Edgar H. Sibley
Panel Editor

A general-purpose data-compression routine—implemented on the IMS database system—makes use of context to achieve better compression than Huffman's method applied character by character. It demonstrates that a wide variety of data can be compressed effectively using a single, fixed compression routine with almost no working storage.

DATA COMPRESSION ON A DATABASE SYSTEM

GORDON V. CORMACK

Compressing information in a database system is attractive for two major reasons: storage saving and performance improvement. Storage saving is a direct and obvious benefit, whereas performance improvement derives from the fact that less physical data need be moved for any particular operation on the database.

To achieve these benefits, general data-compression and expansion programs were added to IBM's "Information Management System" (IMS). These routines are currently in use in production databases at many installations (see footnote, page 1342). This article provides a description of the design, implementation, and use of these methods that should be of use to anyone wishing to add a compression utility to a similar system.

SYSTEM CONSTRAINTS

Data in an IMS database are stored as a set of segments (records). Although any particular database may contain segments of many different formats, in general, a segment is the concatenation of one or more fields of information, with each field having a specific data type.

The IMS system makes provision for a data-compression exit to be invoked whenever a segment is stored [3], and for a complementary expansion exit to be called whenever a segment is retrieved. Thus, compression and expansion exits become an integral part of the IMS system and are therefore heavily constrained. In this scenario, it is inconvenient for exit routines to have to determine any information about the format of a segment being processed, such as where the field boundaries are or what type of data the fields contain. It is also prohibitively expensive for the exits to obtain *any* local storage other than the 18-word save area passed by IBM linkage conventions. Furthermore, segments may be presented by the system in any order, so that any adaptive coding scheme that builds a compression strategy dynamically [1, 7] can use only the information contained in a single segment. The fact that, in IMS, segments tend to be short (less than 100 charac-

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant A5485.

ters), coupled with the lack of local memory, effectively precludes the use of such schemes.

Finally, because the exit routines become part of the IMS system and are loaded into common storage, a proliferation of different exits is undesirable: One pair of exits must apply to a very wide variety of data. Indeed, each exit should be able to handle all possible character sequences. Although many ad hoc compression methods take advantage of unused characters in the collating sequence, an assumption that some characters are unused is unacceptable for a built-in routine.

All in all, these constraints are not unusual; they are similar to those that would be encountered when attempting to provide compression within any I/O system.

THE METHOD OF COMPRESSION

The compression methods appearing in the literature (surveyed by Reghbati [4], Schuegraf [5], and Severance [6]) range from ad hoc techniques, like replacing strings of blanks by a single character, to general techniques like Huffman's [2] that, subject to some constraints, finds the optimal coding for an arbitrary string. As a rule, the ad hoc techniques perform badly on data not anticipated by their creators, whereas the more general techniques adapt better, provided the variable information used for compression can be re-created.

The method presented here is based on modified Huffman codes. In general, Huffman's algorithm, applied character by character, gives the best possible compression subject to the following constraint: that each character of the text has a unique compressed code, and that the compressed version of a string is constructed as the concatenation of these codes.

In Huffman's algorithm, the Huffman code for a particular character is selected so that its length is as close as possible to $-\log_2 p$, where p is the probability of occurrence of that character in the text. However, this choice cannot be made appropriately if p varies radically depending on the position of the character in the text, and such a situation often exists in the segments to be compressed here: That is, the probability of occurrence of a character depends on the data type of the field in which the character occurs. It is therefore important to use a different set of character probabilities for each different *type of field* that may appear in the database.

Determining Field Types

As mentioned earlier, in the IMS system it is difficult to determine any information about the format of the data to be compressed: Gleaning this information would require passing it to the system via the programmer, a tedious and error-prone process. However, it is possible to take a broad sample of the data to be compressed in advance and base the compression method on this sample.

However, Huffman coding is adversely affected by this approach in that compression deteriorates as more data types occur in the sample. The best that can be done is to provide a compromise coding scheme, weighted by the fraction of each data type found in the sample. Such a compromise achieves suboptimal compression, which deteriorates further if the mixture of types differs from that in the sample.

It therefore becomes desirable to be able to identify fields of differing types, and to provide a different Huffman coding for each type of field. In addition to programmer-defined fields, there are also implicit fields in database records. For example, in a mailing address, there are strings of alphabetic text as well as numbers, which can be regarded as separate fields. When fixed-length fields are padded using a fill character (typically a blank), the filled portion may also be regarded as a separate field.

Because field-type information was not readily available in our case, an automatic method of detecting fields had to be devised. To do this, we assume that the sets of frequent characters for different field types have small intersections, and then guess the type of field by examining a single character. Since fields tend to be several characters long, we assume that the subsequent character also belongs to a field of the same type and compress it accordingly. When a character appears that is uncommon for the current field type, but common for another type, we assume that a new field has been encountered and begin compressing according to this new type.

This automatic field detection is easily implemented by first partitioning the set of possible characters according to the type to which each character belongs. In the IMS system, we chose as sets *letters of the alphabet, numeric digits, packed decimal digit pairs, blank,* and *other.* For each set of characters, the frequency distribution of *all* characters *following* a member of that set is determined, and a (modified) set of Huffman codes is constructed for each frequency distribution.

Once the automatic-field-detection scheme is in place, the compression of a record proceeds as follows: The first character of the segment is compressed using one of the sets of codes (the choice is arbitrary); then we select the set of codes corresponding to the type of character just compressed, using this set to compress the next character. At all times, the current character determines the coding scheme to be applied to the next.

Expansion is accomplished by decoding the first character using the same set of codes chosen during compression. The type of this decoded character is used to determine the expansion scheme for the next character, and so on.

An Example

Storing the following data record using one byte (8 bits) per character will require 29 × 8 or 232 bits:

0187080βGORDONβVβCORMACKβββββ

(The symbol β is used to denote a blank.) The fre-

quency of each character in this data segment and their respective codes are given in Table I. No code has been assigned to any character that does not appear in the sample. Ordinarily, all 256 possible characters would have a representation.

When the codes given in Table I are substituted for the characters in the data sample, the record appears as

010 11110 1010 11111 010 1010 010 00 11000 011 1000 11001 011 11010 00 11011 00 1001 011 100 11100 11101 1001 1011 00 00 00 00 00.

The length of the compressed record is now 102 bits (3.42 bits per character), or 43 percent of the uncompressed length.

Dividing the set of symbols into the five types mentioned earlier, of which only three occur—alphabetic (A, C, D, G, K, M, N, O, R, V), numeric (0, 1, 7, 8), and blank (β)—we then determine the frequencies with which each character follows a character of a given type and assign a code for that character in that context (Table II). It is arbitrarily assumed here that at the beginning of the record the context is alphabetic.

To compress the data segment using Table II, we start in the alphabetic context, compressing the first character (0) to 100. Since 0 is of type *numeric*, the next character (1) is compressed using the second set of codes, yielding 110. All subsequent digits are compressed using the numeric context. Note that the remaining zeros in this field receive a shorter code (00) than did the original. When the blank is encountered, it too is compressed using the numeric context, producing the code 111. The next character (G) is compressed using the blank context. Subsequent characters up to and including the following blank are compressed using the alphabetic context. Note that the first of the string of trailing blanks receives a code of 00, whereas subsequent blanks receive the shorter code 0. The compressed string that results is

100 110 01 10 00 01 00 111 10 010 011 0011 010 100 00 110 00 111 010 011 1101 1110 1010 1111 00 0 0 0 0.

This representation requires 76 bits (2.62 bits per character), or 33.5 percent of the original size.

TABLE I. Optimal Codes for Example (single data type)

Character	Frequency	Code
β	8	00
0	3	010
1	1	11110
7	1	11111
8	2	1010
A	1	11101
C	2	1001
D	1	11001
G	1	11000
K	1	1011
M	1	11100
N	1	11010
O	3	011
R	2	1000
V	1	11011

THE IMPLEMENTATION

The installation and use of the compression routines described above involve four steps. First, the character set is partitioned (manually) into the anticipated data types. Second, a data sample is gathered, and statistics computed based on this sample. Third, using these statistics, sets of codes representing each character are computed, and the actual compression and expansion programs containing the representations are generated. The fourth and final step involves installing the programs in the IMS system where they are called, as necessary, to compress and expand data segments.

Partitioning into Data Types

For the IMS system, the set of data types was fairly easy to select as the main types were those manipulated by the underlying IBM architecture. In addition, characters that make up implicit fields (e.g., digits and the space character) were treated as separate data types. In some applications, the separation of other data types may be desirable: for example, upper- and lowercase letters or other fill characters like the null. Experimentation in our case showed that, in the IMS system, there was little to be gained from partitioning into more than these five or six basic data types.

Finding the optimal automatic partitioning based on a sample of data, as in step 2, is, in general, an intractable problem. We had limited success in applying a number of heuristics, but were unable to achieve the performance of hand-selected partitionings.

One such heuristic starts with 256 different types, one for each character. Although using 256 different sets of codes cannot possibly yield worse compression than a smaller number of types, it is very wasteful of storage. Since many of the sets of codes are presumably similar and could be combined, our heuristic determines which sets should be combined in order to yield the best t sets for a given t. First, the 256 different sets of codes are computed and then, for every possible pair from the 256 different types, the cost of combination is computed. A straightforward but expensive way of measuring this cost is building a set of codes for the combined frequencies and computing the resulting degradation in compression. The pair having the least cost of combination is then replaced by the combined representation. This process continues until the number of types has been reduced from 256 to t. The results of this heuristic are compared with hand selection in Table IV (page 1341).

Gathering Statistics

To generate the compression codes, the probability with which each character follows one of a given type

TABLE II. Optimal Codes for Example (multiple data types)

Alphabetic context			Numeric context			Blank context		
Character	Count	Code	Character	Count	Code	Character	Count	Code
β	3	00	β	1	111	β	4	0
0	1	100	0	2	00	G	1	10
A	1	1110	1	1	110	C	1	111
C	1	1010	7	1	10	V	1	110
D	1	1011	8	2	01			
K	1	1111						
M	1	1101						
N	1	1100						
O	3	010						
R	2	011						

has to be estimated. To this end, we compute a table of character counts (as in Table II), where the size of this table is t by 256; t is the number of types, and 256 is the number of possible characters. The table can be used directly in the construction algorithm and need not be normalized to a table of probabilities. However, each row of the table must contain a representative distribution of counts. To ensure this is the case, care must be taken that there is a significant quantity of each possible type in the sample data. The exact ratios are not critical, as the relative overall weights of the rows of the table do not affect code selection.

Modifications to Huffman Codes

To facilitate implementation, the sets of codes used for compression were not those yielded by Huffman's algorithm. Two modifications were effected: The assignment of codes to represent characters was different, and the codes were constrained so that no code would exceed 15 bits in length.

There are a large number of codings that yield the same compression as Huffman's, but cannot be produced by his construction algorithm. For example, any assignment of codes to characters such that the lengths of the assigned codes are identical to the Huffman codes is as good as a Huffman code.

The particular assignment of codes used in our case is as follows: First it is determined (perhaps using Huffman's construction algorithm) what code length l_i is to be assigned to each character c_i. The character c_i with the largest l_i is assigned the code 111 . . . . . 111 (a string of l_i ones). Note that $l_i \geq 8$. The character c_j with the next largest l_j is assigned the code 111 . . . . . 110 ($l_j - 1$ ones followed by a zero). This assignment continues in order of l_i; at each step the character c_j is assigned a code constructed by taking the binary number formed by the first l_j bits of the previous code and subtracting 1.

This code assignment technique allows the compressed text to be padded on the right with up to seven ones in order to fill an integral number of bytes. Since the IMS system manipulates the length of the compressed data in bytes anyway, this padding removes the necessity for storing the length in bits. On expansion, the compressed string is processed one byte at a time until it is exhausted. We know the padding cannot possibly generate a spurious character since the extra characters are the prefix to a code that is at least eight bits long (the code with the longest l_i). Any processing of such a prefix is merely discarded when the compressed string is exhausted.

The alternate assignment of codes we initiated also facilitates constraining the largest l_i from exceeding 15—a length constraint occasioned by implementation considerations. The compression is done by extracting the codes from a table. In order to have fixed-sized entries in this table, it is necessary to bound the maximum length of any entry. Fifteen bits was determined to be a convenient length as it allowed each entry to be two bytes long, reserving one bit in each entry for a length indicator.

As it is unclear how to modify Huffman's construction algorithm to meet the length constraint, since the algorithm constructs a coding tree from the bottom up, a top-down construction algorithm was used instead. Although much slower than Huffman's, this algorithm was acceptably fast and allows the maximum code length to be constrained. Using the top-down algorithm, code lengths are first selected using a recursive algorithm that examines all feasible choices, and then actual codes are assigned using the method described above.

The Compression and Expansion Programs

Because the compression and expansion routines are built into IMS, we attempted to minimize the amount of time and storage they consume by designing the data structures and algorithms with this constraint in mind. Particular attention was paid to the time required to expand the compressed data. Expansion time was given the greatest priority because data expansion is required whenever data are retrieved, whereas compression is required only when a segment is updated or added, a much less frequent operation. The only time a large amount of compression is done is during a complete reloading of the database. Knowing that expansion time

is commonly considered the biggest drawback of Huffman coding gave us an additional impetus to optimize that operation.

The size of the compression routines is significant because they reside in common storage and therefore compete with every application program for available memory. A size of approximately 10K bytes was decided upon for the compression routine: This size allowed the compression information to be stored in a reasonable fashion without excessive packing, yet does not allow the multiple redundancy that would be required by some speed optimizations.

Separate representations were decided upon for the compression and expansion codes: Compression information is represented as a table, and the expansion information as a graph. Both the compression and expansion routines consist of the appropriate data structure—which is generated by the third step as an assembly-language program—coupled with a small assembly-language program that uses the data structure.

The Compression Program. The data structure used for compression consists of t tables, named $ctable_0$ through $ctable_{t-1}$. The first character to be compressed is encoded using $ctable_0$, and subsequent characters are compressed using $ctable_{type[z]}$, where z is the previous character compressed. A particular $ctable_i$ is indexed by the character to be compressed and yields a 16-bit string. The appropriate code is contained in the rightmost bits of this string. Immediately preceding this code is a single bit with the value one. The remaining bits to the left of this bit are zeros.

The length of the code is computed by linear search. Initially, the bit string 10 is loaded into a register. Next, a single *add, test, and branch* instruction is used to double this number (add it to itself) and compare it to the *ctable* entry. The number of times that this single-instruction loop executes determines the number of bits in the code. The appropriate shift instructions are then used to copy the code to an output buffer.

Although a linear search may at first seem inappropriate in this case, one must consider that the average compressed length for a character is about 2.5 bits and that, as a result, this small loop will iterate only 2.5 times on average. Any method that stored the length explicitly would have complicated the indexing and format of *ctable* sufficiently that more instructions would have been executed.

Using this format, each *ctable* occupies 512 bytes. For $t = 5$, the entire compression routine is smaller than 3K bytes. CPU time consumed is 2.4 μs per byte on an Amdahl 5850.

The Expansion Program. Expansion information is not amenable to representation as a table because, when a code is to be expanded, it is not known in advance how long the code will be; instead, the code must be processed bit by bit until its end is detected, and then the corresponding character is produced. To facilitate expansion, it is common to represent Huffman codes as trees. Using this method, a code is expanded by examining it one bit at a time, and moving to the left in the tree if the bit is zero and to the right if the bit is one. When a leaf is encountered, the end of the code has been reached. Associated with each leaf is the character represented by the code.

Once again, there are t expansion trees, $etree_0$ through $etree_{t-1}$. Each tree is represented programmatically in the following manner: A node in the tree occupies 16 bits. For interior nodes, these 16 bits are a (slightly convoluted) offset pointer to the right subtree. The root of the left subtree immediately follows this node, and therefore no left pointer is required. For leaf nodes, the second byte is the character represented by the code that terminates at this node. The first bit of the first byte is one—a flag indicating that this is a leaf. The remaining 7 bits of the first byte contain the data type of the character in the second byte.

This representation not only saves space, but also facilitates a very fast expansion algorithm. Expansion using the tree method is often slow because the compressed data must be handled one bit at a time, rather than one byte at a time. Therefore, every effort was made to minimize the number of instructions executed per bit.

Conceptually, the expansion algorithm we developed is as follows:

```
p ← root of etree0
for every byte B in compressed data do
  for every bit b in B do
    if b then p ← right link of node at p
          else p ← left link of node at p
    if node at p is a leaf then
      place character represented by node at p in
        buffer
      p ← root of etree_type[character]
```

The outer loop is synchronized in bytes, rather than bits, and because of the code-assignment process described earlier in Modifications to Huffman Codes (page 1339), there is no need to perform end-of-data checking in the inner loop. Furthermore, because the inner loop iterates exactly eight times, it is written as a macro and therefore expands at assembly time. The processing required for every bit is merely the body of the inner loop.

One further optimization was applied: removing the *else* clause on the first *if* statement. This removal saves time whether or not the test is true; if the test is false, there is no *else* clause to execute; if true, no branch is necessary to bypass the *else* code. The optimization takes advantage of the fact that there is no left link in the data structure; and therefore no work is done at execution time to follow this link.

For internal nodes, the expansion algorithm executes 4.5 instructions per bit. Since the average compressed text is 2.5 bits per character, the total overhead for processing the compressed text bit by bit is about 12.0

TABLE III. Size of Compressed Data

t	Type names	Bits per character
1	None	3.55
2	Blank, other	3.23
3	Blank, alphanumeric, other	2.92
4	Blank, alphabetic, numeric, other	2.68
5	Blank, alphabetic, numeric, packed, other	2.62
8	Blank, vowels, consonants, 0, 1–9, null, packed, other	2.56

TABLE IV. Automatic versus Manual Type Selection

t	1	2	3	4	5	8	16	32	64	128	256
Automatic	3.52	3.24	3.18	3.14	3.12	2.79	2.68	2.54	2.44	2.38	2.38
Manual	3.52	3.23	2.92	2.68	2.62	2.56	—	—	—	—	—

instructions per byte. The remaining cost of compression is that of moving the compressed and expanded data byte by byte.

Using this data representation, each graph consumes 1K bytes. Thus, for $t = 5$, the expansion routine occupies slightly over 5K bytes. On an Amdahl 5850, the expansion routine consumes 2.2 μs per character, somewhat less than the compression routine.

COMPRESSION RESULTS

The results reported below were derived from the student-records database at the University of Manitoba. First, we present the results of applying the compression method to data extracted from this database and then describe the in situ performance of the routines.

The compression program was generated using from 1 to 8 manually selected data types. The size of the compressed data for each value of t is presented in Table III. In addition, the automatic partitioning heuristic was used to select from 1 to 256 data types. The compression achieved is shown in Table IV; for purposes of comparison, the results of Table III are duplicated here as well.

It can be seen from these results that a carefully handpicked set of types can achieve much better compression for the same t than the automatic method. Indeed, the automatic method requires $t = 16$ to match the compression of $t = 4$ achieved using the manual method. Other heuristics were tried, but most yielded comparable results.

Table IV also shows that the best possible compression using the automatic technique yields 2.38 bits per character. The manual scheme using $t = 5$ exceeds this optimum by less than 10 percent. Note also that the compressed size of 3.52 bits per character for $t = 1$ is equivalent to Huffman's method applied character by character.

To compare the automatic method with a competing compression technique, digram coding, a single optimal coding was constructed for each possible character pair, or *digram*. The digram-coding method, which takes advantage of the correlation between adjacent characters, yielded a compressed size of 2.85 bits per character, a somewhat worse compression than that described above using $t = 4$. Moreover, the size of the digram table is as large as all the tables using $t = 256$. Encoding only a subset of the digrams, or using a different coding scheme, would yield poorer results.

In Situ Results

Based on the above results, it was decided that a compression and expansion program should be generated and installed using $t = 5$. Although the choice of $t = 4$ yielded similar compression on this particular sample, it was felt that the choice of $t = 5$ would make the program perform better if it were used on a database with a large amount of packed and binary data.

After installing this program, the student-records database was loaded on an Amdahl V7 computer. The statistics for this installation (Table V) show that the overall size of the database was reduced by 42.1 percent, somewhat less than the 65.0 percent reduction that might have been expected from examining the results shown in Tables III and IV. This shortfall is accounted for by the fact that an IMS database contains a considerable amount of information that is not eligible for compression—internal pointers, length fields, indexes, and key fields. Nonetheless, the compression program does save a considerable amount of space.

TABLE V. Compression Results on the Student-Records Database

	Overall size	Physical I/O	CPU (V7)
Without compression	2205K	1022	32.0 s
With compression	1275K	681	37.5 s
Ratio	57.9%	67.3%	117.2%

As a consequence of this decreased size, 32.7 percent fewer disk operations were required to load the database. Although compression added 17.2 percent to the total CPU time required to load the database, the increase in CPU time is less than that actually consumed by the compression process because, measured independently, the *compression* routines required 11.0 s to compress the same data. (The V7 time is roughly twice as long as the 5850 time reported on page 1340.) The net increase of only 5.5 s results from the fact that the IMS system consumes CPU time in copying the data.

Other statistics on the performance of the compression program could not be determined reliably. The real time required to load the database, for example, could not be measured with any degree of consistency since this is a function of CPU time, the number of disk operations, and system load. In general, however, the time needed to load the database was found to be less with compression than without. It was also impossible to measure the amount of CPU time consumed on retrieval: Some retrieval programs ran faster with compression, and some were faster without. In either case, the difference in CPU time was not significant.

Other Databases

The compression routines described here have allowed the University of Manitoba to store all of its administrative data on nine 3350 disk volumes, instead of fourteen. Other sites have installed the compression routines with similar results.[1]

Most databases compress well with the routines originally generated from the student-records database. Typically, users report that an additional savings of between one and three percent can be realized by generating custom routines. However, in the case of the employee-records database, which contained a fair amount of binary and packed information that was virtually absent in the original sample (i.e., there were only unsigned numbers), new routines were able to *double* the compression. Common routines could have been generated from a more representative sample with similar results, but this was not done due to the complexity of converting the many existing databases using the original compression routines.

CONCLUSIONS

The methods presented here achieve significant savings in disk storage and disk activity. On average, the compression method can be expected to reduce the size of the raw data by approximately a factor of three, resulting in an overall reduction in database size approaching a factor of two.

The speed of the routines seems more than adequate, particularly on a large mainframe computer, and at least part of the cost of compression is recovered by savings in data movement. This performance is achieved, and the synchronization problem due to variable-length bit strings solved, through the design of special data structures and algorithms. These techniques can be applied on any CPU to any method based on Huffman's, thus overcoming a major objection to this method.

[1] The author has been involved directly with 5 sites and has received correspondence from approximately 50. The routines have been distributed freely, so it is virtually impossible to count the actual number of installations. Readers interested in acquiring a copy should contact Computer Services Dept., University of Manitoba, Winnipeg, Canada, R3T 2N2.

Acknowledgments. The author thanks University of Manitoba Computer Services for providing the live data and computer resources with which to experiment.

REFERENCES

1. Cormack, G., and Horspool, R. Algorithms for adaptive Huffman codes. *Inf. Process. Lett. 18*, 3 (Mar. 1984), 159–166. Contains algorithms that alter a set of Huffman codes to reflect changing the probability of one symbol, as well as adaptive coding schemes based on these algorithms.
2. Huffman, D. A method for the construction of minimum-redundancy codes. *Proc. I.R.E. 40*, 9 (Sept. 1952), 1098–1101. This original paper on Huffman's compression method contains a proof that the method is optimal.
3. IBM. Information management system: Programming reference manual. 9th ed. SH20-9027-8, IBM, 1981, pp. 3.3–3.38. Describes the IMS system from the system programmer's point of view, including the implementation and use of the data-compression exit facility: Contains a sample compression routine that implements run-length encoding.
4. Reghbati, H. An overview of data compression techniques. *Computer 14*, 4 (May 1981), 71–75. Surveys briefly a number of common data-compression methods.
5. Schuegraf, E.J. A survey of data compression methods for non-numeric records. *Can. J. Inf. Sci. 2*, 1 (May 1977), 93–105. Provides an abstract overview of the information theory behind data compression, and a presentation of several classes of compression methods.
6. Severance, D. A practitioner's guide to data base compression. *Inf. Syst. 8*, 1 (1983), 51–62. Covers in detail a large number of data-compression methods that may be applicable to database systems: Includes a comprehensive list of 80 references to the literature.
7. Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3 (May 1977), 337–343. Describes an adaptive coding scheme that encodes progressively longer input strings as integers.

CR Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: *data compaction and compression*; H.0 [**General**]: *IMS*; H.1.1 [**Models and Principles**]: Systems and Information Theory—*information theory*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: database, data compression, Huffman code

Received 12/84; accepted 3/85

Author's Present Address: Gordon V. Cormack, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada.