Lawrence Berkeley National Laboratory
LBL Publications

Title
A COMPRESSION TECHNIQUE FOR LARGE STATISTICAL DATABASES

Permalink
https://escholarship.org/uc/item/79n2j3rv

Authors
Eggers, S.J.
Olken, F.
Shoshani, A.

Publication Date
1981-03-01

eScholarship.org Powered by the California Digital Library
University of California

LBL-12353
c.2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

RECEIVED
LAWRENCE
BERKELEY LABORATORY
AUG 17 1982
LIBRARY AND
DOCUMENTS SECTION

Physics, Computer Science & Mathematics Division

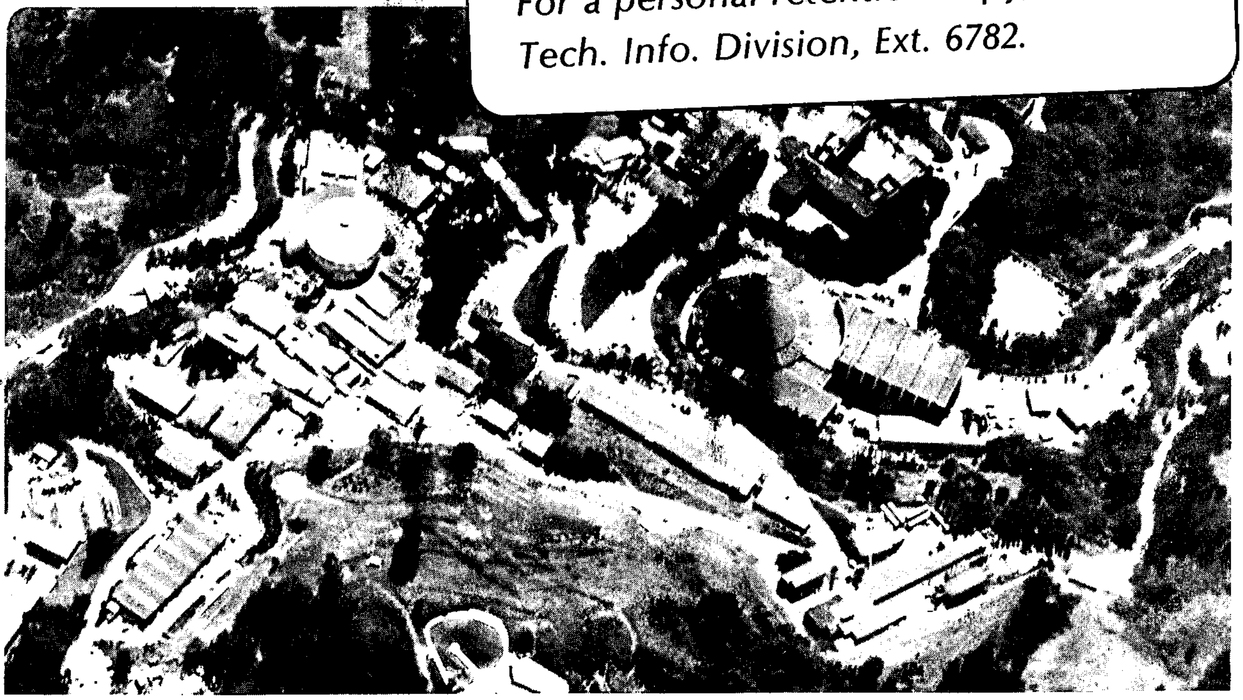
Presented at the 7th International Conference on Very Large Data Bases, Cannes, France, September 9-11, 1981; and published in the Proceedings

A COMPRESSION TECHNIQUE FOR LARGE STATISTICAL DATABASES

Susan J. Eggers, Frank Olken and Arie Shoshani

March 1981

TWO-WEEK LOAN COPY
This is a Library Circulating Copy which may be borrowed for two weeks. For a personal retention copy, call Tech. Info. Division, Ext. 6782.



Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

LBL-12353 c.2

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

A COMPRESSION TECHNIQUE FOR LARGE STATISTICAL DATABASES[1]

by Susan J. Eggers, Frank Olken and Arie Shoshani

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

Abstract

In this paper we explore the compression of large statistical databases and propose techniques for organizing the compressed data, such that the time required to access the data is logarithmic. The techniques exploit special characteristics of statistical databases, namely, variation in the space required for the natural encoding of integer attributes, a prevalence of a few repeating values or constants, and the clustering of both data of the same length and constants in long, separate series. Our techniques are variations of run-length encoding, in which modified run-lengths for the series are extracted from the data stream and stored in a header, which is used to form the base level of a B-tree index into the database. The run-lengths are cumulative, and therefore the access time of the data is logarithmic in the size of the header. We discuss the details of the compression scheme and its implementation, present several special cases and give an analysis of the relative performance of the various versions.

[1] This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U. S. Department of Energy, under contract DE-AC03-76SF00098.

1. Introduction

The storage of very large databases may constitute a significant portion of the cost of managing them. The compression of data therefore becomes important as the amount of data grows. In addition, data management systems need to provide efficient access to the compressed data. In this paper we explore the compression of large statistical databases, and propose techniques for organizing the compressed data such that the access time required is logarithmic.

The term, ***statistical databases***, is used here as a generic name for numeric databases which are amenable to statistical analysis. Examples are demographic databases, such as the U.S. census, business trend data and results of laboratory experiments. They have three important characteristics which we have exploited in the design of the compression algorithm. The first is the distribution of integer values over a large range, for attributes such as population counts. As a result, the space required for the natural encoding of these attributes varies (one to four bytes depending on the size of the value). In addition, the distribution of values is often skewed towards the lower end, making smaller values more likely. Several schemes have been proposed which compress or encode data to their minimum byte/bit length, but require serial decoding for data access [ALSB75, GOTT75, HUFF52]. Under the compression scheme presented in this paper the data are compressed to a minimum byte size (by suppressing leading zeros) but accessed in logarithmic time.

The second characteristic is the prevalence of a small set of data values, typically zero or a missing data indicator. These values represent a large portion of the database. In such cases it is advantageous to remove the repeating values, which we refer to as *constants*, from the data stream. Here again, many schemes which do so require a sequential decoding of the database in order to search and access the data [ARON77, HAHN74, KNUT73]. Tarjan and Yao

[TARJ79] have developed a row displacement scheme in which access is logarithmic, but the database is not fully compressed. Our approach removes multiple types of constants from the data stream with the same technique which was used for compressing multiple length data.

The third characteristic is the clustered distribution of data values throughout the database. Data of a similar subrange of values, and therefore of a potentially equal length, and data with identical values, i.e., constants, tend to cluster in separate series in the database. Clustering is likely in statistical databases because of the nature of ordering and collecting the data. For example, seismic monitoring data consists of long periods of low activity (generating small numbers or zeros), followed by short periods of high activity. In our compression scheme the clusters form the basic unit for compression.

We have exploited these characteristics of statistical databases to construct a compression method along the lines of run-length encoding. Modified run-lengths are extracted from the data stream and stored in a header, which is used to form the base level of a B-tree index into the database. The run-lengths are cumulative, and therefore the access time is logarithmic in the size of the header. This scheme is a generalization of the specialized compression technique presented in [EGGE80].

Section 2 of this paper describes the general header compression scheme for compressing multiple types of constants and multiple length data. Aspects of the implementation which improve upon the degree of compression and the access are presented in section 3. In section 4, we consider two simplified versions of the general scheme which can be used on special types of databases to achieve better compression, and in some cases, better performance. The first is the basic header scheme for compressing a single constant and single length data [EGGE80], and the second involves the compression of multiple constants and single length data. Section 5 summarizes a comparative analysis of the three schemes.

2. The General Double-Count Header Compression Scheme {DCH} for Multiple Constants and Multiple-length Data (MCML)

A "constant" is defined here to be any value which recurs consecutively a sufficient number of times, so that the space required for its compressed representation is less than that needed for explicitly storing the sequence. The decision to compress a sequence of constants can be made while the data is being compressed by applying a breakeven calculation between the two amounts of storage. By the above definition, any number of different constants can occur in a database. Multiple length data arise because each data item is stored in the minimum number of bytes necessary to contain its value in its natural encoding. The general form of the header compression scheme presented in this paper, which we shall call *the double-count header scheme* {*DCH*}, is capable of compressing both multiple types of constants and multiple length data (MCML).

In order to access data on databases which have been compressed, mappings are necessary between the uncompressed, logical database and the compressed, stored database. In order to execute the mappings, both forms of the data must be represented. The compressed form is, of course, explicitly stored. But the logical form must be computable, because the uncompressed location of the data values must be determined during query processing.

In the double-count header compression scheme the logical, uncompressed form of a database is thought of as consisting of consecutive series of data values. Some of the series are different types of constants which will be compressed in entirety when the stored form of the database is created. Others contain data which can be compressed to a common minimum data length. Both the logical and stored forms are vertically partitioned, i.e., stored by attribute, rather than tuple or row [BATO79, EGGE80, HAMM79], so that the lengths of the series are maximized. Usually the partitioning is complete, i.e., a single attribute in each partition, in order to take advantage of the natural concentration of constants for a single measured attribute.

2.1. Description of the Header

In the double-count header compression scheme the relationship between the logical and physical forms of a database is represented by a double-count header, which consists of entries for each of the various types of series in the logical form. Each double-count entry contains two accumulations: a "logical" count of the number of values in the logical form of the database and a "physical" count of data lengths in bytes in the stored form.[2] Each type of count (logical and physical) is taken at the termination of each series of constants or equal length data values in the logical form.

A database which has been compressed according to the double-count header compression scheme is depicted in Figure 1. LF and SF are the logical and stored forms of the database, respectively. In this example, different sized series are represented by F for four-byte values and I for two-byte values; 2 and 3 are the constants.[3] In the double-count header, H(DCH), the double-count entries are represented by dc_i, where i is the ordinal position of a double-count. The first component of each entry is a bit tag whose purpose is explained below. The second component is a logical count of data values in the logical form, and the third is a physical count of data lengths in the stored form.

The one bit tag indicates whether a particular series contains constants or stored data of a uniform length. We have arbitrarily chosen the set bits to indi-

[2] Bytes, rather than bits, were chosen as the unit of compression, because the smaller unit would have broken up the series and required more complicated processing for accessing the data.

[3] Constants may of course be of different byte lengths as well, but for simplicity we have not shown this in the example.

LF: $F_1\ F_2\ F_3\ I_1\ I_2\ 2\ 2\ 2\ 3\ 3\ 3\ 3\ I_3\ I_4\ I_5\ 2\ 2\ 2\ F_4\ F_5\ F_6\ F_7\ I_8\ I_7\ 3\ 3$

| | | | | | | | |

H(DCH): $dc_1\quad dc_2\quad dc_3\quad dc_4\quad dc_5\quad dc_6\quad dc_7\quad dc_8\ dc_9$

dc_1	=	1:3:12
dc_2	=	1:5:16
dc_3	=	0:8:17
dc_4	=	0:12:18
dc_5	=	1:15:24
dc_6	=	0:18:25
dc_7	=	1:22:41
dc_8	=	1:24:45
dc_9	=	0:26:46

SF: $F_1\ F_2\ F_3\ I_1\ I_2\ 2\ 3\ I_3\ I_4\ I_5\ 2\ F_4\ F_5\ F_6\ F_7\ I_8\ I_7\ 3$

Figure 1

cate multiple length data and the cleared bits to indicate the constants. A single occurrence of a constant for each constant series is actually retained in the stored form of the database to identify the value of the compressed series. Thus, when the tag indicates a series of constants, the physical count associated with that tag is increased by the length in bytes of only one constant. The logical count, however, still reflects the total number of constants in the logical form.

2.2. The Mapping Algorithm

Both the logical and physical counts in the double-count header are cumulative from the beginning of the database. Therefore a binary search can be utilized in the mapping algorithm, and access to the stored data is achieved in logarithmic time.

Let the double-count header be represented as the sequence, $T_0{:}L_0{:}P_0$, $T_1{:}L_1{:}P_1$, $T_2{:}L_2{:}P_2$, ..., $T_i{:}L_i{:}P_i$, ..., $T_n{:}L_n{:}P_n$, where the T's are the bit tags, the L's are logical counts of data values and the P's are the physical counts of data lengths. Let LP and SP be the ordinal positions in the logical and stored forms of the database, LF and SF.

When mapping from a position in the logical form of the database to the corresponding position in the stored form, the binary search for a given LP is done on the logical counts, L_i. The corresponding physical count of data lengths is then used to determine the relative starting byte of the data in the stored form. If the tag in the ith double-count entry, T_i, indicates that the series is one of stored values, the starting byte position, SP, is calculated as follows:[4]

$$SP_1 = \frac{P_i - P_{i-1}}{L_i - L_{i-1}} * (LP - L_{i-1} - 1) + P_{i-1} + 1.$$

If the tag indicates that the series contains constants, the representative constant kept in the stored form of the database can be reached by applying a similar form of the same calculation. Since only one constant is stored, rather than the $L_i - L_{i-1}$ constants which appear in the logical form, the formula is simplified to the following:

$$SP_0 = P_{i-1} + 1.$$

In principle, the stored to logical mapping is also logarithmic, and would be done by a search on the physical counts. In practice, however, the search for LP is subsumed in the query processing which precedes the mapping. Assuming that there is no index on the stored data, locating a desired value requires a

[4] In this formula $\frac{P_i - P_{i-1}}{L_i - L_{i-1}}$ = the number of bytes per element of the series; $LP - L_{i-1} - 1$ = the number of positions that LP is from the beginning of the series; and $P_{i-1} + 1$ = the physical position at the beginning of the series.

sequential search of the data. Since the stored data is of different lengths, the header is needed in order to determine data boundaries and to correctly group the bytes into values. A linear scanning of the header is therefore also required before stored to logical mapping is done. As the stored values are checked, i is appropriately increased. When a qualifying value is found, its LP can be calculated in constant time. (In section 4 we discuss two other schemes in which stored to logical mapping is done in logarithmic time.)

2.3. Load-time Compression Decisions

One would like a compression scheme to have the facility to determine which series of values should be compressed, either entirely, as in the case of constants, or to their minimum byte length, as in the case of multiple length data. It is desirable that the decision to compress be made in one pass during data loading, and be based on the length of the series, rather than pre-selected for the entire database by a database manager. The compression scheme can therefore take advantage of locality in the clustering of certain values or of data of a particular length.

The compression decision is based on a comparison of the savings in space in the stored form of the database versus the storage overhead incurred in additional header space. A typical local decision rule would compare any series with the series immediately preceding it. For the compression of multiple length data the breakeven point occurs when the header storage overhead of compressing a series, i.e., the size of one double-count entry, $(T_i{:}L_i{:}P_i)$, is less than the additional space required to store the series in the units of the previous series,[5] i.e.,

[5] In this formula the additional storage per value is calculated by subtracting the number of bytes per value of the series at i, $\frac{P_i - P_{i-1}}{L_i - L_{i-1}}$, from the number of bytes per value of the series at i-1, $\frac{P_{i-1} - P_{i-2}}{L_{i-1} - L_{i-2}}$. This quantity is then multiplied by the number of values in the logical form,

$$\left(\frac{P_{i-1} - P_{i-2}}{L_{i-1} - L_{i-2}} - \frac{P_i - P_{i-1}}{L_i - L_{i-1}} \right) * (L_i - L_{i-1}).$$

For example, assuming four byte counts, it is better to do minimum byte compression if it will eliminate more than eight bytes from the stored form. For the compression of constants the breakeven point occurs when the extra storage incurred in the header plus the identifying constant is less than the storage of the series of constants, $L_i * (\textit{size of the constant})$. Again, allowing four byte counts and two byte constants, space would be saved by compressing constant series of five or longer.

3. Implementation Considerations

The double-count header compression scheme is currently being implemented in C under the UNIX[6] operating system. The implementation differs from the description given in the previous section of this paper in several respects. In all cases the aim of the difference is to achieve either greater efficiency in access or less header storage overhead.

3.1. Page B-tree for the Header

For large databases each header can take up hundreds, even thousands of pages. To avoid doing the binary search of the mapping algorithm on disk blocks, a B-tree has been imposed upon the header. The B-tree effectively increases the radix of the logarithmic search, thereby greatly reducing the number of disk accesses needed to locate a particular logical position in the header. The binary search is done only on counts *within* a particular page of a tree or header.

Since under the double-count version of the header compression scheme, the binary search is used only when mapping from the logical to the stored

$(L_i - L_{i-1})$.

[6] UNIX is a trademark of Bell Laboratories.

forms of a database, each node of the tree contains only the logical counts of the double-count header. When the logical counts are accumulated by page (cf. section 3.2 below), the tree can be stored as a heap, eliminating the need for page pointers.

3.2. Header Accumulations by Page

Both the logical and physical counts in the header, and consequently the logical counts in each level of the B-tree, accumulate values only within each page, rather than throughout the entire file. An example is illustrated in Figure 2. H(DC) represents the double-count header, where for simplicity the tags and physical counts have been removed and only the logical counts are depicted. The two lines above H(DC) are the tree for the header. Each grouping, in both the header and the B-tree, represents a page. In each page of the header the logical count accumulation is restarted. For example, the second logical count on the second page of the header, {1 4 8}, represents the fourth logical value for that header page, but the twelfth logical value for the entire file. Each entry in a page of the B-tree contains the maximum logical count accumulation of its own subtree and all the subtrees of its left siblings. For example, the accumulated logical count, 12, in the {7 12 23} page of the tree is the maximum accumulation in its own subtree, {2 3 5}, plus that of the subtree of its left sibling, {1 3 7}. And the value, 56, in the root of the tree, {20 43 56}, is the accumulation over the entire header. The overall structure of the B-tree is similar to the partial sum hierarchy used by Bennett and Kruskal [BENN75] in their LRU stack processing algorithm.

```
                         20 43 56

          8 16 20          7 12 23          7 13

H(DC):  3 4 8   1 4 8   2 3 4   1 3 7   2 3 5   3 8 11   3 6 7   1 5 6
```

Figure 2

When descending the B-tree, the value of the logical position which is sought, L_i, must be adjusted at each level to reflect the per page accumulation. Each intra-page binary search terminates at a particular logical count, i. L_i is then decreased by the accumulation in its left sibling, L_{i-1}. This latter value represents the total accumulation in the subtree headed by the logical count at i-1. The search for the adjusted logical position then continues on the next level of the B-tree.

For most pages of a header, the maximum accumulation on a page is small enough so that the size of each count, and therefore the size of the entire header, is halved. Despite the added storage of the B-tree, the storage savings it produces in a header results in a reduction in the total storage overhead incurred by the header compression scheme.

4. Simplified Versions of the Double-Count Header Compression Scheme

The double-count header compression scheme presented in the second section applies to databases in which stored data is compressed to their minimum byte length and multiple types of constants are entirely compressed. For many databases such generality is not needed, and simplified versions of the header compression scheme can be employed which require less storage overhead and occasionally have faster access. There are a number of special cases, the differences between them relating to whether the length of the stored data is fixed or variable and to the number of types of constants. These can be handled with two variations of the double-count header compression scheme. The first treats the basic case in which a single constant is compressed from a database of single length data. The other is a more flexible variation in which multiple types of constants are compressed, but the stored data is still of a fixed length.

4.1. The Single-Count Header Compression Scheme {SCH} for the Case of a Single Constant and Single Length Data [SCSL]

There are numerous applications for compression schemes in which only one type of constant is suppressed and the data which are stored are all the same length. A large number of statistical databases have a prevalence of the constant zero, with the remaining data stored in floating point format. An example is county cancer mortality rates. Other statistical databases are ordered by composite keys[7] whose values constitute an extremely incomplete cross product [EGGE80, SVEN80]. In these databases, (for example, the results of laboratory experiments), the tuples of the incomplete cross product are stored, and the tuples with invalid composite key values are considered the constant and are compressed. For applications of this type, the constant is chosen before the time of data compression. In the latter application, in particular, the size of the "constant" which is compressed, i.e., a tuple, is sufficiently large in most statistical databases that the calculation of a breakeven point for storage savings is superfluous.

[7] Keys which are made up of several attributes.

4.1.1. Description of the Single-Count Header Compression Scheme {SCH}

In [EGGE80] a constant suppression scheme was presented in which all occurrences of one pre-selected constant are eliminated from the database, and the values which are stored are of a single length. Under this scheme, called single-count header compression {SCH}, the consecutive series in the logical form of the database alternate between series of the stored data values and series of the one type of constant. The header which depicts them is composed of alternating entries for the two series. In this scheme the double-count in each entry is reduced to a single-count by keeping separate logical accumulations for each type of series. The counts reflect the number of values of a particular type (either constant or stored) at each point at which the series alternate.

As in the double-count scheme, the counts for each type accumulate from the beginning of the database, and a binary search is utilized in the mapping algorithm. The access of the data is therefore still a logarithmic function. The sum of any two adjacent counts is the logical position at the end of the series associated with the second count in the pair. Thus, when mapping from the logical to the stored form of the database, the search for a given logical position can be done on the sums of adjacent counts. When mapping in the other direction, the search for a given stored position is done only on the counts of stored data. A more detailed description and analysis is given in [EGGE80].

4.1.2. Comparison of the Single-Count {SCH} and Double-Count {DCH} Header Compression Schemes

For applications in which only one type of constant is compressed and the stored data are all of a single length, the double-count header compression scheme requires two and a half times the storage overhead of the single-count scheme. The data length information which was explicitly represented by physical counts in the double-count version is not needed in the single-count scheme because of the fixed length of and the separate logical count for the stored data. Assuming that both logical and physical counts are the same size, the header itself is half as large. In addition, since there is only one constant, there is no need to explicitly identify its value for each constant series, and no constants need be stored in the database.

Let N be the number of counts in the single-count header, L be the size of a logical or physical count and C be the size of a constant. The size of the single-count header is then $H_{SCH} = LN$. The double-count header is $H_{DCH} = 2LN + \frac{N}{2}C$.[8] Since it reasonable to assume that the size of a constant and the size of a header count are identical, the ratio of the two headers is

$$\frac{H_{DCH}}{H_{SCH}} = \frac{2LN + \frac{N}{2}L}{LN} = 2.5.$$

As explained in section 2, when searching databases with multiple length data, the double-count header must be sequentially scanned, prior to performing stored to logical mapping. When the stored data are all the same length, however, the header is not needed to determine data boundaries, and the mapping can be done by a binary search on the physical counts. As before, when mapping from the logical to the stored form of the database, the logical counts are used. In order to minimize each search, i.e., minimize the height of the B-tree and therefore the number of disk accesses, a separate B-tree is used for each type of count. In the single-count scheme the nodes of the B-tree contain pairs of logical counts. The number of internal nodes in its B-tree and the B-tree fan-out are both half that of the double-count tree. As shown below, the storage of the double-count B-tree approaches two times that of the single-count

[8] Since both logical and physical counts are stored in the double-count header, the total number of counts is 2N; since the series of constants and stored values alternate, the number of constants which are stored as tags is $\frac{N}{2}$.

version, as the size of the headers increase.

Let A be the number of pairs of alternating counts in the single-count header[9] T_{SCH} be the total number of *internal* nodes of a full single-count B-tree of height h, and b be the number of nodes per page in the single-count B-tree, i.e., the degree of the tree. Using a bottom-up approach for building the tree,

$$T_{SCH} = \sum_{i=0}^{h-1} \frac{A}{b} * \frac{1}{b^i} = \frac{A}{b} * \frac{\left(\frac{1}{b}\right)^h - 1}{\frac{1}{b} - 1}.$$

For large A and b, $T_{SCH} \approx \frac{A}{b-1}$. In the double-count header each node contains only one count, rather than pairs of counts. The number of nodes per page is 2b, and the number of counts in the header is 2A. The number of internal nodes of *both* double-count B-trees is

$$T_{DCH} \approx 2 * \frac{2A}{2b-1} = \frac{2A}{b-0.5}.$$

As $A \to \infty$, the double-count B-trees approach twice the size of the single-count B-tree.

Despite the savings in storage of the single-count scheme, the usual tradeoff in access time does not occur. The access time between the two schemes is, in fact, quite comparable. Each mapping, regardless of direction or compression scheme, requires one descent of a B-tree. The access for both versions approximates $\log_b A$, where A is either the number of pairs of logical counts in the single-count scheme or the number of logical or physical counts in the double-count scheme.

The approximation can be derived as follows: In the single-count scheme the access for stored data is $A_{SCH} = \log_b A + 2$. Since the constant is known, the access for a constant value is one less. In the double-count scheme one constant per series is stored, and the access time for both data and constants is

[9] In the terms used above, A = N/2.

identical, $A_{DCH} = \log_{2b} 2A + 2$. For large b, $\log_{2b} 2A + 2 \approx \log_b A + 2$.

4.2. The Two Header Compression Scheme {TH} for Multiple Constants and Single Length Data [MCSL]

In many statistical databases attributes can have a range of values which is still fairly restricted, and all the data is therefore of a single length. However, the databases do include several types of constants. The compression requirements for these databases lie between those met by the double-count and single-count schemes presented above. The capability for handling multiple length data in the double-count scheme is not needed, but some facility for differentiating between the different types of constants is required.

4.2.1. Description of the Two Header Scheme {TH}

To handle databases of the type described above, the single-count header scheme can be enhanced to provide a facility for compressing multiple types of constants. The single-count header is modified so that it contains alternating logical counts of stored values and *series of constants of multiple types*. A second header, known as the constant header, is also created to differentiate between the various types of constants. Such a database is depicted in Figure 3, in which L is the logical, uncompressed form of the database, containing values which will be stored (v) and several constants which will be compressed (0, 1, 2). The stored database, containing only the v's is depicted by S; and the single-count header of the two header scheme by H(SCH). As in the single-count header compression scheme, the odd-positioned counts in H(SCH) accumulate stored values. However, rather than accumulating one particular type of constant, the even positioned counts in H(SCH) accumulate all types of constants in the logical form.

```
L:        v_1 v_2 0 0 0 v_3 v_4 1 1 v_5 v_6 v_7 0 0 2 2 2 v_8 v_9 2 2 2 2 2 0 0
               |       |     |   |         |        |     |               |
H(SCH):        2       3     4   5         7       10     9              17

S:        v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9


CS:  0  0  0  1  1  0  0  2  2  2  2  2  2  2  2  0  0
           |     |     |                       |  |
H(CH): T0:3  T1:5  T0:7                     T2:15 T0:17
```

Figure 3

The constant header is based on a logical view of the compressed constants as consecutive series of different types of constants. Each logical count in this header has two components: a tag which indicates which constant has been suppressed, and an accumulation of all constants up to that point in the database.[10] In Figure 3 above H(CH) is the constant header; T0, T1 and T2 are the constant tags, and the numbers succeeding them are the logical counts of constants. The logical form of compressed constants on which H(CH) is based is represented by CS. (Of course, there is no stored data associated with the constant header, H(CH), since CS contains suppressed constants.)

The mapping algorithms for the two header scheme are similar to that for the single-count version. Some additional processing is necessary, however, for locating constants. For logical to stored mapping, a binary search of the con-

[10] Like the double-count header, this header could be considered an accumulated run-length encoding scheme.

stant header is needed to yield the constant type. For stored to logical mapping, the single-count header search is performed on the counts of constants, rather than on the counts of stored values.

4.2.2. Comparison of Two Header {TH} and Double-Count {DCH} Header Compression Schemes

For a comparison of the two header and double-count header compression schemes, let A be the number of pairs of alternating counts in the single-count header of the {TH} version, L and C be the sizes of counts and constants, respectively, and S be the average number of different series of constants represented by any multiple constant count in A. The storage overhead of the two header scheme is $H_{SCH} = 2AL$ for the single-count header plus $H_{CH} = AS(L + C)$ for the constant header. When the double-count scheme is used on applications with multiple constants and single length data, the storage of its double-count header is $H_{DCH} = 2L(A + AS)$; the space for the constants which identify each constant series is ASC. Assuming that the counts and constants are of equal size, the ratio of the two storage requirements reduces to

$$\frac{H_{DCH}}{H_{TH}} = 1 + \frac{S}{2(1 + S)}.$$

As $S \to \infty$, header storage for the double-count scheme increases from 1.25 to 1.5 times that of the two header version. Note that S = 1 is the single constant, single length data situation [SCSL], and the single-count header scheme {SCH} should be used to realize the greater savings (2.5 times greater).

The B-tree storage of the two schemes is comparable. For the two header version, the number of internal B-tree nodes for the single-count header is $T_{SCH} \approx \frac{A}{b - 1}$ and for the constant header, $T_{CH} \approx \frac{AS}{b - 1}$. Each B-tree for the double-count scheme contains $T_{DC} \approx \frac{A + AS}{2b - 1}$ nodes. As $A \to \infty$, the number of internal B-tree nodes in both trees approaches $\frac{A(S + 1)}{b}$.

The access time of the two versions differs depending on the direction of the mapping (logical to stored or vice versa) and whether the output is a stored value or a constant. When mapping from the logical to the stored forms of the database, the access time for the two header scheme is

$$A_{TH} = \log_b A + 2$$

if the output is a stored value and

$$A_{TH} = \log_b A + \log_b AS + 2$$

if a constant. (The second term is for the search of the constant header.) Logical to stored access under the double-count scheme involves the descent of a B-tree whose nodes contain only logical counts. The fan-out of each node is therefore twice as large. The access is

$$A_{DCH} = \log_{2b} (A + AS) + 2$$

for both stored data and constants. For large b, the two header version wins slightly on access of stored data. The double-count scheme, however, has better performance for accessing constants; the two header scheme uses approximately $\log_b A$ more accesses in this case.

For mapping from the stored to the logical forms of the database, the double-count scheme utilizes a B-tree whose nodes contain physical counts,[11] and the time for that search is identical to the double-count logical to stored mapping. If all occurrences of the constants are eliminated from the database, the access of the two header scheme is $A_{TH} = \log_b A + 1$ for both stored data and constants. In this case, the scheme betters the time of the double-count version. However, if a storage breakeven calculation is used in the compression decision, both {TH} headers must be searched, and the general scheme has better access.

[11] Cf. p. 17, 18.

4.3. Variations of the Special Case Schemes, {SCH} and {TH}

Some statistical databases, especially those which are the results of surveys, contain non-key attributes with very small domains. Often the different domain values occur in the database in sufficiently large clusters that they can be considered constants. In this case the data values themselves need not be explicitly stored, but instead can be represented by either a single-count header or a constant header. If the domain of the attribute is two (labeled [2C], for two constants), the alternating logical counts of the single-count header can represent the two types of constants. A typical example in many statistical databases is the values male and female for the attribute sex. If the domain is larger than two (labeled [MC], for multiple constants), only the constant header of the two header scheme need be used to represent the series of different types of constants. An example here are the values for the attribute race.

These variations not only offer the savings in storage realized by the single-count and two header schemes, but also result in a decrease in access time. Since the data is entirely represented in a header, the extra access to the stored database for the constant is eliminated. In addition, in the case of the two header variation, the single-count header is not needed, and only the constant header search is required for all data. The access times for the {SCH} and {TH} variations are $A_{SCH} = \log_b A + 1$ and $A_{TH} = \log_b AS + 1$, respectively. The former is better than the access of the double-count version, and the latter is comparable to {DCH} and improves upon the standard {TH}.

For completeness, we shall consider the cases of a single constant and multiple length data [SCML] and multiple length data without constants [ML]. A slight modification to suppress the storing of the identifying constant per series in the double-count scheme can decrease the storage overhead for these cases by ASC and reduce the logical to stored access of constants by one.

5. Summary and Future Work

A double-count header compression scheme {DCH} and two special case variations, single-count header {SCH} and two header compression {TH}, have been presented, and it has been shown for which types of statistical databases they provide the best storage compression and access performance. Table 1 summarizes these results.

Header Compression Scheme Usage					
Database Characteristics	Compression Schemes				
	double count {DCH}	two header {TH}	single count {SCH}	constant header only {CH}	modified double count
MCML	X				
SCML					X
ML					X
MCSL		X			
SCSL			X		
MC				X	
2C			X		

Table 1

Since the most general version, {DCH}, is the only technique with the capability of handling multiple size data (MCML), it clearly should be used for those statistical databases in which there is variation in the space needed to store the integer attributes. When only one constant is compressed (SCML) or there is no constant compression (ML) from a database with multiple size data, a slight modification (to suppress the storing of the identifying constant per series) decreases the storage overhead of the scheme, and in the (SCML) case, reduces the logical to stored access time for constant values.

Since it offers a header storage savings of 1.25 to 1.5 times that of the double-count scheme, the two header scheme should be used for compressing databases with single size data and multiple constants (MCSL). However, if the database has a preponderance of constants and access time is more critical than storage, the double-count scheme should be used, since its performance in accessing constants is better. For databases in which the attribute domains contain so few values that all values can be considered constants (MC), the single-count header of the two header scheme is not necessary, and the constant header alone can represent the data.

The greatest benefits in storage occur for statistical databases of the simplest type, those with single size data and only one constant (SCSL) or with attributes with only two values (2C). In these cases the single-count header scheme can be used with a storage savings of 2.5 times that of the double-count scheme, but with no loss in access.

It was our intention to develop a compression scheme which not only achieved a high degree of compression, but emphasized fast access. When comparing the header compression schemes with other compression techniques, it is this later factor which we consider most important. In [EGGE80] the single-count scheme was contrasted to two other techniques for compressing constants, run-length encoding and null suppression with bit map. It was found that the former achieved a degree of compression similar to that of the single-count scheme, and the latter was usually less successful. Both, however, required linear access to the data.

The other two versions of the header compression scheme presented in this paper, the two header and double-count schemes, produce similar comparison results. They are functionally analogous to standard run-length encoding and run-length encoding extended to handle multiple length data. The two header scheme requires slightly more storage overhead than its run-length counterpart; the overhead for the double-count and extended run-length schemes are identical. Both run-length variations, however, require linear, rather than logarithmic, access.

Ziv and Lempel [ZIV77, ZIV78] have proposed a universal serial data compression algorithm whose compression efficiency is asymptotically optimal. As described, their scheme requires sequential decoding, hence linear access. In order to achieve fast random access, additional storage for an index is needed. We expect that our scheme will prove competitive in practice on statistical databases, but detailed empirical and theoretical comparisons remain an area for further research.

To conclude, we have exploited common characteristics of statistical databases to create a simple, effective data compression algorithm which provides logarithmic access.

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., June, 1977, 3-5.

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

(4) Bennett, B. T., Kruskal, V. J. "LRU Stack Processing", *IBM Journal of Research and Development*, July, 75, 353-357.

(5) Eggers, S. J., Shoshani, A. "Efficient Access of Compressed Data," *Proceedings of the International Conference on Very Large Databases*, 6, Montreal, 1980, 205-211.

(6) 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.

(7) Hahn, B. "A New Technique for the Compression and Storage of Data," *Communications of the ACM*, Vol. 17, no.8, August, 1974, 434-436.

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

(9) Huffman, D. A. "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of IRE*, Vol. 40, September, 1952, 1098-1101.

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

(11) 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.

(12) Tarjan, R. E., Yao, A. C. "Storing A Sparse Database," *Communications of the ACM*, Vol. 22, no. 11, November, 1979, 606-611.

(13) Ziv, J., Lempel, A. "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, Vol, IT-23, no. 3, May, 1977, 337-343.

(14) Ziv, J., Lempel, A. "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, Vol, IT-24, no. 5, September, 1978, 530-536.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

A RCO OSTI CDL