated trigram and tetragram encodings[4, 64].

Our example of digram encoding maps two characters into 8 bits. A more general *numerical code*[28] 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 ($A = 0, B = 1, \ldots, J = 9$) with the base-10 number system, the data sequence "*CAB*" can be represented by the number 201_{10} ($= 2*10^2 + 0*10^1 + 1*10^0$). Since the maximum code value in this example is 999_{10}, a machine with a 10-bit word (with value range $2^{10} = 1024$) could store any 3-character sequence in each word. In the same way, if all alphabetics were possible, then "*CAB*" would become $201_{26}(2*26^2 + 0*26^1 + 1*26^0)$, or 676_{10}. 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[29] evaluates values of B between 14 and 73 and shows that five to eight characters can combine in a single number, giving 20–50% compression over as 8-bit code. With $B = 37$ (for alphanumeric data), compression is 20% and numerical coding is inappropriate, since the simpler 6-bit code offers 25% 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," 0, 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% reduction again compares favorably with Huffman coding of individual characters.

Heaps[70, 74] 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[77, 43, 74] 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[30, 70]. For a vocabulary of N distinct terms, codes of lengths $N_1 < N_2 < \cdots < N_r$ 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 $(3, 6, 9, 12, \ldots)$ and $(4, 8, 12, 16, \ldots)$ perform effectively and were easily adaptable to machines with 6- 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% of optimal.

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 length codes 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[63] propose a means of generating *equifrequent word fragments*[14] 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[32]. 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 ap-

†Arithmetic coding is a significant extension of this fundamental idea recently offered by Rissanen and Langdon[58] as a unifying theory for a large class of data encoding techniques. The mathematics of that theory are beyond the depth of this survey.

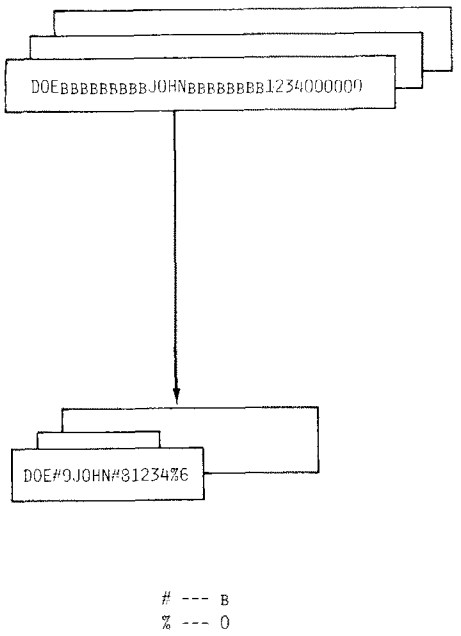


Fig. 5. Null suppression—run length.

proximately 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[44] is reported in [62].

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*[46] illustrated by Fig. 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 Fig. 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[20, 54] report such a use of null suppression to compress a variety of alphanumeric data files by 24–61%. Hill[33] reports an even more dramatic compression of census files by 63–75%.

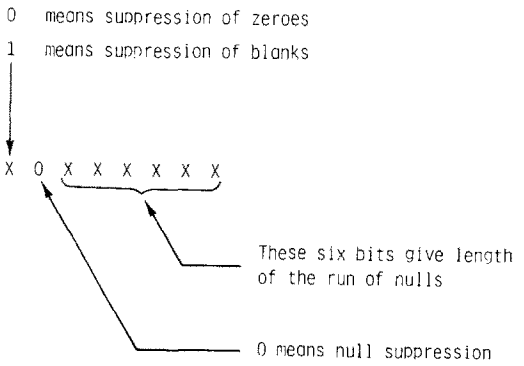


Fig. 6. Null suppression with *EBCDIC*.

Martin[47] describes a run suppression technique for EBCDIC data with no lowercase characters. Here the second code bit is always "1" (see Fig. 2) so that a setting of "0" 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 Fig. 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[29]. 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[23]. 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 hardward 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 50–70% 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 Fig. 4 by 64%; however, the high frequency (55.5%) of zeroes suggests that as much as 50% compression might be achieved via null suppression alone. Martin[47] addressed this question by applying both methods to three commercial databases obtaining the results shown in Table 4. Here Huffman coding provided 12–28% additional compression which may or may not be significant enough in a particular application to justify its additional complexity. If the savings is significant then a variable-length, fixed-increment code should also be considered for its relative simplicity and self-synchronization.

Table 4. Compression results for three commercial databases as reported in[42]

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

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[25]. Ruth and Kruetzer[60] and Villers and Ruth[71] 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%[51].

Comparable relationships between successive data values rarely occur in business applications. Knuth[42] 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 differences 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%.

Date[17] 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 (0) 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. Reghgabti[56] describes a related compaction technique where, prior to compression of leading characters, those trailing characters not required to distinguish a key from adjacent keys are truncated.

Young and Liu[76] analyze the use of difference methods for reducing the main memory storage requirements of encoding/decoding tables themselves. By clustering encoding units based upon either their lexical order or a minimal-spanning-tree procedure described by Kang[41] it is possible to significantly reduce table storage requirements (i.e. by a factor of 5–10) at some cost in loss of code efficiency and decompression speed. The tradeoffs among these factors are analyzed and results of a variety compression experiments are described.

4.4 *Compression enhancement with formatted database records*

For reasons of process efficiency, commercial databases are typically composed of formatted data records[7, 45]. 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 sections 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[12, 40]. 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[47] suggests, for example, that an 8-bit code is sufficient to distinguish 256 surnames, which account for more than 90% of all last names used in the United States. Walker[73] provides an algorithm which has achieved 76% compression on large samples of first names by generating 8-bit codes for equifrequent word fragments. For all but the largest organizations, 16

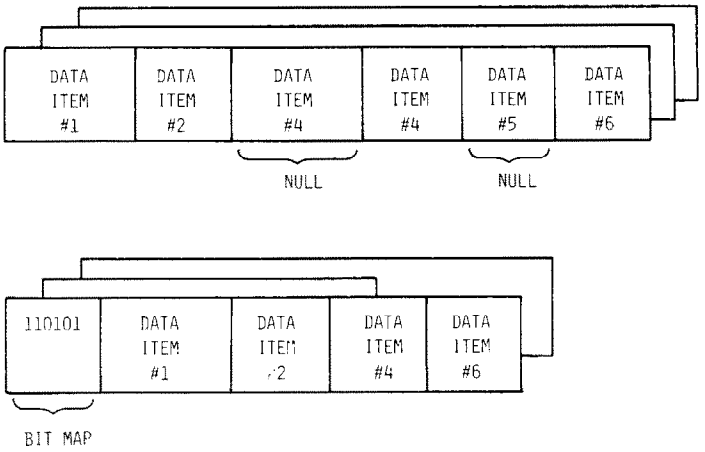


Fig. 7. Null suppression—bit map.

bits (65,536 code values) are ample to distinguish such data as customers, suppliers, employees, or inventory items.

Data 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 yr, which is sufficient for most business applications[2, 47]. In records where a sequence of related dates is stored, each can be represented as its difference from the previous date with significantly fewer bits[25].

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 60–70% compression) that it tends to be the only compression technique routinely supplied with commercial data management systems[16, 36, 39, 69]. A number of methods for compressing null data item values have been described by Olle[53] and analyzed by Maxwell[48]. As illustrated in Fig. 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[27], IMS[38]) 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 required to describe 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 30–90%, 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[15, 16, 40, 69] or easily developed. They execute quickly, require neither code tables nor record formats, and achieve compression of 50–70% 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 30–40%.

Fixed-length codes are a third alternative. They are simple to implement and have the important advantage of providing fixed field and record boundaries, which 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 near uniform occurrence distribution, fixed-length codes achieve near optimal compression.

While Huffman codes in principle achieve optimal compression, their complexity and synchronization problems often make them unattractive. 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, easily adapted to a machine of arbitrary word length, and provide near optimal compression.

Acknowledgments—I am endebted to Dean Wilder, Pat Lung and Jim McKeen, for valued assistance in the preparation of this paper.

REFERENCES

[1] N. Abramson: *Information Theory and Coding.* McGraw-Hill, New York (1963).
[2] P. A. Alsberg: Space and time savings through large data base compression and dynamic restructuring. *Proc. IEEE*, **63**, 1114–1122 (1975).
[3] AMDAHL Corporation, AMDAHL/470 V8 computing system machine reference manual, Rep. G1014, 0–01A, Sunnyvale, CA (Aug. 1979).
[4] J. Aronson: *Data Comparison—A Compression of Methods*, 39 pp. National Bureau of Standards, special publication 500–12 (June 1977).
[5] R. Ash: *Information Theory, Interscience*, New York (1965).
[6] I. J. Barton, S. E. Creasey, M. F. Lynch and M. J. Shell: An information-theoretic approach to text searching in direct access systems. *Comm. ACM* 17, 345–350 (1974).
[7] F. H. Benner: On designing generalized file records for management information systems. *Proc. FJCC*, pp. 145–156 (1968).
[8] E. R. Berlekamp (Ed.): *Key Papers in the Development of Coding Theory*, 296 p. IEEE Press, Piscataway, New Jersey (1974).
[9] M. W. Blasgen, R. G. Casey and K. P. Eswaran: An encoding method for multifield sorting and indexing. *Comm. ACM* **20**, 874–878 (1977).
[10] A. Bookstein and G. Fouty: A mathematical model for estimating the effectiveness of bigram coding. *Int. Proc. Manag.* **12**, 111–116 (1976).
[11] C. P. Bourne and D. F. Ford: A study of methods for systematically abbreviating english words and names. *J. ACM* **8**, 538–552 (1961).
[12] O. Bray and D. G. Severance: Field encoding analysis routine: User's manual and system documentation. *MISRC Techn. Rep.* 77–20, 77–21, GSBA, University of Minnesota (1977).
[13] C. Chen and I. T. Ho: Storage-efficient representation of decimal data. *Comm. ACM* **18**, 49–52 (1975).
[14] A. G. Clare, G. M. Cook and M. F. Lynch: The identification of variable-length, equi-frequency character strings in a natural language data base. *Comput. J.* **15**, (1972).
[15] T. Corey: File compression proves viable alternative. *Computerworld* SR8-SR10 (1980).
[16] Cullinane Corporation *IDMS Concepts and Facilities* Order No. D001, Wellesley Mass. (1977).
[17] C. J. Date: *An Introduction to Data Base Systems*, Addison Wesley, New York (1975). Chap. 2.4, pp. 34–35.
[18] L. D. Davisson: Universal noiseless coding. *IEEE Trans. Information Theory* **19**, 783–795 (1973).
[19] L. D. Davisson and R. M. Gray (Eds.): *Data Compression (benchmark papers in Electrical Engineering and Computer Science*, Vol. 14), Dowden, Hutchinson, Ross, Inc., Stroudsburg, PA. (1976).
[20] P.A.D. DeMaine: The integral family of reversible compressors. *J. IAG*, (IFIPS, Amsterdam) **3**, 207–219 (1971).
[21] W. Diffie and M. E. Helleiman: Exhaustive cryptanalysis of the NBS data encrytion standard. *Comput, IEEE*, 74 (1977).
[22] J. L. Dolby: An algorithm for variable-length proper name compression. *J. Library Automation* 257 (1970).
[23] R. Fajman and J. Borgelt, WYLBUR: An interactive text editor and remote job entry system. *Comm. ACM* **16**, 314 (1973).
[24] E. N. Gilbert and E. F. Moore: Variable-length binary encodings. *Bell System Tech. J.* 933 (1959).
[25] D. S. A. Gottlieb, P. G. Hagerth, H. Lehot and H. S. Rubinowtiz: A classification of compression methods and their usefulness for a large data processing center. *Proc. 1975 Nat. Comput. Conf.*, AFIPS, Vol. 44, pp. 453–458 (1975).
[26] M. Guazzo: A general minimum-redundancy source-coding algorithms. *IEEE, Trans. Information Theory* **26**, 15–25 (1980).
[27] E. Gudes, H. S. Koch and F. A. Stahl: The application of cryptography for database security. *Proc. 1976 Nat. Comput. Conf.* pp. 97–107 (1976).
[28] W. D. Hagamen, D. J. Linden, H. S. Long and J. C. Weber: Encoding verbal information as unique numbers. *IBM Systems J.* **11**, 278 (1972).
[29] B. Hahn: A new technique for compression and storage of data. *Comm. ACM* **17**, 434 (1974).
[30] H. S. Heaps and L. H. Thiel: Optimization procedures for economic information retrieval. *Information Storage Retrieval* **6**, 137–153 (1970).
[31] H. S. Heaps: Storage analysis of a compression coding for document data base. *INFOR* **10**, 47–61 (1972).
[32] H. S. Heaps: Compression of databases for information retrieval and management information systems. Working Paper, Computer Science Department, Sir George Williams University, Montreal.
[33] G. L. Hill: Maximizing computer access to public data files. *Proc. Comput. Sci. Conf. ACM-SIGCSE* (1975).
[34] D. A. Huffman: A method for the construction of minimum-redundancy codes. *Proc. I.R.E.* **40**, 1098–1101 (1952).
[35] IBM Corporation: *IBM System/370 Model 165 Functional Characteristics*, Technical Newsletter, No. GN22-0401 (July 1971).
[36] IBM Corporation: *Utility Reducing Subroutines for Suystem/360/370*, Program Number 5798 AZW (1974).
[37] IBM Corporation: *Customer Information Control System/Virtual Storage (CICS/VS) Application Programmer's Reference Manual*, pp. 402–403. SH20-9003-0, Palo Alto, CA (1974).
[38] IBM Corporation: *IMS/VS, System/Application Design Guide.* SH20-9025-6, (1978).
[39] Informatics, Inc.: *SHRINK/2 User Reference Manual.* Order number 561, Canoga Parck, CA (1978).
[40] Informatics, Inc.: *MARK IV Reference Manual, Version B*, Canoga Park, CA (1979).
[41] A. N. C. Kang, R. C. T. Lee, C. L. Chang and S. K. Chang, Storage reduction through minimal spanning trees and spanning forests. *IEEE Trans. Comput.* 425–434 (1977).
[42] D. E. Knuth: *The Art of Computer Programming*, Vol. 3, Chapter 6.1, Addison Wesley, New York (1973).
[43] M. E. Lesk: Compressed text storage. *Computing Science Techn. Rep.* 3, Bell Telephone Laboratories (1970).
[44] 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).
[45] H. Liu: A file management system for a large corporate information system data bank. *Proc. FJCC* 145–156 (1968).
[46] M. F. Lynch: Compression of bibliographic files using an adaption of run-length coding. *Inf. Stor. Retr.* **9**, 207–214 (1972).
[47] J. Martin: Data compaction. In *Computer Data-base Organization*, 2nd Edn., Chap. 32, pp. 572–587. Prentice-Hall, Englewood Cliffs, New Jersey (1977).

[48] W. L. Maxwell and D. G. Severance: Comparison of alternatives for the representation of data items values in an information system. *Data Base*, **5**, SMIS Special Rep. (1973).

[49] J. P. McCarthy: Automatic file compression. *Int. Comput. Symp.* pp. 511–516. North-Holland, Amsterdam (1973).

[50] J. F. Mulford and R. K. Ridall: Data compression techniques for economic processing of large commercial files. *ACM Proc. Symp. Information Storage Retrieval*, 207–215 (1971).

[51] W. Myers, M. Townsend and T. Townsend: Data compression by hardware or software. *Datamation* 39–43 (1966).

[52] W. L. Newman and E. J. Buchinski: Entry/title compression code access to machine readable bibliographic files. *J. Library Automation* 2 (1971).

[53] T. W. Olle: Data structures and storage structures for generalized file processing. *Proceedings of the FILE* 68 *Int. Seminar on File Organization*, Copenhagen, pp. 285–294.

[54] R. A. Overbeek and P. A. D. DeMaine: The integral family of reversible compressors. The SOLID System Rep. 2, Com. Sci. Dept, Pennsylvania State University (1972).

[55] W. W. Peterson and E. J. Weldon: *Error Correcting Codes*. MIT Press, Cambridge, Mass. (1972).

[56] H. K. Reghbati: An overview of data compression techniques. *Comput.* **14**, 71–75 (1981).

[57] F. M. Reza: *An Introduction to Information Theory*, Chap. 4. McGraw-Hill, New York (1961).

[58] J. Rissanen and G. G. Langdon: Arithmetic coding. *IBM J. Res. Development* **23**, 149–162 (1979).

[59] F. Rubin: Experiments in text file compression. *Comm. ACM* **19**, (1976).

[60] S. S. Ruth and P. J. Kreutzer: Data compression for large business file. *Datamation* 62 (1962).

[61] W. S. Schieber, and G. W. Thomas: An algorithm for the compaction of alphanumeric data. *J. Library Automation* **4**, 198–206 (1971).

[62] E. J. Schuegraf and H. S. Heaps: Selection of equifrequent word fragments for information retrieval. *Inform. Stor. Retr.* **9**, 697–711 (1973).

[63] E. J. Schuegraf and H. S. Heaps: A comparison of algorithms for data base compression by the use of fragments as language elements. *Infor. Stor. Retr.* **10**, 309–319 (1974).

[64] E. S. Schwartz: Dictionary for minimum redundancy encoding. *J. ACM* **10**, 413–439 (1963).

[65] E. S. Schwartz and B. Kallick, Generating a canonical prefix encoding. *Comm. ACM* **7**, 166 (1964).

[66] C. E. Shannon: A mathematical theory of communication. *Bell Syst. Tech. J.* 27 (1948).

[67] A. J. Smith: Comments on a paper by T. C. Chen and I. T. Ho. *Comm. ACM* **18**, 463 (1975).

[68] M. Snyderman and B. Hunt: The myriad virtues of text compaction. *Datamation* 36 (1970).

[69] Software AG *ADABAS DBA Reference Manual*. Software AG of North America, Reston, VA (1981).

[70] L. H. Thiel and H. S. Heaps: Program design for retrospective searches on large data base. *Inform. Stor. Retr.* **8**, 1–20 (1972).

[71] J. J. Villers and S. R. Ruth: *Bibliography of Data Compaction and Data Compression Literature with Abstracts* Government Clearing House Study AD 723525 (1971).

[72] R. A. Wagner: Common phrases and minimum-space text storage. *Comm. ACM* **16**, 148–152 (1973).

[73] V. R. Walker: Compaction of names by X-grams. *Proc. Am. Soc. Inform. Sci.* **6**, 129–135 (1969).

[74] G. Weiderhold: *Database Design*. McGraw-Hill, New York (1977).

[75] M. Wells: File compression using variable-length encodings. *Comput. J.* **15**, 308–313 (1972).

[76] T. Y. Young and P. S. Liu: Overhead storage considerations and a Multilinear method for data file compression. *IEEE Trans. Software Engng* **6**, (1980).

[77] K. G. Zipf: *Human Behavior and the Principle of Least Effort, An Introduction to Human Ecology*. Addison-Wesley, Reading, MA (1949).

[78] J. Ziv: Coding of sources with unknown statistics—part I. Probability of encoding error. *IEEE Trans. Inform. Theory* 384–394 (1972).

[79] J. Ziv and A. Lempel: A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory* **23**, 337–343 (1977).

[80] J. Ziv and A. Lempel: Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory* **24**, 530–536 (1978).