Database Compression

Mark A. Roth and Scott J. Van Horn
Department of Electrical and Computer Engineering*
Air Force Institute of Technology
mroth@afit.af.mil

Despite the fact that computer memory costs have decreased dramatically over the past few years, data storage still remains, and will probably always remain, an important cost factor for many large scale database applications. Compressing data in a database system is attractive for two reasons: data storage reduction and performance improvement. Storage reduction is a direct and obvious benefit, while performance improves because smaller amounts of physical data need to be moved for any particular operation on the database.

We address several aspects of reversible data compression and compression techniques:

- general concepts of data compression;
- a number of compression techniques;
- a comparison of the effects of compression on common data types;
- advantages and disadvantages of compressing data; and
- future research needs.

Reversible and Non-reversible Compression

Non-reversible data compression, also known as data compaction, reduces the size of the physical representation of the data, but preserves only a subset (that which is considered relevant) of the original information.[9] For example, in some data types leading zeros or trailing blanks may be considered irrelevant information and discarded during compression. The main disadvantage is that, after decompression, the original data representation can never be reconstructed.

In reversible data compression, all of the information is considered relevant, and decompression will recover the original data representation. This type of compression can be divided into two separate techniques: semantic independent and semantic dependent. Semantic independent techniques can be used on any data type, with varying degrees of effectiveness, and don't use any information regarding the content of the data. Semantic dependent techniques depend, and are based, on the context and semantics of the data.[9] All the data compression techniques discussed here are of this latter type.

Types of Data Redundancy

Data redundancy occurs when strings of data patterns are predictable, and therefore carry little or no "new information". Finding, and exploiting, this redundancy in databases is the basis for any data compression technique. There are four basic types of redundancy present in databases,[15] with some databases having a mixture of types.

- *Character Distribution.* In a typical data string, some characters are used more frequently than others. For example, in the eight-bit, 256 ASCII character set, nearly three-fourths may never be used in a specific file. Consequently, an average of only two bits, out of an eight-bit data packet, might be used, equating to a possible 75% space savings. In the English language, characters usually occur in a well-documented distribution, with the letter *e* and "space" being the most popular. By determining the distribution of data values, specific distribution codes can be developed that allow frequently used characters to be coded with a smaller code, and less frequently used characters to be coded with a larger, more elaborate code, all with a smaller average storage requirement.

- *Character Repetition.* Within a data string, repetitions of a single character can occur. The string can usually be compacted into two fields, coding the repeating character symbol into one field, and the number of times the character is repeated in the other. Such strings are infrequent in text files, occurring mostly as blanks. But in scientific and numerical databases, these repeating characters are much more prevalent, composed mainly of strings of blanks or zeros in unused data fields, or in higher-order numeric fields. In graphical data, these strings are composed mostly of sets of large homogeneous spaces.

*AFIT/ENG, Bldg 642, 2950 P St, Wright-Patterson AFB, OH 45433-7765

□ *High-Usage Patterns.* Similar to character distribution but with sequences of characters, patterns will reappear with relatively high frequency, and thus can be represented with fewer bits. For instance, in textual data, a period followed by two spaces is more common than most other three character combinations and therefore could be coded to use fewer bits. Numeric fields contain only sequences of digits, with no letters or special symbols intermixed. These could be encoded at less than four bits per digit, rather than the five to eight bits needed for text.

□ *Positional Redundancy.* Certain characters, or patterns, could appear consistently at a predictable place in each block of data. This occurs, for example, in raster data or any type of preformed data structure where a line or graphical representation occurs in the same position each time. Here a compression code could be developed using only the redundant character, or symbol, and its location(s).

Compression Performance Measurement

The most popular method of measuring the performance of a compression technique is the compression ratio,[13] CR:

$$CR = \frac{\text{data size without compression}}{\text{data size with compression}}$$

where the unit of data depends on the application. It could be a single character, a sequence of characters, or an entire file. If the compression is on more than a single character, then CR represents an average value per character. Given the compression ratio, the expected data size reduction is given by

$$REDUC = \left(1 - \frac{1}{CR}\right) * 100\%$$

Data Compression Effectiveness

As with any compression technique, the effectiveness of compressing the data not only depends on the system, but depends on the characteristics of the data in the system. The following data characteristics offer an excellent environment for the application of data compression techniques.[2]

□ *Sparseness.* In sparse databases, fields tend to have large clusters of zeros, blanks, or missing data indicators. Sparseness is the most important property in determining the overall compression percentage that could be achieved in the database.

□ *Value Distribution.* In numeric databases with a wide range of values, numbers tend to have a large distribution, usually skewed toward the lower end. Thus many values can be stored using, on average, fewer bits than the maximum bit length in the distribution.

□ *Joint Attributes.* In databases with many key values, the repetition of key values in data tables usually takes the form of a cross product, and impose a large storage overhead that is multiplicative in nature.

□ *Frequency.* In large textual databases, characters with alphanumeric attributes do not occur with the same frequency. This can be taken advantage of with redundancy codes.

Data Compression Techniques

A wide variety of techniques are available today for compressing data. Some of the more popular techniques are discussed next.

Abbreviations

Abbreviations[13] tend to fit into a data encoding category, rather than a data compression category, since conversion takes place either before the database is loaded to disk from memory, or during data reorganization. The data reduction obtainable using abbreviations can be dramatic, with typical reductions of 50-90%. However, the encoding and decoding processes require an extremely large "look-up" storage table if the number of possible data items is quite large. Sometimes, the decoding is even done manually with the tables in hard copy.

Although compression ratios can be very good with abbreviations, great care must be taken when different categories of abbreviation codes are placed into a single table (e.g., CANADA (CA) and CALIFORNIA (CA) both have the same two-character abbreviation). Also, abbreviations can only be used when the data can be predetermined, or is static, so that the abbreviations can be made unique. On the good side, unlike most compression techniques, abbreviations are limited only by the imagination of the designer, so there are an endless number of codes which could be developed. Some examples include standard abbreviations such as CIA, DOD, and NASA; arbitrary codes like C2 for Chicago and O1 for Omaha; or the numerous date coding methods such as 6.15.68 for 15 Jun 68, or Julian dates.

Null Suppression and Run-Length Encoding

Null suppression is a generic term used for techniques that suppress either zeros or blanks. The

technique takes advantage of data in which zeros or blanks are predominant. Generally, null suppression is simple, and can be easily implemented as a generic routine for use with many different types of data files.[10] On the other hand, null suppression does not usually achieve as high a compression ratio as some other techniques (reductions of 20-45%). Two basic methods are used for compressing and storing the data:

□ Represent sequences of zeros and blanks by a special character, followed by a number that indicates the length of the sequence.[13] Because of this, sequences of one or two characters are not efficiently represented in this manner and are normally left unchanged.

Original Data:
DUNNbbbbbbCbAbb450000b55
Compressed Data:
DUNN#6CbAbb45@4b55

□ Use a bit-map[10] appended to the beginning of each data record, and ending on a word boundary. When a unit of data is fixed, such as a word or byte, units containing all null or zero data values can be denoted by a zero-bit, and units containing nonnull or nonzero values can be denoted by a one-bit. Therefore, each bit in the bit-map represents the fixed sized data unit chosen for the application. These bits and their positions correspond exactly to the positions of the characters and words within the data record itself. Also, unless a field type indicator and length is appended to the bit-map, compressions will be limited to either nulls or zeros, and not both, within a data record. The following example assumes half-word data units:

Original Data:
DUNNbbbbCb45bbbb
Compressed Data:
Bit Map: 11001100
Text: DUNNCb45

A similar technique is the idea of run-length encoding.[2] Run-length encoding is very effective in picture processing, and in databases where there are long sequences of repeated zeros or missing values. Simply, run-length encoding replaces sequences of identical values by a count field, followed by an identifier for the repeated value. The count field must be flagged so that it can be recognized from the other data values. And as above, the selected sequence must have enough repeated values to warrant its replacement by the count and character fields.

Pattern Substitution

In this technique, an algorithm scans the entire database, or text file, looking for common patterns of two or more characters occurring frequently, and substitutes an unused pattern for the common long one.[13] At the same time, a substitution dictionary ("look-up" table) is created and/or updated. Efficiency can be a serious problem in this technique since many comparisons must be made between the data and the dictionary to identify a pattern. Additionally, patterns can be of various lengths and may be subsets of other patterns. Pattern substitution generally achieves a greater compression ratio than null suppression since patterns, in addition to strings of zeros and blanks, contribute to the compression. Here compressions are on the order of 35-50%.

Original Data:
CED3690000BB52X0
CED3700000BB86X0
Compressed Data:
#369?/52X0
#370?/86X0
Patterns:
= CED ? = 0000 / = BB

Differential Compression

Differential compression coding[2,11] involves the replacement of a character sequence with a code value which defines its relationship to either a previous similar character sequence, or a specific sequence. Because of this, compressions of this type can only be used in applications when data are of uniform size and tend to vary relatively slowly, such as telemetry data. Compressions of this type of data can be as high as 98%. An example of this could be the following series of names or integers:

Original Data:
Johnson, Jonah, Jones, Jorgenson
1500, 1520, 1600, 1550, 1570, 1610
Compressed Data:
(0)Johnson, (2)nah, (3)es, (2)rgenson
1500, 20, 80, -50, 20, 40

Statistical or Variable-Length Encoding

Perhaps the most common and popular compression method is to convert fixed-sized pieces of data into variable-length symbols,[13] otherwise known as minimum-redundancy coding or Huffman coding.

This method first gained prominence with the publication of David Huffman's classic 1952 article.[4]

The standard Huffman encoding procedure[15] defines a way to assign codes to data such that each code length, in bits, is approximately

$$\log_2(1/\text{symbol probability})$$

where *symbol probability* is the relative frequency of occurrence of a given symbol, expressed as a probability. For example, if the "blank" character is used one-eighth of the time, then the blank character is encoded into a three-bit symbol; if the character *Z* is found to occur only 0.1% of the time, it is represented by ten bits. A portion of a Huffman code table[6] for a specific commercial database follows:

Symbol	Probability	Code Value
0	.555	0
1	.067	1000
A	.032	10101
E	.009	110101
X	.0012	1001111111
Q	.0003	101111111101

In normal use, the size of input symbols is limited by the size of the translation table needed for the compression.[15] The table is basically a "look-up" table that lists each input symbol and its corresponding code. If a symbol is represented by the common eight-bit byte, then a table of 256 entries is sufficient. Such a table is quite economical in storage costs, but because of the table entry limitation, it can limit the degree of compression achieved.

Although Huffman coding is relatively easy to implement, the decompression process is very complex, and is the major problem with the technique.[15] First of all, the length of each code to be read, for decompression, is not known until the first few bits are interpreted. Second, the basic method for interpreting each code, is to interpret each bit in sequence, and then choose a suitable translation from the table according to whether the bit is a one or zero, in other words, a binary tree operation. If only one bit is interpreted wrong, the whole string will not be decoded correctly. In general, decompression with Huffman codes gives a cost and performance disadvantage whenever the data volume is high.

A second minor problem with Huffman encoding[15] is that the frequency distribution for the set of input symbols must be known. For text files, in the English language using single-character symbols, the distribution is well known and relatively stable. But, the distributions for specialized data files can vary widely. The common solution in these cases is to analyze each data block individually to determine the character distribution unique to that block. In this scenario, two passes over the data are needed; one to count characters, compute the codes and build the table, and one to encode. Because the table is built for specific blocks, the table must also be carried with each compressed data block. This approach is good if high transfer rates through the compressor are not required, and if the data blocks being compressed are very large compared to the translation table.

The Huffman encoding technique really only pays off with a skewed character distribution.[16] For example, if all the characters in the English alphabet were used equally, Huffman encoding gives a mean number of bits per character of 4.70 bits. Again, using the English alphabet, if the distribution of letters is what is actually used, Huffman encoding gives a mean number of bits per character of 4.17 bits. Generally, compressions have consistently been reported in the 40-55% range, with some slightly higher. Much work in the area of Huffman encoding has been done and there are many variations, mostly through algorithms based on enumerative and combinatorial techniques, and using splay trees[5] or heaps[11] instead of binary trees for decompression.

LZW Compression

Unlike the Huffman technique, the LZW compression technique,[15] named after the originators: Lempal, Ziv, and Welch, converts variable-length strings into fixed-length codes. The symbol strings are selected so that all have an almost equal probability of occurrence. Consequently, strings of frequently occurring symbols will contain more symbols than a string having infrequent symbols. An example conversion table might be as follows:

Symbol String	Code
A	1
AB	2
AND	4
Z	5
0	16
0000	19

Restricted Variability Codes

Since almost all computers are word, rather than bit oriented, the length variability of the Huffman

code for different characters can be a drawback. We can avoid this by using restricted variability codes.[9] This advantage though, is usually achieved at the cost of the less efficient use of bits in representing the characters.

As an example, the so-called 5/10 variability code can represent up to 63 characters. In this technique, the 31 most frequent characters are represented by the first 31 representations of a 5-bit code. The 32nd position is usually used as a flag character to indicate that the actual character is encoded in the last five bits. In other words, the 32nd position and the first five bits, plus the second five bits, provide encodings for the less frequent characters. Thus, ten bits are used to denote the 32 less frequent characters. A 2/4 restricted variability code using "11" as the flag character, and assuming that the characters $\{A, B, C\}$ are more frequent than the characters $\{D, E, F, G\}$, the coding is as follows:

Character	Code
A	0000
B	0001
C	0010
D	1100
E	1101
F	1110
G	1111

Arithmetic Coding

A relatively new technique, arithmetic coding[17] represents a message by an interval, or range, of real numbers between 0.0 and 1.0. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify the interval grows. Successive symbols of the message reduce the size of the interval in accordance with symbol probabilities generated by the technique. The more likely symbols reduce the range by less than the unlikely symbols, and therefore add fewer bits to the message.

As an example, before anything is transmitted, the initial range for the message is the interval $0 \leq x < 1$. As each symbol is processed, the range is narrowed to the portion of it allocated to the symbol. For example, suppose an alphabet is defined as $\{a, e, i, o, u, !\}$, and the corresponding probabilities and ranges are as follows:

Symbol	Probability	Range
a	.2	[0.0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

As the message *eaii!* is read, the technique first determines the range it needs to start, initially this range equals 1.0. Then, a new high value (HV) and a new low value (LV) for the range is determined as each symbol is read, as follows:

$$Range = HV - LV$$

$$HV = LV + (Range \times SymbolHV)$$

$$LV = LV + (Range \times SymbolLV)$$

This procedure is repeated for each character in the message, until the message ends. A special EOM character is then encoded as the last symbol in the message to indicate message termination for the decoder. The *eaii!* message builds up in the following way (*!* is the EOM symbol):

After Reading	New Range
Initial	[0.0, 1.0)
e	[0.2, 0.5)
a	[0.2, 0.26)
i	[0.23, 0.236)
i	[0.233, 0.2336)
!	[0.23354, 0.2336)

Decoding works in just the opposite way. Looking at the final range, [0.23354, 0.2336), it can be seen immediately that the first character was an *e*, since its range lies entirely within the range computed for *e* in the first table above. From here the technique computes a new range, as follows:

$$NewHV = \frac{HV - SymbolLV}{SymbolHV - SymbolLV}$$

$$NewLV = \frac{LV - SymbolLV}{SymbolHV - SymbolLV}$$

The decoder next determines what symbol range surrounds these new values, and thus the next symbol is decoded. This continues until the decoder reads the entire message, stopping at the EOM symbol. For example, back to the message above:

After Reading	Sym	New Range
[0.23354, 0.2336)	e	[0.1118, 0.112)
[0.1118, 0.112)	a	[0.559, 0.56)
[0.559, 0.56)	i	[0.59, 0.6)
[0.59, 0.6)	i	[0.9, 1.0)
[0.9, 1.0)	!	EOM

Although there is widespread belief that the Huffman coding technique cannot be improved upon,[17] there are those that argue that arithmetic coding is superior in many respects to the Huffman technique. This may, in some respects, be due to the fact that arithmetic coding is relatively new, and not well known. Bottom line, arithmetic coding does represent information at least as compressed as the Huffman technique.

Hardware

Only recently has there been much work done with hardware support for data compression. To a certain extent this is due to the opinion of many database designers that compression is of limited use, and the sophisticated algorithms used in many data compression techniques.

There are several possible formats that hardware support can take.[2] One is to include data compression, encryption, and some key searching capabilities in an advanced disk controller. Another technique uses a microprocessor-assisted system to offload the process of data compression from the front-end computer to a structure of minicomputers, each performing a specific data management function. When the host computer needs data, it sends the request to this system, which analyzes the request, and sends subrequests to the appropriate minicomputers. The minicomputers do the actual disk reads, data decompression and compression, and forwarding of the data to the requesting computer.

Several researchers are using VLSI technology to design special purpose hardware for real-time compression and decompression of data. Neural networks[3] and memory based[7] architectures are joining standard algorithm-to-silicon techniques.[1,14] Performance evaluation on these implementations is not complete and needs further research.

Compression Results

As stated before, the performance of any compression technique is expressed as a ratio between the number of bits representing the message before and after compression. Below are some average results of compression for a variety of data types using the LZW compression technique.[15]

□ *English Text.* English text results were reasonably good for simple text, with reductions at around 45%. Many word processing files, however, compressed better when they contained figures, formatted data, or graphical data. Long individual documents did not compress as well as groups of short documents, indicating that redundancy is not due very much to content.

□ *Floating Point Numbers.* Data consisting of floating point numbers look pretty much like random numbers, especially the fractional part, so they compress rather poorly at around 10%. The exponent does compress somewhat better when most of the numbers have the same magnitude. In some extreme cases, floating point data actually expanded when the exponents varied widely.

□ *Formatted Scientific Data.* This type of data, composed mostly of integers, tended to compress rather well at around 50%.

□ *System Log Data.* Information describing past system activity is mostly formatted integers, and therefore compressed at around 65%. This type of data tends to be in a tightly packed, fixed-length format, so the compression achieved is due to the null fields and repetitions in the data values.

□ *Programs.* Source code tended to be compressed by a factor of better than two. In general, it can be compressed better than text because the words are frequently repeated, and blanks are prevalent. Highly structured programming yields source code with greater compressions of around 70%. Object and executable code, on the other hand, consists of arbitrary bit patterns and does not compress as well.

□ *Mixed Character Sizes.* Sometimes data is recorded in nonstandard character sizes. When eight-bit ASCII symbols are stored in nine-bit bytes, the observed compression is just slightly greater than eight-bit symbols stored in eight-bit bytes. This is because, the length of the compressed data depends on the number of symbols encountered rather than their encoding. Six-bit data compressed surprisingly well when compressed on a nine-bit basis because repeated patterns, such as blanks, are word aligned and have the same pattern on each occurrence. As a side note, when nine-bit data was compressed as if it were eight-bit characters, the results were poorer but not unacceptable. Basically this all says, that compression can be effective even when dealing with mixed character sizes.

□ *Recovery Tapes.* Computer backup or recovery tapes, containing a cross-section of data used in system and user files, have been compressed to slightly less than 50% for scientific data to slightly more than 50% for program development data.

Data Compression Advantages

Data compression techniques can have a positive cost effect not only on the storage and transfer of data, but also in the areas of database security, backup and recovery procedures, as well as enhancing database performance. Bassiouni[2] describes the following advantages of data compression:

□ *Storage.* Of course the most obvious advantage of data compression is that of reducing the storage requirements for the information, and at the same time, increasing the capacity of the storage medium. Although advances in technology continuously reduce the costs and increase the capacities of the physical data storage medium, interest in and the need for data compression research has actually slowly and steadily increased. The main reason for this is the explosive growth of data storage and data processing applications which continue to outpace advances in technology.

□ *Data Transfer.* Since compressed data use smaller number of bytes, transfer of compressed information requires less time, resulting in a higher transfer rate. In I/O bound systems, this faster transfer rate from disk to main memory or vice versa, can reduce transaction turnaround time and user response time. In addition, since this will also reduce I/O channel loading, the CPU can process many more I/O requests and thus increase channel utilization. But, more important, is compression's role in reducing the costs of transmitting tremendous amounts of data over long-distance communication links.

□ *Data Security.* This advantage is actually a side-effect of data compression. To use or interpret compressed data, it has to be restored to some uncompressed form. To do this a decoding algorithm, along with a decoding key (a table, tree structure, etc.), must be available. Even with the same algorithm, different files may use different decoding keys. These keys can be kept separate from the compressed files to restrict access to the corresponding data. But, because data security is a by-product, data compression can only enhance data security, and cannot, by itself, provide the high level of security needed by some applications.

□ *Backup and Recovery.* In most systems, the basic backup/recovery scheme is based on keeping backup copies of the database files, along with the audit trail, on tape. Backup copies of the database could be kept in compressed form, reducing the number of tapes required to store the data and reducing the time of reading from, and writing to, these tapes. This will also reduce the time needed for recovery, and allows for the reduction of the time span between checkpoint tapes.

□ *Performance Enhancement.* In some applications, data compression can lead to other types of system performance improvements. For example, in some index structures it is possible, through compression, to pack more keys into a given index block. When the database is searched for a given key value, the key is first compressed and then compared against the compressed keys in the index block. The net effect is that fewer blocks have to be retrieved, and thus the search time is reduced. Similar arguments can apply for the buffers used to store subsets of the database in main memory. By compressing records, the effective capacity of the buffer increases, and the probability that the key record will be found also increases.

□ *User.* When properly implemented, data compression and decompression processes can be made totally transparent to the user.[9]

Data Compression Disadvantages

Although data compression has some very enticing advantages, it does introduce several problems. Bassiouni[2] describes the following disadvantages of data compression:

□ *Process Overhead.* The overhead caused by the compression process to encode and decode the data is one of the most serious disadvantages of data compression. In some applications this overhead can be so great that it discourages any consideration for applying compression techniques. In stable, primarily read-only databases the expansion process has greater impact on the performance than the compression process. But, for most databases, which are quite volatile, the encoding process is slower and more complicated than the decoding process. Note, however, that until recently most data compression has been implemented with software. With the advent of VLSI technology, many techniques become open to complete or partial hardware implementation, thus reducing the process overhead dramatically.

□ *Data Property Disruption.* Because these techniques manipulate the actual data, many data

compression techniques can, and do, disrupt data properties that may be important for some applications. For example, by not preserving the lexical order of compressed data, efficient sorting and searching schemes may become inapplicable. In some cases, the data must be expanded first and the search done against the original data.

□ *Portability.* As in many systems, software implementations of data compression techniques are coded in assembler languages for efficiency for a particular machine, and since there are not enough well-defined standards in the area, data compression utilities are not completely portable or easily modifiable.

□ *Output Length.* The size of a compressed record is usually unknown in advance and depends on the compression technique used, and the contents of the record. For example, different records having the same length may produce compressed records with different sizes. This uncertainty of the output length creates problems in space allocation planning and in updating records within compressed files. In addition, some search and access methods become more difficult to implement, as the length of the key field may vary.

□ *Reliability.* By reducing redundancy in data, data compression techniques also reduce the ability to recover from data errors. For example, a single bit error in the Huffman technique, will cause the decoder to misinterpret all subsequent bits, producing incorrect output. This problem, at least on the hardware side, is somewhat alleviated by available reliable disk storage and communications technology.

□ *Decoding Keys.* As seen above, many data compression techniques create large numbers of decoding keys and tables that are necessary for correct expansion of compressed code. These keys and tables produced by the compression techniques run opposite to the main purpose of data compression, that of reducing storage requirements. But in most cases, the space savings of the main data far exceeds that of the added space needed by the keys or tables.

Data Compression Acceptance Factors

Since compression techniques consistently reduce data storage and provide a number of other benefits, it is surprising that they are not more widely used and accepted. Three facts can help to explain this:[11]

□ *Misunderstood Compression Limits.* Database designers typically underestimate, through lack of knowledge, the amount of compression which is possible for a given database. Also, the consequences and advantages of data compression are not fully appreciated.

□ *System Complexity.* Data compression adds a layer of complexity to the design, implementation, and operation of a database system. Designers are reluctant to accept the additional complexity without clear and substantial benefits upfront.

□ *Mathematical Mystique.* Much of the available literature "talks around" individual data compression techniques, surrounding them in a mathematical mystique. Designers will usually avoid those areas in which they feel uncomfortable.

Further Research

Although effective data compression techniques have been around for 40 years, there are still several areas in which research is lacking. Further investigation should be accomplished to make data compression a more viable and less mysterious tool to the database designer. Bassiouni[2] describes some of the more important areas as follows:

□ *Technique Comparison.* Although literature on data compression has waned in the past few years as memory has grown cheaper, it still is rich and growing. Despite this, little has been done on comparing the different compression techniques available, or identifying the best technique for a given environment. A benefit of comparing methods is the possible integration of a number of appropriate compression techniques applied to subsets of the database, instead of just one technique which works marginally on the entire database. Another benefit may be the combination of two or more methods in a single algorithm. For example, LZW compression has been successfully combined with arithmetic coding to gain compression ratios superior to either of the techniques used alone.[8]

□ *Data Transmission.* As we depend more and more on computer generated information, we also rely more and more on the transmission of the generated data. More attention needs to be given in designing better, or just using and modifying existing data compression algorithms to reduce the cost of data transmissions. In particular, these algorithms need to have a very low tolerance to propagate errors, and to allow for delays in storage before and/or after the data are packaged for transmission.

□ *Hardware Techniques.* Little work has been done in hardware techniques for data compression. Generally, work needs to be started in comparing the costs and performance of the different techniques known, and then identifying the best methods suitable for hardware implementation.

□ *Metadata Compression.* Metadata, or data on data, files in databases can be quite large. Although metadata compression can use the standard techniques, efficient and fast access to metadata is very important, since the metadata is central to the accessing, interpreting, and displaying of the basic information. Research needs to be focussed on the effects of system performance of compressing metadata, such as processing time to expand metadata, relative size of metadata to the basic data, and number of accesses needed for each access of the basic data.

□ *Algorithm Development.* Further work in the development of methods to achieve higher compression, faster access time, and more efficient ways to operate on compressed data, can do nothing but improve the field.

Conclusion

In general, the most common motivation for database compression is a storage problem or a design requirement. But a simple technique which offers just the basic data compression necessary for the application is often selected, instead of combining several methods, or selecting a more complex compression method which would offer more advantages than just the compression of the data. In addition, data compression may not be, and in some cases is not, desirable. The benefits associated with data compression of a specific database are affected by many variables: the size of the database, the amount and type of redundancy in the database, the type and frequency of retrieval and update requests, and the efficiency and complexity of the compression technique. All these factors must be considered before any decision is made on which technique(s) to use.

REFERENCES

1. Anderson, Robert, et al. "A Very High Speed Noiseless Data Compression Chip for Space Imaging Applications." In Storer and Reif[12], 462.

2. Bassiouni, M. A. "Data Compression in Scientific and Statistical Databases," *IEEE Transactions on Software Engineering*, *SE-11*(10):1047–1058 (October 1985).

3. Fang, Wai-Chi, et al. "A Neural Network Based VLSI Vector Quantizer for Real-Time Image Compression." In Storer and Reif[12], 342–351.

4. Huffman, David A. "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, *40*:1098–1101 (September 1952).

5. Jones, Douglas W. "Application of Splay Trees to Data Compression," *Communications of the ACM*, *31*(8):996–1007 (August 1988).

6. Martin, James. *Computer Database Organization*. New Jersey: Prentice-Hall, Inc., 1977.

7. Mukherjee, Amar, et al. "Multibit Decoding/Encoding of Binary Codes Using Memory Based Architectures." In Storer and Reif[12], 352–361.

8. Perl, Yehoshua, et al. "The Cascading of the LZW Compression Algorithm with Arithmetic Coding." In Storer and Reif[12], 277–286.

9. Reghbati, H. K. "An Overview of Data Compression Techniques," *Computer*, 71–75 (April 1981).

10. Ruth, Stephen S. and Paul J. Kreutzer. "Data Compression for Large Business Files," *Datamation*, 62–66 (September 1972).

11. Severance, Dennis G. "A Practitioner's Guide to Database Compression—A Tutorial," *Information Systems*, *8*(1):51–62 (January 1983).

12. Storer, James A. and John H. Reif, editors. *DCC '91 Data Compression Conference*, IEEE Computer Society Press, 1991.

13. Teorey, Toby J. and James P. Fry. *Design of Database Structures*. New Jersey: Prentice-Hall, Inc., 1982.

14. Venbrux, Jack and Norley Liu. "A Very High Speed Lossless Compression/Decompression Chip Set." In Storer and Reif[12], 461.

15. Welch, Terry A. "A Technique for High Performance Data Compression," *Computer*, 8–19 (June 1984).

16. Wiederhold, Gio. *Database Design*. New York: McGraw-Hill Book Co, 1983.

17. Witten, Ian H., et al. "Arithmetic Coding for Data Compression," *Communications of the ACM*, *30*(6):520–540 (June 1987).