LBL-10648
CONF-801009--2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

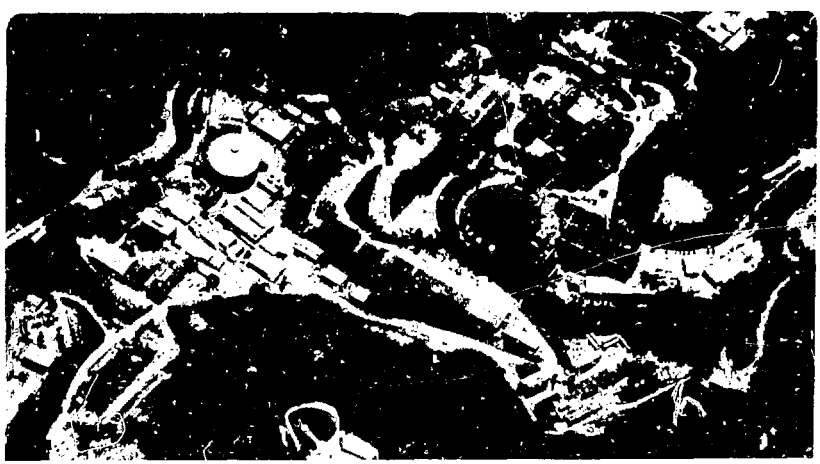
Physics, Computer Science & Mathematics Division

To be presented at the Sixth International Conference on Very Large Data Bases, Montreal, Quebec, Canada, October 1-3, 1980

EFFICIENT ACCESS OF COMPRESSED DATA

Susan J. Eggers and Arie Shoshani

June 1980



Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48

Efficient Access of Compressed Data

by Susan J. Eggers and Arie Shoshani

Department of Computer Science and Applied Mathematics
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Abstract

In this paper a compression technique is presented which allows a high degree of compression but requires only logarithmic access time. The technique is a constant suppression scheme, and is most applicable to stable databases whose distribution of constants is fairly clustered. Furthermore, the repeated use of the technique permits the suppression of a multiple number of different constants.

Of particular interest is the application of the constant suppression technique to databases whose composite key is made up of an incomplete cross product of several attribute domains. The scheme for compressing the full cross product composite key is well known. In this paper, however, the general, incomplete case is also handled by applying the constant suppression technique in conjunction with a composite key suppression scheme.

1. Introduction

An effective method for storing very large databases is the use of compression techniques. Unfortunately, since compression involves encoding the data, some efficiency in access[1] must usually be sacrificed to achieve the smaller database. Some compression schemes require that the data be decompressed to its original form before being searched. Others allow the data to be used in the compressed form, but still require accessing overhead. In both cases the savings in space is accompanied by additional access time of order n, where n is the number of values or tuples which must be processed in order to find the data [Alsb, Aron, Gott, Hahn, Knut]. Tarjan and Yao have developed a row displacement scheme in which access is logarithmic, but the database is not fully compressed [Tarj].

In this paper a compression technique is presented which allows a high degree of compression but requires only logarithmic access time. The technique is a constant suppression scheme, and is most applicable to stable databases whose distribution of constants is fairly clustered. Furthermore, the repeated application of the technique permits the suppression of a multiple number of different constants.

Another type of data repetition occurs when tuples are identified by the cross product of several attribute domains. These composite keys can be eliminated from the database, the domains of their category attributes stored in a dictionary and the tuples located computationally. The computation scheme fails, however, when all possible values of the cross product

---

[1] Compressed data, of course, requires fewer secondary storage fetches. In this paper, however, the term access refers to the amount of time needed to locate values already in core.

are not represented in the relation. This situation, which shall be referred to as an incomplete cross product, can be nicely handled by using the constant suppression technique presented here in conjunction with composite key suppression. The access time for the combined techniques is still logarithmic.

In section 2 the characteristics of databases which are amenable to the above constant suppression technique are identified. A description of the technique and its data access algorithms follows in section 3. Section 4 contains the application of the technique to the case of multiple constants in a database. Section 5 discusses the use of the constant suppression technique in conjunction with a key suppression scheme to handle databases with an incomplete cross product. The implications for data access of complete file transposition when using the constant suppression scheme appear in section 6. And section 7 contains a storage compression and access time comparison with two other techniques.

2. Characteristics of Statistical Databases

The compression technique presented in this paper was designed to be effective for certain types of statistical databases which are widely used in sociological and scientific applications. The sociological databases are comprised of tabulations of socio-economic and environmental survey data, usually geographically based and summarized over demographic attributes, such as race and sex. Examples in use at Lawrence Berkeley Laboratory (LBL) are population data in the 1970 U.S. Census Summary Tapes (4th Count), age-adjusted mortality rates, air quality data for individual monitoring stations and county business patterns. The scientific data results from laboratory experiments in which some parameters are held constant

while others are systematically varied. An example from Battelle's Pacific Northwest Laboratory is the measurement of skin temperature and blood flow on laboratory rats in response to varying intensities of radiation, and positions and types of probes.

The databases are organized into relations. Each tuple consists of category attributes, which uniquely define it and constitute the composite key, and non-category or non-key attributes, which supply the content of the database. For example, in the LBL database of mortality data by race, sex, disease, age group and geographical area, the latter five attributes are the category attributes and serve as the composite key, while the mortality counts and rates are the non-category attributes.

The databases are usually sorted in the lexicographic order of their composite key. Thus, for most tuples, all key attribute values but one are repeated from tuple to tuple. The databases are therefore potential candidates for composite key compression. In certain cases, some instances of the composite key are invalid; for those missing combinations no tuples exist. This incomplete cross product often occurs in the recording of experimental data for which some parameter combinations are unavailable or meaningless.

In addition to the repetition of values in the composite key, there is repetition in the non-key (non-category) attributes. The repetition stems from the prevalence of a particular constant value in the database, most typically zero or null. Depending on the ordering of the composite key values, the constants for a particular non-category attribute tend to cluster, i.e., appear in logically consecutive tuples. For example, since there are relatively few Chinese in Vermont, a census database which is

comprised of a count of population by race, state and county is likely to contain a series of zeros in the section for counties in Vermont.

Two factors cause the databases to be extremely large. First, they may contain hundreds of non-key attributes. More importantly, the cardinalities of the individual domains in the composite key can themselves be quite large; and the number of tuples generated is the product of these cardinalities. The mortality database contains the cross product of four races, two sexes, 70 diseases, six age groups and 3000 counties, amounting to over ten million tuples. The category attributes alone would occupy over sixty megabytes, if explicitly stored in the database.

For the databases at LBL query selection predicates[2] are most commonly applied to the category attributes of the composite key, while output is projected from both the category and non-category attributes[3]; only occasionally is selection based on non-category attributes[4]. Thus, most accessing will involve locating the non-category attributes in the compressed database, given their uncompressed position, rather than searching for them in the compressed form. The former type of access can therefore be optimized at the expense of the latter. In addition, queries tend to output relatively few of the attributes in a tuple. Consequently, the databases are amenable to complete file transposition.

---

[2] A query is assumed to contain two parts: a selection predicate which specifies the conditions of qualification and an output clause which projects attributes for display.

[3] e.g., SELECT race = white, state = California; OUTPUT county, population.

[4] e.g., SELECT population > 500,000; OUTPUT county.

Since the databases are summaries of survey data and results of experiments, they tend to be extremely stable; updates are practically nonexistant. Therefore, compressed databases will rarely have to be recompressed.

3. Suppression of Constants

In this section we consider the compression of constant values in the non-category attributes; the compression of repeating values in the composite keys is discussed in section 5. However, in order to clearly illustrate the constant suppression technique, the discussion here assumes that composite keys have been compressed, and that the tuples contain only the non-category values.

For the purpose of explaining a constant supression scheme, one can think of the original, uncompressed tuples as the logical form of the database, and the compressed data which is actually stored as the physical form. A desirable mapping between the two would have the property of providing efficient access, while requiring little storage overhead.

The compression technique presented here considers the logical form of the database to be a vector of values, in which a series of constants alternates with a series of unsuppressed values. (Cf. L in Figure 1 below.) Whether a relation is converted into a vector by rows or columns is unimportant to the presentation of the technique, but is discussed later in section 6. The vector representation merely enables the compression scheme to be concerned with the database as a whole and allows non-category attributes to be compressed across either tuple or column boundaries. The physical database is considered to be a vector of unsuppressed values.

The mechanism for mapping between the two vectors uses a header which records the distribution of constants and unsuppressed values in the logical form. The header is a vector of counts, with the odd-positioned counts for unsuppressed values, and the even-positioned for constants. Each count contains the cumulative number of values of its type at the point at which a series of that type switches to a series of the other. The counts reflect accumulations from the beginning of the vector.

For example, consider the vector L in Figure 1, which represents the logical form of a database, in which the 0's are the constants to be suppressed and the v's are the unsuppressed values. Directly beneath the vector at the points at which one type of series gives way to the other, is the list of counts which comprise the compression header, H. The odd-positioned counts hold accumulations of unsuppressed values; and the even-positioned counts hold the accumulations of zeros. The physical, compressed form of the data is represented by P.

The header was designed so that a single structure can be used for mapping from the logical to the physical vector, and vice versa, in logarithmic time. Since the counts are cumulative, and their sums are therefore in ascending order, a binary search on the sums can be utilized for

L: v_1 v_2 0 0 0 0 0 0 0 0 v_3 v_4 v_5 v_6 v_7 0 0 v_8 v_9 v_{10} 0 0 0

H: 2 9 7 11 10 14

P: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10}

Figure 1

both mappings. The algorithms run in O(log s), where s is the number of counts in the header.

The mapping algorithm works as follows:

(1) Forward mapping (logical to physical vector):

Given the ordinal position of a desired attribute instance in the uncompressed, logical vector, we want to determine whether this instance is a suppressed constant or an unsuppressed value; and, if unsuppressed, find its position in the compressed vector. First, the logical position is located in the compression header by doing a binary search on the pairwise sums of adjacent counts. Note that the sum of each successive pair of counts represents the logical position at the end of the series denoted by the second count in the pair. Thus, in the example above (Figure 1), the sum of the second and third counts (16) yields the logical position at the end of the second series of unsuppressed values, and the sum of the third and fourth counts (18) is the logical position at the end of the second series of constants. The binary search is terminated when a sum is found which is just greater than or equal to the given logical position. If the second of the two counts in the terminating sum is an unsuppressed value count, the attribute instance sought is an unsuppressed value. Its ordinal position in the physical vector is its logical position minus the number of suppressed constants up to that point. That accumulation is represented by the first count in the terminating sum. If the second of the counts in the terminating sum accumulates constants, the output value is the constant.

More formally, let the header of counts be represented by the sequence, u_0, c_0, u_1, c_1, u_2, c_2, ..., u_i, c_i, ..., u_s, c_s, where the u's

are the unsuppressed value counts and the c's are the constant counts.[5] Let C be the value of the suppressed constant, V be the desired attribute value, and l and p designate the logical and physical ordinal positions, respectively. A binary search for l on the sums of adjacent counts in the header yields one of two possible cases:

(1) $u_i + c_i < l <= c_i + u_{i+1}$,

or

(2) $c_{i-1} + u_i < l <= u_i + c_i$, for some i.

Case (1) implies that V is unsuppressed, and $p = l - c_i$. Case (2) implies that V is a constant, i.e., $V = C$ [6].

In the example database above, the twentieth position in the logical form (l = 20) maps to the ninth physical location (p = 9). The binary search of the header terminates at the sum of 11 (c_2, for constants) plus 10 (u_3, for suppressed values). The number of accumulated constants, 11, is subtracted from l, leaving a physical position, p = 9. The value at that position is v_9. On the other hand, when l equals eighteen, the search terminates at a sum ($u_2 + c_2$) in which the second of the two counts accumulates constants. Therefore the value of the eighteenth attribute is the constant, C = 0.

---

[5] u_0 and c_0 always contain the value zero. They are included in the header, so that an attribute instance occurring in the first series of uncompressed data does not have to be treated as a special case in the forward and backward mapping algorithms. For the sake of clarity, however, u_0 and c_0 will be omitted from the example figures.

[6] For the case in which the logical vector begins with a series of constants, u_1 contains a count of zero.
For the presentation here, it is assumed that c_{i-1} and c_{i+1} exist.

(2) Backward mapping (physical to logical vector):

Given the physical position p of an unsuppressed value, only the unsuppressed value counts in the header are searched to find the i, such that $u_{i-1} < p <= u_i$. The equivalent logical position is calculated by adding to p the number of constants suppressed from the database up to p, i.e., $l = p + c_{i-1}$.

Again, using the above example database, a binary search on the compression header for the ninth physical position (p = 9) terminates at the third unsuppressed value count (which contains a 10). Adding the preceding count of compressed constants (11) to p results in the twentieth position in the logical form (l = 20).

The size of the header is, of course, dependent on the distribution of constants in the database. The more clustered the suppressible values, the smaller the header; and, the more constants in the database, the higher the likelihood of their occurring in long series. In the worse case, the database is filled with alternating constants and unsuppressed values, and the header is the same order of size as the uncompressed data. In this case, a compression scheme which is independent of the distribution of constants, e.g., constant suppression with a bit map [Aron], may be preferable. In actual practice, however, statistical databases are often sufficiently sparse and their sparseness is sufficiently clustered to produce satisfactorily small headers. The previously mentioned mortality database is aproximately 70 percent sparse, and the database on county business patterns is 50 percent empty.

An analysis of the break-even point at which storage will be saved by the constant suppression scheme, i.e., the point at which the overhead of the header is less than the storage saved by compression, will give a rough estimate of the size of the header. Let a be the average length of a series of constants or unsuppressed values, p the fraction of constants in the database and n the length of the uncompressed, logical database. For simplicity, let us assume that the size (in bytes) of a count is the same as that for a data value. The break-even point is reached when the number of series, n/a, equals p*n or p = 1/a. For example, if the average length of a series is 100, then space is saved as long as the database contains more than one percent constants.

4. The Case of Multiple Constants

Often there is the prevalence of more than one common value in a database. Each of these constants can be eliminated, by applying the constant suppression scheme repeatedly, once for each different constant. There exists a header for each compression, and the mapping algorithm computes from a position in the header for the first compression to a position in the header for the next, and so on, until a position in the physical data is reached.

Consider the example in Figure 2. The uncompressed database is represented by L, where the v's are the unsuppressed values, and 1 and 0 are the constants. The constant 1 is suppressed first; therefore in the first compression header, H1, the unsuppressed value counts reflect both unsuppressed values and zeros. The suppression of 1's results in the logical representation depicted in L', where the v's are again the unsuppressed values and 0 is the remaining constant. H0 is the compression header for

L: v_1 v_2 v_3 1 1 v_4 v_5 0 0 v_6 1 1 0 0 v_7

H1: 3 2 8 4 11

L': v_1 v_2 v_3 v_4 v_5 0 0 v_6 0 0 v_7

H0: 5 2 6 4 7

P: v_1 v_2 v_3 v_4 v_5 v_6 v_7

Figure 2

suppressing zeros. That final compression results in the physical representation of the data, P. As an example, the tenth position in the first logical representation, L, maps to the eighth position in the second logical representation, L', and then to the sixth location in the compressed, physical data, P.

An alternative scheme for compressing multiple constants from a database is to eliminate all types of constants on the first compression, and then utilize different headers to successively differentiate between the constants. Using the above example, the logical form of the database, L, the compression header which eliminates all types of constants, HC, and the compressed data, P are represented by:

L: v_1 v_2 v_3 1 1 v_4 v_5 0 0 v_6 1 1 0 0 v_7

HC: 3 2 5 4 6 8 7

P: v_1 v_2 v_3 v_4 v_5 v_6 v_7

Figure 3

The constants which have been compressed from the database, C, and the header which indicates their distribution, H, are represented as:

C: 1 1 0 0 1 1 0 0

H: 2 2 4 4

Figure 4

The variation is advantageous either for forward mapping when the output is an unsuppressed value or for backward mapping. Under both circumstances only one application of the mapping algorithm is required. However, the scheme is less efficient than the previous one when outputting all constants but the last one suppressed. In this case an extra mapping is necessary.

5. Incomplete Cross Product

An important application of the constant suppression scheme occurs for databases with an incomplete cross product. In order to explain the treatment of incomplete cross product, we first describe the compression of composite keys in the case of a full cross product.

If every combination of category attribute values is present in the database, the composite key need not be stored. As long as the attribute

values are encoded (presumably in a dictionary), and the codes are ordered in some way, the position of the tuple sought can be calculated. For many statistical databases the ordering of the composite key values is lexicographic, and the well-known array linearization algorithm[7] can be applied to the encoded values of the category attributes to calculate the tuple position.

However, statistical databases often have an incomplete cross product, i.e., there are no tuples which correspond to certain combinations of category attribute values. In order to utilize the key suppression technique, null tuples have to be introduced into the database to expand it to its full cross product. If the database is too sparse, the cost of storing the null tuples outweighs the savings of eliminating the composite key and the computational access. Svensson [Sven] treats this problem by representing the composite key as a trie and compressing the repetitive values from the children of a node using the run-length encoding scheme described in section 7. However, since the compression does not apply across the children in the trie, there is still some repetition of values in the composite key.

---

[7] $j_1 * N_2 * N_3 * \ldots * N_k +$
$j_2 * N_3 * \ldots N_k + \ldots +$
$j_{k-1} * N_k +$
j_k

where j_i is the encoded value of a category attribute i used for selection; N_i is the cardinality of each category attribute i; k is the number of category attributes [Horo, among numerous others]. Assuming three races, two sexes and ten diseases, with female = 0, white = 2, lung cancer = 6, the tuple in which

sex=female, race= white, disease=lung cancer,

would occupy the twenty-sixth or (0 * 2 * 10 + 2 * 10 + 6) position.

An alternative, which eliminates both the null tuples of the key suppression scheme and the duplicates of the Svensson approach, is the key suppression scheme used in conjunction with the constant suppression technique presented in this paper. For their combined usage, the null tuples introduced into the cross product by the key suppression scheme are compressed from the database, and a compression header is constructed to reflect their absence. The combined algorithms still require only logarithmic access time.

For queries in which selection is given on category attributes and output is projected from non-category attributes, query processing would proceed as follows. First, the tuple position in the cross product is calculated using the array linearization algorithm on the encoded, ordered category attribute values. Then, using the header for compressed null tuples, the forward mapping algorithm of the constant suppression technique is applied to determine whether the tuple actually exists. If constants have been compressed as well, the forward mapping algorithm is again applied, once for each different constant suppressed.

Queries selecting on non-category attributes can also be satisfied. The backward mapping algorithm is first applied to determine the tuple position. The category attributes' values are then computed using the reverse of the array linearization algorithm.

The combination of the key and constant suppression schemes gives best results for databases with highly clustered null tuples. In this case the savings in composite key storage far outweighs the overhead of the compression header. However, even in the worst case, in which the database is filled with alternating null and actual tuples, storage savings can be

realized. Assuming the worst case, let n be the number of tuples in the incomplete cross product (also the number of null tuples), k be the size of a tuple and c the size of a header count. The break-even point at which storage will be saved by using the constant suppression scheme to enhance composite key suppression occurs when nk > 2nc, i.e., when the tuple is more than twice the size of a count.

Some orderings of attributes in the composite key will clearly produce more tuple clustering than others. A more fortuitous distribution of null tuples could be achieved by analyzing the data for an optimal ordering.

6. Implications of the Constant Suppression Technique on File Transposition

File transposition [Bato, Hamm, Turn] of a database refers to the physical division of tuples on attribute lines and the storing of each partition of attribute values separately. The technique is advantageously employed on statistical databases whose tuples contain a large number of attributes, only a few of which are accessed in a single typical query. By partitioning the tuples and accessing a subset of attributes as needed, more tuples can be accessed per page, thereby reducing secondary storage access and increasing the efficiency of query processing [8].

When the constant suppression technique has been applied to a database, it is advantageous to completely partition it, rather than to cluster into the same partition those attributes which tend to be requested together. When only one attribute is stored in each partition, the physi-

---

[8] No correspondence between partitions and files is being made here, i.e., a partition is not necessarily a file.

cal position needed for backward mapping can be located by sequentially searching the partition for a particular value. If the attributes are clustered into partitions, data compression makes it impossible to differentiate between the attributes in the partition. Each attribute value must therefore be checked by first locating its logical position in the compression header and then mapping to the physical location.

As mentioned in section 2, it is quite probable that a series of constants will occur for a single attribute. Clustered file transposition tends to break up that natural concentration of constants. It is much less likely that all instances for different attributes in a partition will have the same value. Therefore the compression header that is generated by clustered file transposition will generally be longer than that for complete file transposition.

A disadvantage of complete file transposition for databases compressed using the constant suppression technique is that each output attribute in a qualifying tuple must be located by a separate application of the mapping algorithm. However, since the algorithm runs in O(log s), the overhead is not prohibitive.

There are, of course, other factors, such as the unpredictability of query attribute usage, which would effect the decision of whether to utilize complete or clustered vertical partitioning. A discussion of these factors, as they pertain to statistical databases, is given in [Turn]. However, since they are issues which apply regardless of compression, they will not be discussed here.

7. Comparison With Other Compression Schemes

In this section we contrast the constant suppression technique to two other techniques for compressing constants. One achieves a degree of compression similar to that of the scheme presented in this paper; the other is usually less successful. Both, however, generate a greater overhead in access time.

(1) Constant suppression with a bit map [Aron].

Similar to the scheme presented here, constant suppression with a bit map eliminates commonly occurring values from the database and depicts the logical form of the data with a bit map. In the bit map all set bits indicate unsuppressed values and all cleared bits indicate suppressions. For forward mapping the set bits must be counted up to the logical position of the attribute selected. Their sum is the physical position of the instance. (Of course, if the logical position indicates a cleared bit, the desired value is the constant.) For backward mapping, the set bits must also be counted until the value of the physical position of the instance selected is reached.

While this scheme is comparable in concept to the constant suppression with a header technique, its access time for both forward and backward mapping is of O(n) time, where n is the number of bits in the map. Although for some applications bit operations can be utilized, the algorithm is still linear, and is computationally costly for large n. It was shown in section 3.1 that the algorithms for both forward and backward mapping in the constant suppression scheme presented here run in O(log s) time, where s is the number of counts in the header (or constant and unsuppressed

series in the uncompressed database). Since, for most databases, s is much smaller than n, it follows that $O(\log s) << O(n)$[9].

Since both schemes eliminate all constants from the database, their storage requirements for the data are identical. Therefore, to compare space savings, we will compute the break-even point for the storage of their respective headers. Let a be the average length of a series of constants or suppressed values, n be the number of values in the uncompressed database and c be the size of a header count in bits. The break-even point occurs when $(n/a)c$ equals n, or $c = a$. Thus, more space is saved by the constant suppression scheme utilizing a header with sixteen bit counts, if the average length of a series of constants or unsuppressed values is greater than sixteen.

(2) Run-length encoding [Aron].

In a run-length encoded database, a series of constants is replaced by a series indicator and a count of the number of constants eliminated. For both forward and backward mapping, one must sequence through the data, counting the number of unsuppressed values and references to the number of constants. Forward mapping requires O(n) time. In backward mapping, the counting for the logical position is subsumed in the search of the data.

Space requirements for the two schemes are comparable. In addition to the data actually stored, run-length encoding requires, for each series of constants, a series indicator and a count. For the same series of constants, the overhead for the constant suppression scheme presented here is two counts, one for the constant series and one for the unsuppressed value

[9] It is assumed that both algorithms are evaluated in software.

series preceding it.

8. Summary

We have presented a technique for data compression which is applicable to databases with a high prevalence of common values. Its main advantage is fast access time. The access for both forward and backward mapping is logarithmic, and is achieved through the binary search of a single header which describes the distribution of the constants and unsuppressed values.

The successive use of the constant suppression technique can handle multiple constants, and its use in conjunction with a composite key suppression technique is applicable to the general case of an incomplete cross product of composite keys. For databases which have been compressed using the constant suppression scheme, complete file transposition is advantageous.

The technique and its applications are currently being implemented for performance evaluation.

Acknowledgement.

We would like to thank our colleagues, Rowland Johnson and Peter Kreps, for many helpful discussions. This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U. S. Department of Energy, under contract W-7405-ENG-48.

REFERENCES

(1) Alsberg, P. A. "Space and Time Savings Through Large Database Compression and Dynamic Restructuring," Proceedings of the IEEE, Vol. 63, no. 8, August, 1975, 1114-1122.

(2) Aronson, J. Data Compression - A Comparison of Methods, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C., 3-5.

(3) Batory, D. S. "On Searching Transposed Files," ACM Transactions on Database Systems, Vol. 4, no. 4, December, 1979, 531-544.

(4) Gottlieb, D., Hagerth, S., Lehot, P., Rabinowitz, H. "A Classification of Compression Methods and their Usefulness for a Large Data Processing Center," Proceedings of the National Computer Conference, Anaheim, 1975, 453-458.

(5) Hahn, B. "A New Technique for the Compression and Storage of Data," Communications of the Association for Computing Machinery, Vol. 17, no. 8, August, 1974, 434-436.

(6) Hammer, M., Niamir, B. "A Heuristic Approach to Attribute Partitioning," Proceedings of the International Conference on Management of Data, Boston, 1979, 93-101.

(7) Horowitz, H. and Sahni, S. Fundamentals of Data Structures, Computer Science Press, Inc., Potomac, Md., 1977, 62-66.

(8) Knuth, D. E. The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973, 401.

(9) Svensson, P. "On Search Performance for Conjunctive Queries in Compressed, Fully Transposed Ordered Files, Proceedings of the International Conference on Very Large Databases, 5, Rio de Janeiro, 1979, 155-163.

(10) Tarjan, R. E., Yao, A. C. "Storing a Sparse Database," Communications of the Association of Computing Machinery, Vol. 22, no. 11, November, 1979, 606-611.

(11) Turner, M. J., Hammond, R. and Cotton, F. "A DBMS for Large Statistical Databases," Proceedings of the International Conference on Very Large Databases, 5, Rio de Janeiro, 1979, 319-327.