This file is ©1977 and 1983 by McGraw-Hill and ©1986, 2001 by Gio Wiederhold.

Chapter 14

Coding

We do not recommend the use of “Roman Numerals” in Western Electric Information Systems.

M. J. Gilligan
in McEwan[74]

The contents of a database is intended to be a representation of the real world. It is desirable to have an accurate and complete representation of our knowledge about reality so that decisions made on the basis of extracted information are applicable to the real world. It is, however, soon obvious that the model of the real world, as entered into a database, is both crude and incomplete.

While searching for a representation there is a reliance on abstraction: one speaks of a specific individual as an employee, a welder, a manager; and then this abstraction is encoded into numbers or characters, which are subsequently transformed into binary computer codes. Statements that *the database content represents reality* may indicate the goal of a system designer, but will be misleading to the user who has to reconstruct useful information from these codes.

During encoding and decoding a number of distortions and biases are introduced. The first section will investigate representation issues with particular regard to these problems. Standard codes will be presented in Sec. 14-2 and the chapter will close with a presentation of code-compression techniques.

This material is presented to provide guidance to applications designers when faced with the initial stages of a database problem. It is not unusual that a programmer is asked to *computerize* an information problem. Since a traditional computer science or engineering education has been concerned only with data-processing rather than with the transformation of knowledge to data, this facet of the database design effort may be neglected. Decisions implemented at the level of representation will bind the database to a specific content for many years of its operation. A designer of a database systems should be aware of alternatives of representation so that the system will support diversity and only restrict the unreasonable.

14-1 REPRESENTATION OF KNOWLEDGE

In this section we will present in a logical sequence the steps which are used to transform observations regarding the real world into data elements. In practice these steps can be combined, be ordered differently, or occur more than once.

The initial filter which reduces the capability of the database to represent the real world is the fact that we can only consider available knowledge, correct or wrong, verified or presumed, for representation. The fact that we don't know everything, or even very much, becomes clear when we have to act based on available information.

14-1-1 Taxonomy

Not all the available knowledge will be a candidate for incorporation into a database. Which aspects of our knowledge about an entity will be most useful is a decision that has important consequences on the eventual ability to reconstruct reality. The decisions leading to the categorization of observations have been formally studied by taxonomists, and their results have been applied to the selection of distinguishing characteristics in plants and animals. In database design the categorization is, in general, based on tradition and availability of data. In the traditional view the productivity of an enterprise is represented by its gross sales, and maybe by its tax contribution or payroll in the community. The satisfaction provided by the work to the employee or its effect on the environment has not been measured and collected, and hence will not appear in the database.

A similar decision concerns the level of aggregation of elemental data. Data can be collected in fine detail, or data collection can restrict itself to summarized data.

As an example we can take data collection regarding a patient. The medical diagnosis is an obvious data element. But when this data element is analyzed, one can see that a diagnosis in most cases is a convenient shorthand notation for a constellation of symptoms, complaints, and physical observations. These will not be identical for everyone who receives the same diagnoses; at the same time patients with several similar symptoms will be diagnosed differently. So one may decide to collect data at a lower level of aggregation and enter all the parameters which lead to a diagnosis. Some of these parameters themselves are results of aggregations. The appropriate level is determined by the intended purpose: public health control measures, individual patient records, disease research, cell physiology, or molecular biology will require different levels of aggregation.

At a lower level of aggregation there will be more individual entries, increasing the cost of collection, storage, and processing. There may also be a greater variety of entities, and a greater variety of appropriate attributes. The benefit of having very detailed data may be hard to quantify. In the case of a bill-of-materials problem, there will be little benefit in obtaining control of minor standard parts: bolts, nuts, washers, and lubricants, so that these will only be registered as aggregate supplies. Other parts at the same hierarchical level may be expensive or specialized and warrant specific record keeping.

When a database is envisaged to serve the needs of users who need different levels of aggregation, data at the most detailed level has to be collected. Periodic or immediate updating of aggregate measures can be used to keep data at higher levels available.

When the entities and their attributes to be collected are defined, the specification of what constitutes membership of an entity class has to be further refined. It is, for instance, not obvious whether the paint on a bicycle is a part or an attribute value. Often there is an established basis which can be used to determine membership. Rules of auditing, for instance, specify in detail the types of income and expenses which are to be summed into ledger categories.

The outcome of a taxonomic decision requires documentation to be used at data entry:

"The number of children is based on the children of either parent, living in the same household, and less than 18 years old."

An alternative specification can be by reference:

"The number of children is as defined for federal tax purposes."

It is obvious that without such statements, different data collectors will record different information. An extended schema provides a central repository for these specifications.

14-1-2 Classification

When the entities and their attributes have been selected, the observation choices have to be classified. Sometimes the decision is simple:

"The number of children of a mother is specified by the integers 0, 1, 2, ... "

More frequently the observed data are not classified this simply. Classification of

" The weight of the children ... "
" The hair color of the children ... "

requires further decisions.

We will list some classification choices below and then comment on their usage. The classification of a type of observation is not always obvious. Once a data element type is classified, however, important aspects of the domain for the attribute values are established and an appropriate representation can be selected.

Whenever a *natural* order exists, the data values can be *ranked* into a sequence, which will be exploited in coding and data-processing. When there is no such natural order in the observations, an artificial order may be specified by convention.

Table 14-1 Classification of Attribute Values

Sortability	Measurability	Value choices	Sample domain
Ranked	Metric	Continuous	`weight`
		Integer	`number_in_stock`
	Ordinal	–	`friendliness`
Unranked	Nominal	Unlimited	`name`
		Limited	`haircolor`
	Existential	–	`female`

A schema which provides a ranking specification was shown in Example 8-6; SOCRATE allows a domain to be specified by a list, (e.g. `hair_color(black, brown, red, blond)`), and the order in the list defines the ranking. It remains, however, risky to formulate queries comparisons having `GREATER` or `LESS` operations unless the order definition or *collating sequence* is clear to the user.

If the values are *metric*, the data values are determined by measurement. Arithmetic operations can be applied to these data. The values within a domain `weight` can be added or subtracted as well as compared. This also means that operations which generate a mean or a standard deviation make sense. When data is *continuous*, decisions about range and accuracy have to be made. The limits imposed by computer hardware makes the choices often seem simple. The fact, however, that a computer retains six-decimal-digit precision does not mean that this degree of precision is warranted in data collection. Subsequent calculations can produce numbers of apparent accuracy which was not intrinsic to the original data.

The response to a query:

```
Joe weighs 68.0385 kg
```

does not correspond in meaning with the data as they were entered:

```
Joe's weight is 150 pounds.
```

Documentation in the schema of the accuracy of data entered can be used here to provide an appropriate truncation: `68.0 kg`.

For *ordinal* or *nominal* data, arithmetic operations should be restricted. An investigator may indeed develop a metric scale for data values which are intrinsically ordinal, but the results have to be taken with care. The measure used for intelligence (IQ) is such an example. An average is computed based on the assumption that intelligence has a normal distribution about the mean. With this assumption individuals can be assigned meaningful percentiles.

The statement

```
Proteus is twice as intelligent as Berringer.
```

cannot be based on such an artificial metric. When there is no reason to assume a specific distribution, the use of averages themselves is also objectionable:

```
Joe was involved in an average car accident.
```

The mode and the median remain computable in ordinal as well as in integer data since they can be obtained by comparison and counting.

When data values are strictly *nominal*, the most frequent occurrence, the mode, can still be found. "`Lee`" can be the most frequent name in an area, but no ranking

is implied by the value "Lee". Joe's hair color cannot be considered less or greater than Mike's unless some ranking is introduced: perhaps light versus dark.

Nominal data values which have *unlimited* choices allow arbitrary text. Names of individuals, names of businesses, statements of opinion, etc., are data of this type. Accuracy becomes difficult to control and the values can only be used for comparison. Nominal data values cannot always be avoided, but if a limited domain can be established, many opportunities for data verification and control become available. The list of values may be so great that a lexicon will be required to define the domain. Sometimes a domain definition can be available as part of a coding convention, as shown in Sec. 14-2-2.

14-1-3 Encoding

Once the value domains to be collected have been classified, the selection of the actual data representation in the computer completes the conversion of knowledge to data specifications.

For values which have a metric classification the choice is straightforward. Continuous data can be represented by binary integers or fractions if the range of values permits. The position of a binary or decimal point can be noted in the schema. An alternative available on many computers is the use of a floating point notation. Occasionally a representation of values as rational fractions, using two integers for the dividend and the divisor, can be useful. An adequate representation of UNDEFINED or UNKNOWN values may prove difficult. On systems which use sign-magnitude representation for numeric values, the value "-0" has been used, but this convention can be recommended only where the generation of numeric values is controllable by the database-system designer.

Ordinal data are often given integer representation. A lexicon, internal or external to the system, is required for the translation. If ordinal data are not translated to integer or sortable alphabetic codes, a lexicon is required to carry out comparison operations:

Example 14-1 Encoding Table for a Domain

```
Grade:  RELATION
     impression ⟨:⟩ value;
   Excellent        4
   Good             3
   Adequate         2
   Inadequate       1
   Failure          0
```

Nominal data of a limited domain is generally associated with a lexicon to verify correct data entry. The data may be represented in integer form for convenience in handling and storage. This will require another conversion, using the same lexicon in the opposite direction, whenever the value is to be produced as a result. Occasionally coded nominal data is not reconverted for output. This tends to be valid only where there is a standard encoding and where immediate intelligibility is not important. Medical diagnosis codes, for instance, can be part of the output

from hospital billing systems, since they will be entered as such into the computers of the medical insurance companies. Better public relations and an integrity check is provided when the patient can also see whether the code on the communication signifies a tonsillectomy or a heart transplant. Both code and nominal value should be given in such cases.

If the encoded values carry a check digit or are otherwise redundant, some internal processing errors can be detected. Inadvertent computations will generate bad check digits, and when the lexicon is called upon to regurgitate the data value corresponding to the code, the error will be noted.

Unlimited nominal values can only be translated to an appropriate character code. These codes, in general, will have some redundancy which helps to locate errors. A listing of values in the database which occur only once or a few times can provide a tool to check on misspellings. Since unlimited nominal codes participate rarely in processing, some systems (RDMS, TOD) have replaced nominal values with reference codes. The actual character strings are kept on a remote file and accessed mainly during output procedures.

Bit positions

4321 \ 765	000	001	010	011	100	101	110	111
0000	NUL	DLE	blank	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Bit numbering is right to left within a character.

Code	Meaning	Code	Meaning	Code	Meaning
ACK	Acknowledge	ESC	Escape	NUL	Null
BEL	Audible signal	ETB	End of transmission block	RS	Record separator
BS	Backspace			SI	Shift in
CAN	Cancel	ETX	End of text	SO	Shift out
CR	Carriage return	FF	Form feed	SOH	Start of heading
DCx	Device control	FS	File separator	STX	Start of text
DEL	Delete	GS	Group separator	SUB	Substitute
DLE	Data-link escape	HT	Horizontal tabulate	SYN	Synchronous idle
EM	End of medium	LF	Line feed	US	Unit separator
ENQ	Enquire	NAK	Negative acknowledge	VS	Vertical tabulate
EOT	End of transmission				

American Standard Code for Information Interchange

Figure 14-1 ASCII character coding

14-2 MACHINE REPRESENTATION

The representation of all categories discussed in the previous chapter is accomplished within a computer system by strings of bits. For all categories except unlimited nominal data, a data element can be coded as an integer or floating-point number. These data elements are typically of fixed size.

When data remain in character string form, the data element is represented by a sequence of binary coded characters, typically of varying length. Figure 14-1 presents the standard encoding for characters in the interchange of information, i.e., files and communication lines. Each character is represented by 7 bits; on machines with 8-bit character spaces, one bit will remain unused. Alternate character encoding methods in use for data storage are listed in Table 14-2.

Table 14-2 Character Codes and Their Sizes

Common name (organization)	Size	Number of symbols		
		Data	Control	Undefined
`Baudot (CCIT)`	`5 bits+shift`	`50`	`5`	`3+`
`BCD (CDC)`	`6 bits`	`64 or 48`	`0`	`0 or 16`
`Fieldata (US Army)`	`6 bits`	`48`	`16`	`0`
`ASCII (ANSI)`†	`7 bits`	`95`‡	`33`	`0`
`EBCDIC (IBM)`	`8 bits`	`133`	`64`	`59`

† American National Standards Institute. A similar code has been adopted by ISO, the International Standards Organization.

‡ For international usage some characters can represent alternate symbols.

Control characters are used in data communication to designate empty spaces, to separate data units, and to control output device functions such as `line feed`, `carriage return`. In databases control characters are useful to delimit variable-length fields and to represent `UNDEFINED`. Such usage requires care to avoid conflict with assigned hardware functions. The ASCII symbols `FS, GS, RS, US,` and `DEL` may be suitable.

14-2-1 Conversion

Input data is converted to the binary representations when entered. Most common are computational procedures which take Hollerith character strings and convert the character representation to the desired values. Conversion is a costly procedure. Low error rates and reasonable cost are essential to database operation.

Coding during Data Entry Data-entry techniques can influence the choice of data representation. Of particular importance are the use of multiple-choice forms or display screens which allow nominal data to be selected as seen in Fig. 10-5. The code for the box checked is entered into the computer system. The table to translate from code to the intended term has to match the presented form precisely. Since forms and displays change over time, it is desirable that the code does not represent the position of the entry on the form, but is translated into a code which can be maintained over time.

These self-coding forms or multiple-choice questionnaires are seen in any application field. Digits or arbitrary characters are entered into designated spaces, which

correspond in size to available space in the file records. It is possible to generate prototype forms by using data from the schema. This assures that data entered match the file specifications.

14-2-2 Standard Codes

The representation of nominal data by integer codes is a frequent occurrence. We have seen systems (PASCAL and SOCRATE) which provide definitional capability in order to relieve programmers of the task of assigning codes to words.

When data have to be shared among larger groups, the set of values to be encoded has to be agreed upon in order to ensure completeness, lack of ambiguity or redundancy, and appropriateness of level of aggregation. This can be an extremely time-consuming task.

Many organizations have found it necessary to define standard encodings for a large variety of data elements. Activities in the field have been led by groups in the Bell System, trade organizations, defense agencies, federal procurement agencies, and the Bureau of the Census. Some international standards are also developing. If a standard encoding can be used, a considerable amount of system-design effort can be saved. Further integration of the database with other areas is also simplified. The assignment of observations or concepts to the terms provided in a coding table can still cause some debate when an existing code is brought into a new environment. An example of a problem encountered has been the assignment of values to a currency code for countries which have different conversion regulations for investment and for tourism.

Table 14-3 provides some references to standard code tables. The list is by no means exhaustive. Many of these codes are available on tape to simplify their incorporation into data-processing systems.

14-2-3 Coding of Names of Individuals

The coding of names remains as one of the major problems in data-processing. The use of the social security number to identify individuals is restricted to files which relate to activities involving payments of taxes or disbursements from federal insurance or support programs. These are quite pervasive so that the social security number is used in many files, including some where its use is not appropriate. The use of names is hampered by two issues.

Clustering One problem with the use of names is the fact that some names appear very frequently. The Social Security Administration [SSA[57]] has found about 1 250 000 different last names in its files, but also that 48% of the people share the 1514 most common names. Thirty-nine names occur with a frequency greater than 0.1%. The use of first names can resolve the clusters to some extent, but 768 four-letter combinations comprise the four initial characters of more than 90% of the first names. Endings of first names show frequent changes (i.e., `Johnny` for `John`) and are hence not very reliable as identifiers. Duplication of entries for single individuals seems inevitable. When queries to name files are made in an interactive query situation, all collisions of the search key can be presented. A choice based on further data (`address, age`) can be made if the full name displayed is not yet adequate.

Table 14-3 A Sampler of Standard Codes

Entities code name	Organization location		Number of entries: format of code†
GEOGRAPHY:			
States of the US	National Bureau of Standards		56:AA or :NN
USPS identifier	FIPS 5-1 Gaithersburg, MD		
Counties in states	NBS, FIPS 6-2		:NNN
Standard Metropolitan			
Statistical Areas	NBS, FIPS 8-3		:NNNN
Congressional Districts	NBS, FIPS 9		:NN
Countries	NBS, FIPS 10		
Place names in US and	Comp.Bus.Equip.Manuf.Ass.		130000:AANNNNN C
a Class code for each	(CBEMA) Washington, DC		
Mailing Addresses, ZIP Code	US Postal Service		:NMMNN-MMM
	Canadian and British PO's		:ANA-NAN
ORGANIZATIONS:			
Businesses	Dun and Bradstreet, Inc		¿3000000
D-U-N-S	New York, NY		:NNNNNNNNN
Employers Identification	Internal Revenue Service		:XX-YYYYYYY
EIN	Washington, DC		
Occupations	Bureau of the Census		:XXXX
SUPPLIES:			
Automobiles	Consumers Union		{all cars since 1970}
Commodities (US Transp.)	Assoc. of Am. Railroads		:NNNMMMMMM
STC Code	DOT, Washington, DC		
Commodities (Internat. Trade)	United Nations		8 codes·350
SIT Classification	New York, NY		categories:NNNNN
Biologically active agents	College of Amer.	Chemicals:	700:NNN
SNOP	Pathologists.	Enzymes:	200:NNN
Electric Equipment Parts	Western Electric, Inc.		various
WE standards	Newark, NJ		:hierarchical
Groceries	Distribution Codes, Inc.		:NNNNN MMMMM or
UPC (Bar code)	Washington, DC		XBBBBB BBBBBX
Publications	Library of Congress		(3 standards,
MARC	Washington, DC		100 elements)
Scientific Instruments	Am. Ass. Adv. of Science		:NNNNN
Guide to Sc. Instr.	Washington, DC		
MISCELLANEOUS:			
Data Processing Outputs	Hdq. US Army Material Command		9000:
AMC-DED	Alexandria, VA		
Device Failures	Western Electric, Inc.		:NNNMMM
WE Standard No. 10178	Newark, NJ		
Human Body	College of Amer.	Topology:	2000:NNNN
SNOP	Pathologists	Morphology:	1300:NNNN
(Systematic Nomenclature)	Skokie, IL	Etiology:	1500:NNNN
		Function:	1200:NNNN
Motorists (accidents,insur-	National Highway		various
ance,vehicles,highway plan.)	Safety Administration		
ANSI-D20	Washington, DC		
Work-injuries ANSI Z16.2	Am. Nat. Stand. Inst.		7 categories

† A,C *denotes an alphabetic,* N,M *denotes a numeric,* X,Y *denotes a mixed, alphanumeric code.*

Name Spelling Variations An even more serious problem is the fact that the spelling of a name is not necessarily consistent. Whereas clustering leads to retrieval of too many names, variantion of spellings causes failure to retrieve information. Examples of common variations are listed on the left in Example 14-2. Cross referencing can be used to refer to frequent alternatives, but the synonym list soon becomes large.

Example 14-2 Name Variations and Soundex codes

Names	Soundex codes
McCloud, MacCloud, McLoud, McLeod, M'Cloud	M253,M253,M253,M253,M253,M253
Ng, Eng, Ing	N2,E52,I52
Rogers, Rodgers	R262,R326
Smith, Schmid, Smid, Smyth, Schmidt	S53,S53,S53,S53,S53
Wiederhold, Weiderhold, Widderholt, Wiederholdt, Wiederhout	W364,W364,W364,W364,W363

A technique to locate together all similiar sounding names which has seen intensive use is the Soundex method defined in Table 4-4. It can be viewed as a type of phonetic hashing.

Table 14-4 Soundex Procedure

All nonalphabetic characters (',-,␣, etc.) are eliminated.
All lowercase letters are set to uppercase.
The first letter is moved to the result term.
The remainder is converted according to the next 4 steps:
  The frequently unvocalized consonants H and W are removed.
  The following replacements are made:

Labials	:	B,F,P,V	→	1
Gutterals, sibilants	:	C,G,J,K,Q,S,X,Z	→	2
Dentals	:	D,T	→	3
Longliquid	:		→	4
Nasals	:	M,N	→	5
Shortliquid	:	R	→	6

  Two or more adjacent identical digits are combined;
    thus, LL → 4, SC → 2, MN → 5.
  The remaining letters (the vowels: A,E,I,O,U, and Y) are removed.
The first three digits are catenated to the result.

For some names liable to inconsistent spelling this procedure will generate the Soundex codes shown in Example 14-2. It can be seen that many variations are completely merged, although some remain distinct. This scheme appears biased toward Anglo-Saxon names; other codes have also been constructed.

Use of Soundex codes as a primary key increases the number of collisions. This code provides at most $26 \cdot 7 \cdot 7 \cdot 7$ or 6 734 choices, and the alternatives are not uniformly distributed. The first letter alone causes a poor distribution as shown in Table 14-5 [SSA[57]]. For comparison the occurrences of English words in a text of 4 257 692 words and the distribution of 5 153 unique words in this same text are also shown [Schwartz[63]].

Table 14-5 Distribution of Names, Words and Word Types by Initial Letters

Initial Letter	Names(tokens) %	Rank	Text (tokens) %	Rank	Dictionary(types) %	Rank
A	3.051	15	12.111	2	6.229	4
B	9.357	3	4.129	9	5.550	7
C	7.267	5	3.916	10	9.722	2
D	4.783	10	2.815	13	6.016	5
E	1.883	17	1.838	18	4.386	11
F	3.622	13	3.911	11	5.162	9
G	5.103	8	1.960	16	3.086	16
H	7.440	4	6.937	5	3.842	12
I	0.387	23	8.061	3	3.707	13
J	2.954	16	0.427	23	0.776	21
K	3.938	12	0.576	21	0.602	22
L	4.664	11	2.746	14	3.474	14.5
M	9.448	2	4.429	8	4.560	10
N	1.785	18	2.114	15	1.844	18
O	1.436	19	6.183	7	2.271	17
P	4.887	9	2.897	12	7.801	3
Q	0.175	25	0.199	24	0.427	23
R	5.257	7	1.880	17	5.317	8
S	10.194	1	6.787	6	12.886	1
T	3.450	14	15.208	1	5.608	6
U	0.238	24	1.008	20	1.417	20
V	1.279	20	0.428	22	1.436	19
W	6.287	6	7.643	4	3.474	14.5
X	0.003	26	< 0.001	26	< 0.001	26
Y	0.555	21	1.794	19	0.369	24
Z	0.552	22	0.002	25	0.039	25

When the identity of an individual has to be resolved, secondary information, such as `address, birthdate, profession, birthplace, parents' names` may still be required. Which parameters are appropriate will differ according to circumstance. The month of one's birth, for instance, tends to be remembered more reliably (98%) than the day of birth (96%) or the year of birth (95%).

The relative freedom with which individuals in the United States can change their name makes positive identification of an individual difficult. The social benefits of this data-processing difficulty may outweigh the costs. Concerns about protection of privacy everywhere are discouraging the use of personal identification numbers. Some European countries are discontinuing their use.

The use of numbers as identifiers will not disappear. Individuals will have many numbers as identifiers: tax accounts, license numbers, bank accounts, employee numbers, etc.

14-3 COMPRESSION OF DATA

Data storage remains an important cost factor in many applications. Often the representation of the data causes unused space and redundant representation of data. While some redundancy may be part of schemes to improve reliability, significant opportunities for data compression remain. Compression is possible when any of the following conditions are true:

- High frequency of undefined or zero elements
- High frequency of small integers in fixed size words
- Low precision requirements for floating point numbers
- Values which range over a limited domain
- Values which change slowly
- Fixed length spaces for character strings
- Redundancy in character strings
- Character sizes in excess of character set needs

We will consider only schemes which allow complete reconstruction of the data when the compressed data are again expanded. In Sec. 4-2 we encountered *abbreviation* methods for index entries, which did not permit full reconstruction.

There is, of course, a trade-off between the degree of compression and the cost of processing time. Compression will also reduce data-transfer requirements so that file operations may become faster. The total trade-off to be considered is compression CPU time versus file access speed and storage cost.

14-3-1 Compression of Numeric Data Elements

For computational purposes, it is desirable that numeric values occupy full words. Frequently the data form matrices in order to allow rapid computation across rows and columns. Whenever these data represent observations of real world phenomena, it is not unusual that positions are allocated for values that have not been collected (`u`) or are of zero value.

By *storing only nonzero data values with their coordinates* the zero elements are compressed out of the file. The example below applies this idea to a record having $a = 20$ numeric data spaces.

Original record:

```
0,14,0,0,0,15,0,0,0,3,3,0,0,0,0,0,0,423,0,u;
```

Coordinate representation:

```
2:14,6:15,10:3,11:3,18:423,20:u;
```

Now the record requires $2a' = 12$ data spaces. This method is reminiscent of the record format shown for the pile-file organization presented in Sec. 3-1.

An alternative is the use of a descriptive *bit vector*. One bit is used per field, a `0` indicates the absence of a value. Now the same record is represented by

`01000100011000000101`$_2$`:  14,15,3,3,423,u;`

or, if the bit vector is shown as a single decimal integer

```
280068:  14,15,3,3,423,u;
```

If both zeros and undefined values occur, the bit vector can be used to describe the three choices:

```
 0  zero value
10  undefined value
11  actual nonzero value to be stored
```

The bit vector for the same record, now without `u` stored as data, becomes:

$0110001100011110000011010_2$: `14,15,3,3,423;`

In this case there is no benefit; where `undefined` values are plentiful the effect will be felt. This last bit vector is more difficult to manage, since its size will change depending on the values in the record. In an application where values repeat frequently, four choices can be coded:

```
00  zero value          01  repeated value
10  undefined value     11  stored value
```

so that the fixed-size bit vector now reads

$0011000000110000001101000000000000110010_2$: `14,15,3,423;`

This four-choice encoding for compression was applied to a file system which supported multiple databases using a total of more than three diskpacks and gave a reduction of 44% when applied to all files and 45% when applied only to records which became smaller. Some large statistical files were reduced by more than 80%.

When the *range of numeric values is small*, the numbers can be placed into a smaller space. It is unfortunately difficult to indicate boundaries within a bit string, so that the variable space assignment is awkward to implement. If numeric data are coded as decimal digits of four bits each, one of the $2^4 = 16$ codes can be used to provide a boundary. Such a code can be generated using the four low order bits of the ASCII character code (Fig. 14-1) for the ten digits, leaving space for six symbols, perhaps: `* + - , .` and `US`.

For binary values the conversion to fixed-size decimal digits does not make sense; it is better to use a fixed size for a record based on the values to be stored. A scan of the record can be used to determine the largest required space, and this size can be kept as another prefix with the record. Using again the same values we find that the largest data value fits into 9 bits, so that the record will have as contents

$0011000000\ldots000110010_2$: `9,014|015|003|423;`

using `|` to indicate the boundaries of the 9-bit units. Here one word plus 36 bits is required instead of four words. The method is defeated by the presence of a single large value in the record.

An alternate input for compression can be a schema which specifies the range of values. Then the record can be formatted appropriate to the range in any column. In Example 8-7 the `years` were specified to range from 0 to 20, allowing storage of these values in fields of size $\lceil \log_2 20 \rceil = 5$ bits each.

Low-order positions of floating-point numbers can be truncated if the required precision is low. Such a decision would also depend on the specifications provided through the schema. Instrumentation data is often of low precision, and 16-bit floating-point formats with 10-bit or three-digit precision and a range up to $\pm 10^9$

have been devised. If floating-point numbers are used to represent integers or fractions with a known binary point, then a representation using binary integers may save even more space.

If data elements represent attribute values which *change slowly*, it can be attractive to store the differences of successive values. A capability to store small numeric values efficiently is required to make this scheme effective. A sequence,

```
854,855,857,856,859,860,863;
```

can be represented as

```
854,4,+1|+2|-1|+3|+1|+3;
```

Four bits are sufficient to store the successive differences. Unless we are dealing with transposed files, records containing sequential values tend to be rare. The method could be applied to selected columns over many records, but then the record cannot be processed unless it is sequentially accessed.

14-3-2 Compression of Character Strings

Compression of character strings is more frequent than compression of numeric values. One reason is that there are a number of large systems specifically oriented toward textual data, and a pragmatic reason is that systems which have to support text have had already to support variable-length data elements and records, so that the additional complexity introduced by compression is less.

Most schemes to compress text limit themselves to compression of the text representation. There is also a great deal of semantic redundancy within language sentences. It is estimated that 50% of English text is redundant. The redundant information, however, is not distributed in any regular fashion, so that automatic deletion of this redundancy is difficult.

In Sec. 4-2 techniques for abbreviation of words in indexes were presented. These techniques were effective because the keys appeared in sorted order, so that the redundancies were easily eliminated. The same techniques may at times be of use to compress data portions of files. Low-order abbreviation, however, causes loss of information. Abbreviations are frequently used independent of data-processing. They may range from

`M` for "Monsieur"
to `ADCOMSUBORDCOMAMPHIBSPAC` for "Administrative Command, Amphibious Forces, Pacific Fleet, Subordinate Command."

Such abbreviations present a degree of data compression prior to data entry.

Abbreviations can reduce data-entry volume as well as storage costs. When abbreviations are developed specifically to deal with the database, their effect should be carefully considered. Compression of terms within the computer may be more economical than the abbreviations which simplify manual processing, but a reduction of manual data-processing errors is extremely worthwhile.

Recoding The most frequent technique used to compress textual data is the representation of the data in a character set of smaller size. Data may, for instance, be recoded from an 8-bit set to a 7- or 6-bit set. With 6 bits it is just possible to represent the digits, upper- and lowercase characters, a hyphen, and the blank,

since $2^6 = 10 + 2 \cdot 26 + 2$. Most 6-bit codes do not retain the lowercase characters in order to provide a more complete set of special characters.

Recoding from 8 to 6 bits saves 25% of the space. This is often accomplished by deletion of the leading two bits of the ASCII code. Most modern output devices can present both upper- and lowercase characters; for much of the public the rule that *computer output is always uppercase* still holds. It seems unwise to remove the humanizing appearance of upper- and lowercase print to achieve data compression.

Single-case characters alone can be represented by 5 bits. If alphabetic and numeric characters appear mainly in separate sequences, the use of a method similar to Baudot coding, which uses a shift character to switch between sequences, can be used to represent text in an average of slightly more than 5 bits per character.

If the data represent digits, only the four low-order bits of most codes have to be retained. Starting from ASCII a blank (␣) becomes a `"0"` but the 6 characters listed earlier (`* + - , .` and `US`) will be kept as distinguishable codes. Any other characters present will map into this set and probably cause errors.

Compression by recoding symbols into a specific number of bits will be optimal if the number of symbols to be represented is close to, but less than, a power of two. This is not the case for character sets containing only digits, or character sets containing upper- and lowercase characters, digits, and at least the 16 common special characters from ASCII column `010`. These sets of 10, respectively 78 symbols, require 4 or 7 bits per character, adequate for 16 or 128 symbols. Recoding groups of a few characters into a single representation can produce a denser compression, as shown in Table 14-6. The limit for a group is, of course, 2^{bits}.

Table 14-6 Character Group Encoding Alternatives

Character set	Set size	Group	Group range	Bits	Limit	Saving, %
Digits	10	3 digits	1 000	10	1 024	16.6
Single-case alphabet	26	4 letters	456 976	19	524 288	5.0
Alphanumerics+28	90	2 characters	8 100	13	8 192	7.1
Alphanumerics+18	80	3 characters	512 000	19	524 288	9.5
Alphanumerics+22	84	5 characters	$4.29 \cdot 10^9$	32	$4.19 \cdot 10^9$	9.25

The saving is computed based on an ungrouped dense encoding. The bit sizes for the groups may not match word boundaries, so that multiword units have to be handled in order to exploit this type of compression. The grouping of 5 alphanumeric characters, however, allows a 32-bit word to be used to represent 5 characters, a saving of 20% over the common 8-bit-per-character encoding.

Deleting Blanks In formatted data there tend to be long sequences of blanks. A simple technique to compress such sequences is to replace a string of i blanks by a sequence of two characters `"␣i"` as shown in Fig. 14-2. The value of i will be limited by the largest integer that can be kept in a character space (63, 127, or 255). In the scheme shown a single blank is actually expanded and requires two spaces. This can be avoided by letting single blanks remain and replacing longer strings of blanks by some otherwise unused character, followed by the count.

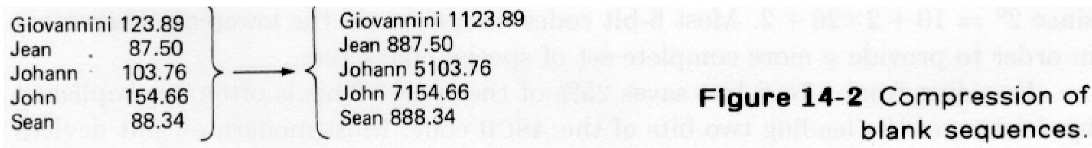

Figure 14-2 Compression of blank sequences.

Variable-Length Symbol Encoding As shown during decryption, using Table 12-3, characters do not occur with equal frequency. A technique, *Huffman coding*, takes advantage of this fact to compress data. The most frequent symbols are assigned to the shortest codes, and all longer codes are constructed so that short codes do not appear as initial bit sequences of the longer codes. No further marking bits are required to designate the separation of one character field and its successor.

Table 14-7 Algorithm for Constructing a Huffman Code

1	A list is initialized with an entry for each symbol, its frequency, and a space for its code.
2	Take the two entries of lowest frequency in the list; assign them the bits 0 and 1.
3	If an entry is a result of a previous combination, then catenate the new bit to the front of each code; otherwise initialize the code field with the bit.
4	Remove the two used entries from the list and insert a single, combined entry which has as frequency the sum of the two entries and attach the two used entries.
5	Repeat steps 2, 3, and 4 with the two entries which have now the lowest frequency, until all symbols have been processed.
6	The code values can be obtained from the tree of entries.

Using the frequency of single characters in English given in Table 12-3, the optimal space encoding begins as shown in Example 14-3.

Example 14-3 Construction of a Huffman Code

Next element	Encoding steps	Combined frequency	New code for previously encoded elements
fr(Z)=1			
fr(Q)=2	Z→ 0 Q→ 1	fr(ZQ)=3	
fr(J)=2	J→ 0 ZQ→ 1	fr(JZQ)=5	Z→ 10 Q→ 11
fr(X)=3	X→ 0 JZQ→ 1	fr(XJZQ)=8	J→ 10 Z→ 110 Q→ 111
fr(K)=5	K→ 0 XJZQ→ 1	fr(KJZQ)=13	X→ 10 J→ 110 ... Q→ 1111
fr(V)=10	V→ 0 KXJZQ→ 1	fr(VKXJZQ)=23	K→ 10 ... Z→ 11110 Q→ 11111
fr(B)=13			
fr(G)=14	B→ 0 G→ 1	fr(BG)=27	
fr(W)=15			
fr(Y)=15	W→ 0 Y→ 1	fr(WY)=30	
fr(P)=22	P→ 0 VKXJZQ→ 1	fr(PVKXJZQ)=45	V→ 10 ... Z→ 111110 Q→ 111111
fr(M)=27	M→ 0 BG→ 1	fr(MBG)=54	B→ 10 G→ 11
etc.			

The codes which are generated for this important case are given as Table 14-8. The encoding can also be represented by a tree as shown in Fig. 14-3. This tree construction can, in general, be used to minimize access to items of unequal frequency, as long as the access cost per link is equal.

Table 14-8 Huffman Code for English Text

E 100	D 11011	G 001111
T 000	L 11010	B 001110
O 1111	C 01110	V 001010
A 1110	F 01011	K 0010110
N 1100	U 01010	X 00101110
I 1011	M 00110	J 001011110
R 1010	P 00100	Q 0010111111
S 0110	Y 011111	Z 0010111110
H 0110	W 011110	

We find that for English text, given the frequencies of occurrence $fr(i)$ shown in Table 12-3, the average character length lc becomes

$$lc = \frac{\sum fr(i)\ len(i)}{\sum fr(i)} = 4.1754 \qquad \text{bits} \qquad 14\text{-}1$$

where $len(i)$ is the number of bits of symbol i in the Huffman code. This result can be compared with the $\log_2 26 = 4.70$ bits required for a minimal nonpreferential encoding of 26 symbols and the $19/4 = 4.75$ bits achievable with a grouping of four characters seen in Table 14-6.

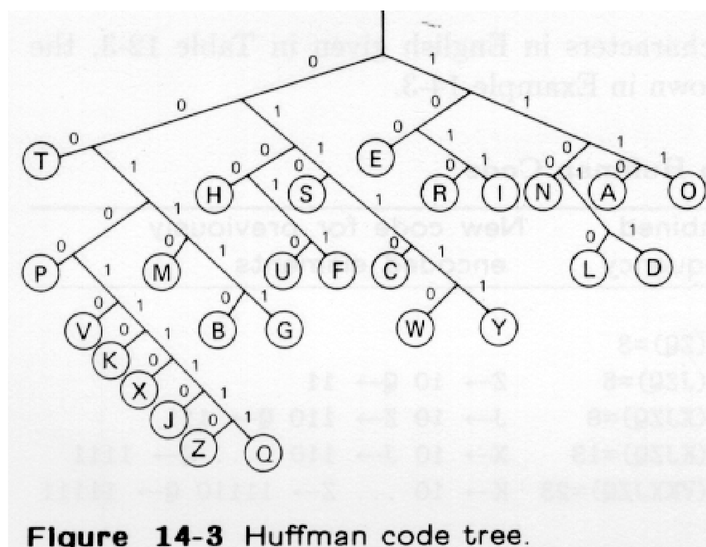

Figure 14-3 Huffman code tree.

Variable-Length String Handling An important factor in string compression is the support of variable-length strings. Names of individuals, organizations, titles of books, and references vary greatly in length, and to accommodate the maximum without truncation is rarely feasible using a fixed string length. A sampling of last names [SSA57] and a word list [Schwartz63] is shown in Table 14-9.

A marking scheme, as introduced in Fig. 2-11, is required. Most character codes provide a control character which is appropriate to terminate a character string. In the ASCII code (Fig. 14-1) `US` is in general appropriate. An alternative is the use of initial count field, although this imposes limits on the maximum string length. A count field of 8 bits limits strings to 255 characters, which is adequate for simple lines of text but is sometimes exceeded when formatting macros are included or when underlining is specified within the line. The use of a count field has computational advantages, since character strings move and catenation operations can be prespecified. When a `US` character is used, the string has to be scanned during processing.

Table 14-9 Length of Names and Words in a Dictionary

	Names (tokens)		Words (types)	
Length	%	Cum.%	%	Cum.%
5 or less	29.53	29.53	40.44	40.44
6	24.22	53.75	17.37	57.81
7	21.56	75.31	14.96	72.77
8	12.81	88.12	10.23	83.00
9	6.10	94.22	7.03	90.03
10	2.87	97.09	5.05	95.07
11	1.15	98.24	2.66	97.73
12 or more	1.76	100.00	2.27	100.00

Replacement Although Huffman codes are easily docoded left-to-right, their use is awkward on most machines. The facilities for handling characters are often much better than bit-oriented operations. Schemes to use *free characters* to replace frequent long strings can be easy and effective. Both unassigned and unused control and data characters can be free. Table 14-2 shows that especially 8-bit codes, bytes, have many free characters. It is possible that about 150 of these character codes will not occur in the text.

Since blank sequences are frequent, some, say 4, free characters can be assigned to represent two blanks, three blanks, four blanks, or eight blanks. If other characters are expected to occur in long sequences, this technique can be extended to those characters. If many of the characters are apt to be repeated, the free characters can be used to indicate two, three, four, or eight repetitions of the preceding character.

If certain words appear with great frequency, they can also be replaced by single free characters. Which words occur most frequently in a text depends greatly on the universe of discourse. Words such as

`THE OF A TO AND IN THAT IS IT FOR ON ARE`

are universally frequent, and hence are, in general, candidates for replacement by single character codes.

Zipf's law In natural text, unfortunately, the shortest words occur already with the greatest frequency. A rule, *Zipf's law* provides an estimate for the relationship between frequency and size in many situations. If n words $\mathrm{w}_i, i = 1 \ldots n$ are listed in

the order of their size, from smallest to largest, we can expect that their frequency of occurrence $fr(\mathrm{w}_i)$ will be

$$fr(\mathrm{w}_1) = c/1,\ fr(\mathrm{w}_2) = c/2,\ fr(\mathrm{w}_3) = c/3,\ \ldots,\ fr(\mathrm{w}_n) = c/n \qquad \text{14-2}$$

The constant c is adjusted to generate the appropriate total. Words of equal size have to be ordered according to the frequencies observed.

If the frequencies fr are expressed as fractions, then $c = H$, the harmonic number for n. Knuth[73S] shows that for n¿100,

$$H_n \approx \ln n + \gamma \qquad \text{14-3}$$

where γ or Euler's constant is 0.5772 . . . .

One can consider replacing longer but yet frequent words by shorter multiple letter combinations, in a sense intensifying Zipf's law by inclusion of unpronounceable combinations. In a stored database textbook the replacement of "␣database␣" by "␣db␣" could be profitable. The lexicons required to effect this translation can become quite large. This approach has been used in files of names. Zipf's law can also be applied to estimate the progression of the type-token ratio for words in a text, ranked according to their frequency of occurrence, or individual type-token ratio. Knuth[73S] shows how a better fit to Zipf's law can be obtained by adding a parameter to the divisor in the sequence, so that

$$fr(\mathrm{item}i) = \frac{c}{i^{1-\theta}} \qquad \text{14-4}$$

With $\theta = 0.1386$ the sequence expresses Heising's 80-20 rule stated in Sec. 6-1-6, and with $\theta = 0.0$ the sequence expresses the type-token ratio for the words from the text by Schwartz[63] used for Table 14-5 for the words ranked from 5 to 4000. Extremely high and low frequencies do not quite match.

Since (as discussed in Sec. 14-2-3) a limited number of last names of individuals accounts for many of the names in files [SSA[57]], it is possible to replace these names by shorter representations. These short names can be made up out of the 20 consonants, giving, for instance, 8000 three-consonant combinations for more than half the names in use. Use of two letters and a digit in an arbitrary position of the representation would avoid the rare but possible confusion with an existing name and provide 20 280 combinations, beginning with "`1AA`" and ending with "`ZZ9`". The names with extremely high frequency can be represented by combinations of two characters, "`1A`" to "`Z9`". Lexicons to control replacement of names or words can be generated by analysis of the file or of a sample of the file.

If a fixed-field encoding is desirable, some of the codes can be used to refer to an overflow file for names which could not be encoded. A format `Zaa`, for instance, could use the digits `aa` to point to records containing unencodable names.

Dynamic Replacement If adequate processor time at data-entry time is available, the selection of a lexicon to control replacement can be done dynamically. The lexicon which is specific to a record now has to be carried along with the record. An example from DeMaine[71] is shown in Fig. 14-4.

Compression of 40% to 60% is reported on strings ranging from 400 to 1500 characters using up to 10 patterns in the lexicon. The same technique is possible using arbitrary bit strings. In this case the record can contain numeric or string data, as well as results from previous compression algorithms.

A Huffman code could also be generated dynamically, but here the code tree would have to be appended to the record. The code tree is difficult to represent in a compact form. A general decoding program for variable character codes will also be larger and slower than a program oriented to a particular Huffman encoding.

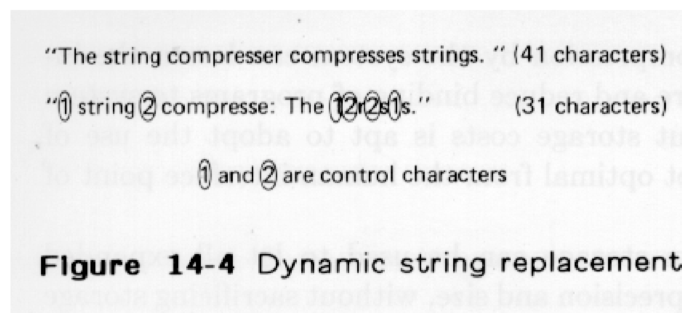

Figure 14-4 Dynamic string replacement.

14-3-3 Implemention of Compression

Compression schemes can be best evaluated by applying them to a representative sample of the data. Several parameters have to be measured. The simplest measurement is the reduction of storage requirements. Changes in data transfer time are another benefit which should be evaluated. The processing cost for compression and expansion can be significant and should not be neglected. Since data retrieval is more frequent than update, complex compression schemes which generate encoding that are simple to expand are preferable. Evaluation requires that the retrieval to update ratio be known.

More than one method can be used in combination with others. In that case, the order in which compression algorithms are applied will make a difference. Replacement should precede Huffman coding, for instance, but if replacement leads to a more uniform use of character codes, subsequent Huffman coding will lose its advantage.

Processing Point for Compression Compression or expansion can be performed at various points in the data acquisition, storage, and retrieval process. If compression is done soon after data entry and expansion is done just before information presentation, the volume of data to be handled throughout the system is less. Data elements which only control selection of other data elements will not be expanded at all, so that processing time for expansion will be saved. On the other hand, extensive usage of compressed data will require that all processing routines can manage compressed data. If computational use of the data requires expansion, the data elements fetched may be expanded many times during processing. Especially statistical-analysis programs tend to require data elements that are of consistent size and cannot cope with compressed elements.

The alternative is to assign data compression entirely to the file system. File systems can be equipped with compression algorithms through the use of database procedures or file-system escape procedures, so that compression and expansion becomes fully transparent to the user. Now the degree of compression performed can be adjusted by a database administrator according to storage versus CPU cost ratios applicable to a particular system or system era.

The automatic management of compression by the system can lessen the involvement of the user with the hardware and reduce binding of programs to system features. A user who is worried about storage costs is apt to adopt the use of representation techniques which are not optimal from the human interface point of view.

Compression of numeric values in storage can be used to let all expanded numbers used in processing be of equal precision and size, without sacrificing storage space. The use of fixed data-element sizes in processing programs can increase the sharing of standard programs.

Hierarchical files, through their multiple record formats and lack of array structure, tend to benefit less from compression than files which are formatted as tables. There are instances, as shown in Sec. 4-5, where the benefits of dense storage are a major factor in the decision to use a hierarchical approach. The availability of compression can reduce the importance of this particular factor.

Compression and File Design The compression of data has a significant impact on the file design:

Compressed files generate variable-length records. Updating of records in compressed files can cause these records to change size.

Since many current file systems do not support varying record lengths, the compression of data has been forced from the level of a file-system support option into the database system or into the user level.

The user without who is limited to fixed-lenghth records has also to create mechanisms to store the representation economically in addition to being burdened with the problem of finding an effective representation for the real-world data. The combination is frequently a difficult task for the user. There are important applications where the data are broken into many linked fixed-length records since variable-length records were not available.

In the opinion of the author this abdication of responsibility by file-system designers is the major reason why the use of file has remained a problem in computer systems. These same systems do provide compilers, editors, and libraries to attract users to their facilities. All these facilities provide a better match to the users processing concepts at the cost of some computing overhead. The provision of equally elegant file-support facilities is the next step.

BACKGROUND AND REFERENCES

The representation of knowledge by symbols is a basic human characteristic. Many of the ideas expressed in the initial section are from an excellent exposition by Feinstein[70]. Discussions on the topic are found in Brodie[81]. Pierce[61] provides background information and Reiter in Gallaire[78] considers the limits of encoding.

Coding conventions for computers can be commonly found in manufacturers' manuals. Loomis in Stone[75] summarizes codes used by computers to represent data. A state-of-the-art report of codes used in commerce is provided by McEwen[74].

The patterns of names and words have been studied; sources here are SSA[57], Schwartz[63], and Lowe[68]. SOUNDEX coding was introduced by Odell and Russel in 1918; it is described by Gilligan in McEwen[74]. O'Reagan[72] encodes verbal responses. Florentin[76] presents coding conventions for associative relations. Existing code conventions can also be found by looking at usage in available databases, a directory is Williams[82].

An extensive study and implementation of compression algorithms is presented by DeMaine[71] and abstracted in Codd[71]. The file system described in Wiederhold[75] includes compression. Bayer[77] places front-abbreviations into the B-tree. Numeric data files are considered by Alsberg in Hoagland[75]. Young[80] places extracted descriptors at higher levels and Eggers[81] places them into B-trees.

Hahn[74] and Martin[77] discuss character replacement. Huffman coding (Huffman[52]) is also presented in Martin[77], analyzed by Knuth[73S], and applied by Lynch[82]. Compression techniques are surveyed by Gotlieb[75]. Knuth[73S] points out the relation between Zipf's law (Zipf[49]), Heising's 80-20 rule (Heising[63]), and the data by Schwartz[63].

EXERCISES

1 Categorize data representations appropriate to the following data:
Car license-plate numbers
Miles driven per year
Driver-license numbers
Type of driver's license
Number of traffic violations
Severity of traffic violations
Location of accidents
Objects destroyed or damaged
Cost of repair and reconstruction

2 Locate the character-coding table for the computer you are using. How many data and control characters are available? Are there characters which are usable for compression algorithms? Are all characters usable in files, in terminal output, or on the printer? Can they all be generated from data input without special processing?

3 Locate a standard coding table, and describe size, format, density, susceptibility to errors, limitations, suitability for computing, and human-interface effectiveness.

4 Convert your name to Soundex. Do this also for misspellings of your name you have encountered.

5 Compress floating-point numbers for the case that only three-digits precision is required. Watch the normalization!

6 Test three algorithms for string compression on a body of text. This exercise can also be assigned as a competition. Discuss the result and the range of its applicability.

7 a Design a compact code for dates within this century.

b Design a compact code for the time of day adequate for computer time accounting.

c Design a compact code for time adequate for the resolution of synchronization problems in your computer.

8 Discuss when the abbreviation scheme used by VSAM (Sec. 4-3-3) is effective, and when it is not.

9 Locate another standard code, similar to those shown in Table 14-3, and describe its domain and utilty for databases.