Transposition Algorithms on Very Large Compressed Databases

Harry K.T. Wong and J. Z. Li*
Lawrence Berkeley Laboratory,
University of California
Berkeley, California

Abstract

Transposition is the dominant operation for very large scientific and statistical databases. This paper presents four efficient transposition algorithms for very large compressed scientific and statistical databases. These algorithms operate directly on compressed data without the need to first decompress them. They are applicable to databases that are compressed using the general (and popular) class of methods called *run-length encoding scheme*. The algorithms have different performance behavior as a function of the database parameters, main memory availability, and the transposition request itself. The algorithms are described and analyzed with respect to the I/O and cpu cost. A decision procedure to select the most efficient algorithm, given a transposition request, is also given. The algorithms have been implemented and the analysis results experimentally validated.

1. Introduction

We are interested in very large Scientific and Statistical Databases (SSDBs) ([Shoshani82], [Shoshani, Olken & Wong84]). SSDBs are prevalent in scientific, socio-economic, and business applications. Examples of SSDBs are experiments and simulation for scientific applications, census, health, and environmental data for socio-economic applications, and inventory and transaction analysis for business applications. These databases typically contain large amount of data in summary form. The main characteristic of such databases is that they contain a combination of descriptive elements for each value of measured (counted, observed, or computed) data.

Supported by the Office of Energy Research, U.S. DOE under Contract No. DE-AC03-76SF00098.

* On leave from Dept. of Computer Science, Heilongjiang Univ., China.

As an example, consider the database in Figure 1 which contains summary data of a multi-factor parametric experiment of corrosion of materials under different conditions such as temperature, acidity, salinity, and duration. The first five attributes (material, temperature, acidity, salinity and time) represent parameter data, the last attribute (corrosion) represents the measured data. The attributes for the parameter data are often referred to as *category attributes*, since they contain category of the measured data. The attributes for the measured data are referred to as *summary attributes*, since they contain data on which statistical summarization procedures are applied.

Typical queries of a database such as Figure 1 involve the retrieval of summary attribute values given some specific combination of the category values (What is the corrosion level for steel in temperature 1500, acidity level of 200, salinity level of 5, and time of 10 units?). To facilitate searching, the database is typically sorted by the category attributes in row-wise fashion (i.e., the values of the rightmost attribute vary the fastest).

Two factors cause these SSDBs to be extremely large. First, they may contain hundreds of summary attributes. Secondly and more importantly, the cardinalities of the category attributes can themselves be quite large; and the number of tuples generated is the product of these cardinalities. For example, the mortality database from the National Institute of Health contains the cross product of four races, two sexes, 70 diseases, six age groups, and 3000 counties, amounting to over ten million tuples.

2. Motivation

The most common operations operating on summary databases (besides searching) are *transposition* and *aggregation*. The former requests an re-ordering of the category attributes for the purpose of presentation and analysis. An example from the database in Figure 1 is to transpose the database so that now temperature and acidity are after material, salinity and time. Transposition operations are also required to obtain the popular file structure called *transposed file* ([Batory79]). Transposed files are the most efficient file structure for many SSDB applications. The motivation of transposed files is that the access to SSDBs is typically long sweep, i.e., a long sequence of individual records is fetched and a small number of attributes extracted. By storing the records as a collection of contiguous attribute columns, i.e., all of the data for a field (attribute) is stored together, only those attribute columns which are needed for a query need be retrieved. We assume also, in this paper, that transposed files are used to store the data in the summary SSDBs.

Aggregation operations are used to "collapse" away some category attributes to obtain a more concise database to facilitate more efficient analysis. An example of aggregation is a request such as "What is the total corrosion level of steel in temperature 1000, acidity 100, and salinity 1?" Since the dimension time is ignored in the request, the corrosion values is aggregated on the time dimension. The answer to the above request is obtained by summing the corrosion level values over all time values for each combination of the other category attributes.

Efficient methods of performing transposition and aggregation are the keys to have an efficient SSDB system to support data analysis. Since most large summary SSDBs are typically compressed, efficient transposition and aggregation methods directly over compressed data without first decompressing are important.

Note that transposition and aggregation operations are closely related. An aggregation operation on attribute A can be realized by first transposing A from its original position to the right of the rightmost category attribute in the database, then the corresponding summary attribute values are aggregated (typically by simple arithmetic operations such as sum, weighted average, etc.). For example, to collapse the temperature dimension from the database involves transposing the attribute to the right of the time attribute, then the corrosion values are aggregated. In this paper, we describe several efficient techniques for performing transposition on compressed summary SSDBs. The results obtained can be extended to the aggregation operation mentioned above.

In section 2, the related area is surveyed and in section 3, some background information about compression is given for the rest of the paper. In section 4, description and analysis of four transposition algorithms are given. In section 5, a decision procedure is developed that will select the most appropriate algorithm for a given transposition request. Section 6 describes our implementation effort and section 7 summarizes the paper and draws some conclusions.

3. Related Work

Almost all SSDB management systems (such as [Turner et al.79], [McCarthy et al.82], [SAS79]) perform transposition over compressed data by first decompressing the data, then transposing the full (typically very sparse) cross-product, and finally recompressing the database. For very large SSDBs, the user may have to wait days or even weeks for the operation to finish. What is needed is algorithms which manipulate compressed data directly.

Several researchers ([Epstein79] and [Klug82]) have tackled the problems of processing aggregates in the relational database context. The reported techniques are not applicable to summary SSDBs since no compression is assumed on the databases and the emphasis is on query optimization.

[Floyd72] gives a very interesting transposition algorithm for dense square matrices residing on disk. Given a matrix of size P by P, the algorithm uses 2 buffers of size P each to transpose the columns of the matrix to rows. Since the matrix is not compressed, a mathematical formula is given to decide where each data item should be moved. The algorithm requires I/O operations in the order of $O(P \log_2 P)$.

[Tsuda et al.83] extends Floyd's algorithm to handle rectangular matrices (still 2-dimensional). The method is to divide the matrix into a multiple of square matrices (the last matrix may have to be padded with nulls to make it square) and Floyd's algorithm can be applied on each. The algorithms presented in this paper have the same order of I/O performance as the algorithms presented in [Floyd72] and [Tsuda et al.83]. But our algorithms can work on compressed *multi-dimensional* databases.

4. Compression of SSDBs

In this section, the concepts and terms of the compression methods we use are introduced. They formulate the background for the algorithms in the next section.

Summary SSDBs such as the one displayed in Figure 1 have a great deal of redundancy in the values of the category attributes. In many databases all possible combinations of the category attributes (i.e., the full cross-product) exist. In such cases, each value of a category attribute repeats as many times as the product of the cardinalities of the remaining category attributes.

A method which eliminates the need to store category attributes is used. This method stores the list of distinct category attribute values of each attribute once. Then, each category attribute can be used to form one dimension of a multi-dimensional matrix. For each combination of values from the category attributes, one can compute the appropriate position in the matrix. A well-known algorithm (called *array linearization*) provides such a mapping. This method transforms a query on the category attributes into a computation of *a logical position* in a linearized matrix. Array linearization is reversible in the sense that given a position in the matrix, there is a unique combination of category attribute values identifying it (this process is called *reverse array linearization*).

Summary attribute values can be quite sparse. As an example, refer to Figure 1. Suppose that temperature does not have effect on certain type of material. then in the corrosion column there would be the same value in consecutive positions for all the acidity, salinity, and time. Here a compression method called *header compression* ([Eggers & Shoshani80]) is used to remove the repeated values by a count and provide efficient access to the compressed data. This method makes use of a header which contains the counts of both compressed and uncompressed sequences in the data stream. The counts are organized in such a way as to permit a logarithmic search over them. A B-tree is built on top of the header to achieve a high radix for the logarithmic access. In addition to the header file, the output of the compression method consists of a file of compressed data items, called *physical file* (the original file, which is not stored, is called *logical file*). Two mappings are provided by the compression method, one is called *forward mapping,* which computes the location in the physical file given a position of the logical file. The other mapping (called *backward mapping*) provides the physical to logical direction. These mappings can be performed in logarithmic time because of the existence of the header. For a more thorough discussion of the header compression method, refer to the original paper.

To make the description of the algorithms more concrete, we assume that each summary attribute of the database has been compressed using the header compression scheme. But an important note is that the algorithms are general in the sense that they can be applied to databases that are compressed using the general class of methods called *run-length encoding* scheme ([Aronson77]), where a repetition of data items is replaced by a count and a value of the data item. Header compression method is just a variation of the run-length encoding scheme. Also, we assume that the category attributes are compressed away by array linearization as mentioned before.

5. Transposition Algorithms

In this section the four transpositions algorithms will be described in detail. Below the main idea and the applicability of each algorithm will be briefly highlighted. The description sections below provide more details for each algorithm.

The first algorithm is a "general" algorithm in the sense that it can be used in all situations. First, the physical database is read, and for each data item, a "tag" is computed and stored with the data item on disk. A tag is the logical sequence number for the data item in the transposed space. The second step involves sorting the tag and data item pairs in ascending order of the tags. After the sorting is done, the tags associated with the data item are discarded. As the tags are stripped, the necessary headers for the data items are generated and these headers and the data items represent the result of the transposition.

The second algorithm performs the operation in main memory in one pass. This is feasible in the event when the transposed subspace is small enough to fit into main memory. The main idea of the algorithm involves scanning the physical database once, and employing the reverse array linearization to find the proper slot for each data item in the memory buffer. A compression algorithm will then run over the data in memory and the result is stored in compressed form on disk.

For the case that the transposed subspace is too large to fit in main memory, a third algorithm can be used. The algorithm takes advantage of the situation when there are a small number of large fragments of transposed subspace that are already in the right position. The algorithm involves the merging of these fragments, and compressing of the result. This algorithm is used instead of the first algorithm if the number of fragments is small. A more quantitative treatment is given in a later section.

A fourth algorithm takes advantage of the situation when the cross-product of the cardinalities of the transposed attributes are relatively small and they are moved as a group. In this situation, N buffers are used to store the temporary result of transposition where N is equal to the product of cardinalities of the transposed attributes. This algorithm is slower but not as memory intensive as the second algorithm. But when applicable, it offers better performance than the first and third algorithms.

The algorithms are listed in the appendix. For the rest of the paper, we will use the following symbols for the relevant compressed database parameters:

N: size of compressed summary data file.
M: number of category attributes.
W: number of buffers.
B: size of buffers and blocks.

5.1. Algorithm GENERAL

5.1.1. Description

This algorithm assumes W buffers each with size B are available. Data from CSF are read into the buffers. For each data item in each buffer, the following is done: Backward mapping is performed to obtain the logical position in the original category attribute space; a reverse array linearization is computed to recover the values of the attributes; and finally a new logical number in the transposed space is computed using the array linearization operation. This new logical number (called a "tag") is then stored with the data item in the buffer. An internal sort is performed on each of these buffers with respect to the tags of the data items. The sorted data items in these buffers are next merge-sorted into a single run and written out to disk along with the tags. This process is repeated for the rest of the blocks in CSF. The runs of data items and their tags are next merged using, again, W buffers. A new header file is constructed for the transposed file in the final pass of the merge sequence. Also, the tags associated with the data items are discarded in this pass. The file produced containing just the (shuffled) data items is the new transposed CSF file.

5.1.2. Analysis

5.1.2.1. Block Accesses

Algorithm GTRANSPO has two major parts as far as I/O activities are concerned. The first part is where all the compressed data is read and sorted by the new logical positions using W buffers. The result of this part is a set of sorted subruns. The second part of the algorithm merges these subruns and compresses them in the last pass of the merge using W buffers. The more precise I/O behavior of this algorithm is summarized as follows.

The reading of the original compressed file and writing out of the sorted subruns require $2\lceil N/B \rceil$ block accesses. The second part of the algorithm requires the merging of the $\lceil N/B/W \rceil$ subruns using W buffers. Hence there are $\log_W \lceil N/B \rceil$-1 passes over the data. Here a buffering scheme is used so that in the odd (even) pass, block reading is done from the first (last) block to the last (first) block. One block can be saved from reading and writing by keeping the first or last block in memory to be used by the subsequent pass. Therefore, there are

$$2(\lceil N/B \rceil - 1) \times (\lceil \log_W N/B \rceil - 1)$$

blocks to read and write.

Finally, the original header file has to be read to compute the logical number and new header file needs to be built, hence we have N_o blocks to read and N_n blocks to write. Therefore, the number of block accesses is

$$2(\lceil \log_W N/B \rceil \times (\lceil N/B \rceil - 1) + 1) + \left\lceil \frac{N_o + N_n}{B} \right\rceil .$$

5.1.2.2. CPU Cost

In the first part of the algorithm, for each value in the summary data file, we need to perform one reverse array linearization and one array linearization. There are also $\lceil N/B \rceil$ blocks, each with size B, to sort and $\lceil N/B/W \rceil$ merges, each with W runs of size B, to merge.

An array linearization operation requires

$$2(M-1)$$

multiplications and additions.

An reverse array linearization operation requires
$$2(M-1)$$
divisions and subtractions.

To sort a block with size B requires
$$B\log_2 B$$
comparisons.

To merge W runs each with size B requires
$$WB\log_2 W$$
comparisons.

The total number of cpu operations, therefore, for the first part (steps 3 to 17) is
$$4N(M-1)+\lceil N/B\rceil\times\lceil B\log_2 B\rceil+\lceil N/B/W\rceil\times\lceil WB\log_2 W\rceil.$$
In the second part of the algorithm, we need to perform $\lceil\log_W N/B\rceil-1$ iterations where each iteration involves the merging of $\lceil N/B/W^i\rceil$ runs each with size W^iB. Thus, in the second part,
$$(\lceil\log_W N/B\rceil-1)\times\lceil N/B/W\rceil\times\lceil WB\log_2 W\rceil$$
comparisons are needed.

The total number of cpu operations of GTRANSPO is therefore,
$$N\times(4(M-1)+\lceil\log_2 N\rceil).$$

5.2. Algorithm MTRANSPO

5.2.1. Description

This algorithm requires a buffer large enough to hold the subspace from R_{I_0} to R_M. The algorithm steps through the non-transposed portion of the database, i.e., the subspace from R_1 to R_{I_0-1}. For each "point" in this fixed space, transposition is performed as follows. Data is read in one block at a time. Tags are computed as described before. Each data item is stored into the buffer using the corresponding tag as an index. When the subspace is exhausted, headers are generated and stored and the buffer is written out. This represents the result of partial transposition under this fixed "point" of the non-transposed space. These partial results are accumulative in the sense that they can be concatenated to form the final transposition result without any more passes over them. The reason is that the non-transposed portion of the space is stepped through in the same order that the original data is stored, i.e., the rightmost index is varying the fastest.

5.2.2. Analysis

Algorithm MTRANSPO requires the reading of the original summary data and writing of the resulting transposed summary data file. Also, the reading of the original header file and writing of the new header file are needed. Hence, the total I/O cost is
$$2\lceil N/B\rceil+\left\lceil\frac{N_o+N_n}{B}\right\rceil.$$

The cpu cost of MTRANSPO is for each value in the summary data file, the cost for performing an array linearization and a reverse array linearization. Hence, the number of cpu operations is simply
$$4N(M-1).$$

5.3. Algorithm STRANSPO

5.3.1. Description

This algorithm takes advantage of the situation where there are a small number of large fragments of transposed subspace that are already in the right position. As a result, no sorting is required on the fragments and the merging is needed on a smaller number of sorted runs.

As an example, consider Figure 2, where two examples of transpositions are shown on a file with four category attributes (called A, B, C, D). The LP columns represent the logical numbers of each row of the category attributes. The first example moves the attribute B from the second column to the right of D ((b) of Figure 2). Notice that the LP column of (b) contains 2 sorted runs (the cardinality of B) each with 6 (the product of the cardinalities of C and D) elements in it. The second example exchanges columns B and D. Again, notice that the LP column of (c) of Figure 2 where there are 6 subruns (the product of cardinalities of A, C and D) each with 2 (the cardinality of D) elements. These phenomena are generalized into the lemmas below, and for space reason, the proofs are omitted.

Lemma 1.

If $R_1 \cdots R_{i-1} R_i R_{i+1} \cdots R_j R_{j+1} \cdots R_M \rightarrow R_1 \cdots R_{i-1} R_{i+1} \cdots R_j R_i R_{j+1} \cdots R_M$,
then

(The symbol "→" is read as "is to be transposed to")

(1) The number of subruns $= |R_i|$;

(2) The length of each subrun $= |R_1|\times|R_{i-1}|\times|R_{i+1}|\times\cdots\times|R_M|$.

(3) The subscripts of the boundary of each subrun are

$r_1\cdots r_{i-1}1 1\cdots 1$

$r_1\cdots r_{i-1}2 1\cdots 1$

.

.

.

$r_1\cdots r_{i-1}k_i 1\cdots 1$

where $R_i=\{1,\ldots,k_i\}$, r_m is in R_m for m=1 to i-1.

This lemma summarizes the patterns of transposition involving the movement of attributes to the right. It presents the expected number of subruns, the length of the each subrun and the boundary of each subrun in terms of a single attribute being transposed to the right. Generalization of this lemma involving the transposition of a subspace of attributes is straightforward.

Lemma 2 below presents the pattern of transposition when two attributes are exchanged in their category attribute space. Again, the generalization of this lemma to more than one attribute is used in our implementation.

Lemma 2.

If $R_1 \cdots R_{i-1} R_i R_{i+1} \cdots R_{j-1} R_j R_{j+1} \cdots R_M \rightarrow R_1 \cdots R_{i-1} R_j R_{i+1} \cdots R_{j-1} R_i R_{j+1} \cdots R_M$,
then

(1) The number of subruns $= |R_1|\times\cdots\times|R_{j-1}|$;

(2) The length of each subrun $= |R_j|\times\cdots\times|R_M|$;

(3) The subscripts of each subrun begins at

$r_1\cdots r_{j-1}1\cdots 1$

where, r_m is in R_m for m=1 to j-1.

The key step of the algorithm STRANSPO is to use these two lemmas to compute the number of subruns and the boundary of each subrun. After that, it is basically a merge sort algorithm for SNUM subruns using W buffers. The first pass over the database involves constructing the tags for each data item as before. The final pass of the algorithm discards the tags and header counts are generated similar to the first algorithm.

5.3.2. Analysis

5.3.2.1. Block Accesses

The I/O performance of STRANSPO is based on the number of subruns (SNUM in the algorithm). Since there are SNUM subruns needed to be merged and there are W buffers, the number of passes required to go over the $\lceil N/B \rceil$ blocks of data is $\lceil \log_W SNUM \rceil$. This plus the reading and constructing of header files brings the total of blocks accesses to be:

$$2(\lceil \log_W SNUM \rceil \times (\lceil N/B \rceil - 1) + 1) + \left\lceil \frac{N_e + N_n}{B} \right\rceil.$$

Note the same buffering scheme mentioned in GTRANSPO is used to save one block of I/O per pass.

5.3.2.2. Cpu Cost

The cpu performance of the algorithm is similar to GTRANSPO except that there is no need to sort the data in the buffers and instead of having $\lceil N/B \rceil$ runs to merge, we have *SNUM* runs. The total number of cpu operations, therefore, is equal to:

$$N \times (4(M-1) + \lceil \log_2 SNUM \rceil).$$

5.4. Algorithm LTRANSPO

5.4.1. Description

This algorithm requires memory space to hold N buffers where N is the size of the transposed subspace. Unlike the algorithm M, where space is required to hold the entire subspace from the leftmost transposed attribute (R_{I_0}) to the rightmost attribute of the database (R_M), this algorithm requires buffer space for subspace starting at R_{I_0} to R_j, the rightmost transposed attribute. Similar to the algorithm M, the non-transposed subspace is stepped through in the row-wise fashion. For each data item, the reverse array linearization operation is performed to identify the correct buffer to which the data item belongs. This algorithm also requires N temporary files to store the overflowed buffers. These N temporary files are merged and header file generated when the original data file is exhausted.

Two general cases of the algorithm are presented in the section below. These two cases are distinguished according to the transposition direction of the group of transposed attributes. The first and second cases represent respectively the left and right direction movement of a group of attributes.

Assume the additional input parameter D, which represents the number of buffers needed in the algorithm. The value of D can be computed as below:

$$D = |R_j| \times \dots\dots \times |R_{j+k}| \;\; or \;\; |R_{i+k+1}| \times \dots\dots \times |R_j|.$$

Algorithm LTRANSPO recognizes the following two situations.

$$(1) R_1 \cdots R_{i-1} R_i \cdots (R_j \cdots R_{j+k}) R_{j+k+1} \cdots R_M \rightarrow R_1 \cdots R_{i-1} (R_j \cdots R_{j+k}) R_i \cdots R_{j-1} R_{j+k+1} \cdots R_M;$$

$$(2) R_1 \cdots R_{i-1}(R_i \cdots R_{i+k}) R_{i+k+1} \cdots R_j\, R_{j+1} \cdots R_M \rightarrow R_1 \cdots R_{i-1} R_{i+k+1} \cdots R_j\, (R_i \cdots R_{i+k}) R_{j+1} \cdots R_M;$$

5.4.2. Analysis

The number of I/O required is equal to 4 times the total number of blocks of the database. The reason is that the temporary files have to be read and concatenated into a single file. Also, there may be up to *kd* more blocks to write to disk when the buffers are not full but the data stream is exhausted, where k is equal to product of the non-transposed space (i.e., from R_1 to R_{i-1}). This plus the reading and constructing of the header file bring the total of I/O operations to

$$4\lceil N/B \rceil + \left\lceil \frac{N_e + N_n}{B} \right\rceil + kd.$$

The number of cpu operations required is just

$$2N(M-1)$$

since only N reverse array linearization operations are needed.

6. Comparing the Basic Algorithms

In this section a partial order among the four algorithms is constructed in terms of I/O and cpu cost. In the following observations, the symbol ">>" is defined as a short hand notation for "is more expensive than". Also, the algorithms will be referred to by their first letter.

6.1. Observations

Observation 1. G >> M, S >> M, and G >> L.

Justification.

(1) G >> M.

The block access difference between G and M is

$$2((\lceil N/B \rceil - 1) \times (\lceil \log_W N/B \rceil - 1)).$$

Since we are interested in very large databases, typically $\lceil N/B \rceil > W$, thus, IO(G) > IO(M).

The cpu time difference is

$$N \log_2 N$$

which is > 0. Hence cpu(G) > cpu(M). Therefore we have G >> M.

(2) S >> M.

$$IO(S) - IO(M) = 2(\lceil N/B \rceil - 1)(\log_W SNUM - 1).$$

Generally, SNUM > W and $\lceil N/B \rceil > 1$, thus IO(S) > IO(M).

$$cpu(S) - cpu(M) = N(\log_2 SNUM) > 0.$$

cpu(S) > cpu(M). Hence (2) is justified.

(3) G >> L.

$$IO(G) - IO(L) = 2((\log_W \lceil N/B \rceil - 2)(\lceil N/B \rceil - 1) - (kd+2)).$$

Since $\lceil N/B \rceil$ is typically much larger than W, k, and d, we have

$$\log_W \lceil N/B \rceil > \frac{kd+2}{\lceil N/B \rceil - 1} + 2.$$

Thus IO(G) > IO(L).

$$cpu(G) - cpu(L) = N(2(M-1) + \lceil \log_2 N \rceil) > 0.$$

Hence we have (3).

Observation 1 gives a partial order of preference in terms of performance. But the memory requirements of M and L should be important considerations. M requires memory space equal to the entire full subspace from the leftmost attribute to be transposed to the rightmost attribute of the database. L requires memory space to hold D buffers where D is equal to the size of the subspace bounded by the leftmost and rightmost attributes to be transposed. In very large databases, M or L may not be applicable for transposition requests which exceed the available memory in the user environment. In such cases, either G or S should be used. A decision procedure will be described to choose the best possible algorithm for a given transposition request.

Observation 2.

(1) If $\log_W SNUM > \frac{kd+2}{\lceil N/B \rceil - 1} + 2$ then S >> L else L >> S.

(2) If $\lceil N/B \rceil > SNUM$ then G >> S else S >> G.

Justification.

(1) We know that

$$IO(S) - IO(L) = 2((\log_w SNUM - 2)(\lceil N/B \rceil - 1) - (kd+2)).$$

and,

$$cpu(S) - cpu(L) = N(2(M-1) + \log_2 SNUM$$

$$= O(N \log_2 SNUM) > 0.$$

If the condition of (1) is true, then S >> L. Otherwise, IO(S) - IO(L) is $-O(N \log_W SNUM)$. Since the differences of I/O and cpu times are the same order, the I/O cost should be the more dominant consideration, hence L >> S in this case.

(2) We also know that

$$IO(G) - IO(S) = 2\log_W \frac{\lceil N/B \rceil}{SNUM}(\lceil N/B \rceil - 1).$$

and,

$$cpu(G) - cpu(S) = N \log_2 \frac{N}{SNUM} > 0.$$

If $\lceil N/B \rceil > SNUM$, then G >> S. Otherwise, the savings of cpu time of S over G are not enough to offset the extra block accesses of S over G, hence we have S >> G in this situation.

Intuitively, the performance of S depends very much on the value of SNUM. As a rule of thumb, algorithm S is attractive if the value of SNUM is small. Since a small SNUM value will indicate long subruns, as a result, less passes will have to be done over the data. Observation 2 gives the formal criteria of choosing between S and G as well as between S and L.

Observation 3 L >> M.

Justification.

$$IO(M) - IO(L) = -(2\lceil N/B \rceil + kd) < 0.$$

$$cpu(M) - cpu(L) = 2N(M-1) > 0.$$

Similar to the justification of (2) of Observation 2, L << M overall.

6.2. A Select Procedure

Below a decision procedure is given which is based on the three observations above to select the cheapest transposition algorithm.

Algorithm DECIDE.

```
IF available memory satisfies M THEN select M
  ELSE IF available memory satisfies L
       and $\log_W SNUM > \frac{kd+2}{\lceil N/B \rceil - 1} + 2$ THEN select L
       ELSE IF $\lceil N/B \rceil > SNUM$ THEN select S
            ELSE select G.
```

7. Implementation

All four transposition algorithms have been implemented using C in a VAX/VMS environment. The Observations given above have all been experimentally validated. These algorithms and the above DECIDE program are now an integral part of our experimental SSDB management system MICSUM ([Wong & Li86]).

8. Summary and Conclusion

Transposition is the dominant operation in many SSDBs. In this paper, a collection of efficient transposition algorithms have been described and analyzed. These algorithms operate directly on compressed data without the need to first decompress them. The methods proposed are applicable to databases that are compressed using the general method of run-length encoding. A decision procedure is also given to select the most efficient algorithm based on the transposition request, available memory, as well as the database parameters. Formulas have been developed which identify the required memory space, the length of the subruns and the number of expected subruns. The algorithms have the same order of I/O performance as that of [Floyd72] and [Tsuda et al.83] where only dense 2-dimensional matrices are dealt with. The algorithms presented can operate on compressed multi-dimensional databases. Since aggregation operations can be developed on top of transposition operations, the result of this paper can be applied directly to efficient aggregation algorithms on compressed data.

In conclusion, direct manipulation over compressed data is an important concept where great efficiency can be achieved. Algorithm need to be developed and analyzed for operators on compressed data. Transposition is just one (and important) such operation in this direction. We are now researching on other operators such as searching, aggregation, and other higher level statistical operators on compressed data.

Acknowledgements

We would like to thank our colleagues Frank Olken and Doron Rotem for their input to the problem and Arie Shoshani for his valuable comments to the paper.

Appendix

Algorithms In this appendix, the algorithms are described in a pseudo- programming language. We will use the following notations for the relevant compressed database parameters:

CSF: Compressed summary data file.
SHF: Summary data header file.
C[i]: Cardinality of the i^{th} category attribute, i=1 to M.
N: size of compressed summary data file.
M: number of category attributes.

The following parameters are assumed to be available for each transposition request:

W: number of buffers.
B: size of buffers and blocks.
A[i]: Transposition assignment for category attribute i, i=1 to M.
E.g., A[5] = 3 implies that the 5^{th} category attribute is to be transposed to be the 3^{rd} category attribute.

Also, routines that perform the backward mapping from the compressed physical file to the logical file, array linearization, and reverse array linearization are referred to respectively BMAP, LIN, and REV_LIN.

```
(1) FOR  i=1  TO  M  DO
(2)   NC[i] = C[A[i]]
(3) FOR  i=1  TO  ⌈N/B/W⌉  DO
(4)     BEGIN
(5)       FOR  j=1  TO  W  DO
(6)         BEGIN
(7)           read ((i-1)*W+j)th block of CSF into buffer[j];
(8)            FOR  each value v in buffer[j]  DO
(9)              BEGIN
(10)               look up v's logical position using BMAP;
(11)               compute subscripts using REV_LIN
                        and store to array z;
(12)               reassign z according to array A;
(13)               compute new logical position using z and NC;
                     and store with v in buffer[j];
(14)             END
(15)          sort buffer[j] in order of logical positions;
(16)        END
(17)     merge the W runs in buffer[1],...,buffer[W] into a single run;
         (if ⌈N/B⌉=W, calculate header counts and write to new header file.)
(18)   END
(19) FOR  i=1  TO  ⌈log_W N/B⌉-1  DO
(20)   merge the ⌈N/B/W^i⌉ runs formed in step(17) or (20);
       (if i = ⌈log_W N/B⌉-1,compute headers and write to new header file.)
```

Algorithm GTRANSPO

Let I_0 denote the index of the leftmost attribute to be transposed.

```
(1) FOR  i=1  TO  M  DO
(2)   NC[i]=C[A[i]];
(4) FOR  each element in cross product R_1, . . . , R_{I_0-1} in ascending order  DO
(5)   BEGIN
(6)       IF  bufferin is empty  THEN  read a block of CSF to bufferin;
(9)       FOR  each value v in bufferin  DO
(10         BEGIN
(11            look up v's logical position using BMAP;
(12            compute the subscripts using REV_LIN
          and store to array z;
(13)           reassign z according to array A;
(14)           compute the new logical position p  using LIN
                    with z and NC as parameters;
(15)           buffer[p]=v;
(16)        END
(18)      write values in buffer to result file, and calculate the
               header counts and write to the new header file;
(19)  END
```

Algorithm MTRANSPO

```
(1) FOR  i=1  TO  M  DO
(2)   NC[i]=C[A[i]];
(3) SNUM=number of subruns using Lemmas 1 and 2;
(4) FOR  i=1  TO  ⌈SNUM/W⌉  DO
(5)   BEGIN
(6)     FOR  j=1  TO  W  DO
          compute the ((i-1)*W+j)^th subrun's boundary
          and compute the boundary's new logical position;
(7)     WHILE  one of the ((i-1)*W+1)^st ,((i-1)*W+2)^nd ,
            ..., and ((i-1)*W+W)^th subsuns is not at end  DO
(8)       BEGIN
(9)         IF  buffer[j] is empty and ((i-1)*W+j)^th subrun
              is not at end (for j=1 to W)     THEN
(10)          BEGIN
(11)            read a block of ((i-1)*W+j)^th run to buffer[j];
(12)             FOR  each value v except boundary in buffer[j]  DO
(13)               BEGIN
(14)                 look up v's logical position using BMAP;
(15)                 compute subscripts using REV_LIN
                          and store to array z;
(16)                 reassign z according to array A;
(17)                 compute the new logical position using LINEAR
                          and store with v in buffer[j];
(18)               END
(19)          END
(20)        merge the W runs in the buffers into single run;
           (if ⌈SNUM/W⌉=1,compute headers and write to new header file.)
(21)      END
(22)  END
(23) FOR  i=1  TO  ⌈log_W SNUM⌉-1  DO
(24)   merge the ⌈SNUM/W^i⌉ runs formed in step(20) or (24);
     (if i = ⌈log_W SNUM⌉-1,compute headers and write to new header file.)
```

Algorithm STRANSPO

```
(1) FOR each combination $z_1 \cdots z_{i-1}$
        in the cross product of $R_1 \cdots R_{i-1}$ increasingly   DO
(2)     BEGIN
(3)       read one block of CSF into bufferin;
(4)        FOR each value v in bufferin DO
(5)          BEGIN
(6)            look up v's logical position using BMAP;
(7)            compute subscripts using REV_LIN
                      and store to array z;
(8)            buffer[$z_j, \ldots, z_{j+k}$]=v
                 (or buffer[$z_{i+k+1} \cdots z_j$]=v);
(9)             IF this buffer is full  THEN
                    write to file $TSF[z_j, \ldots, z_{j+k}]$
                    (or $TSF[z_{i+k+1}, \ldots, z_j]$);
(10)         END
(11)      FOR  i=1  TO  D  DO
(12)         IF  buffer[i] is not empty  THEN
                 write to $TSF[i]$;
(13)      FOR each $z_j \cdots z_{j+k}$ (or $z_{i+k+1} \cdots z_j$) increasingly DO
            read $TSF[z_j, \ldots, z_{j+k}]$ (or $TSF[z_{i+k+1}, \ldots, z_j]$);
            write sequentially to result file;
              compute header counts and write to new header file;
(14)   END
```

Algorithm LTRANSPO

References

Shoshani, A., "Statistical Databases: Characteristics, Problems and Some Solutions", Proc. 1982 International Conference on Very Large Data Bases, Mexico City, Mexico, Sept., 1982.

Shoshani, A., Olken, F., Wong, H.K.T., "Characteristics of Scientific Databases", Proc. 1984 International Conference on Very Large Data Bases, Singapore, Sept., 1984.

Turner, M. J., Hammond, R., Cotton, F., "A DBMS for Large Statistical Databases", Proc. 1979 International Conference on Very Large Data Bases, Rio de Janeiro, Brazil, Sept., 1979.

SAS Institute, Inc., *SAS User's GUIDE,* 1979 Edition, Raleigh NC.

McCarthy, J., Merrill, D.W., Marcus, A., Benson, W.H., Gey, F.C., Holmes, H., "SEEDIS: The Socio-Economic Environmental Demographic Information System", in *A LBL Perspective on Statistical Databases* LBL Technical Report 15393, Dec, 1982.

Eggers, S., Shoshani, A., "Efficient Access of Compressed Data", Proc. 1980 International Conference on Very Large Data Bases, Montreal, Canada, Sept, 1980.

Aronson, J., *Data Compression - A Comparison of Methods,* Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C., 1977, pp 3 -5.

Klug, A., "Access Paths in the Abe Statistical Query Facility", Proc. 1982 SIGMOD Conference, Orlando, Florida.

Epstein, R., "Techniques for Processing Aggregates in Relational Database Systems", Electronics Research Lab. UCB/ERL M79/8, Univ. of Calif., Berkeley.

Klug, A., "Abe -- A Query Language for Constructing Aggregates-By- Example", Workshop on Statistical Database Management, Menlo Park, Calif., Dec. 1981.

Tsuda, T., Sato, T., "Transposition of Large Tabular Data Structures with Applications to Physical Database Organization", Part I, Acta Informatica 19, 13-33 (1983), Part II, Acta Informatica, 19, 167-182 (1983).

Batory, D.S., "On Searching Transposed Files", ACM TODS 4, 531-544, 1979.

Floyd, R.W., "Permuting Information in Idealized Two-Level Storage", *Complexity of Computer Computations* R. Miller and J. Thatcher, editors, pp 105-109, New York, Plenum Press 1972.

Wong, H.K.T., Li, J. Z., "An Experimental SSDB system MICSUM", Working document.

Material	Temprature	Acidity	Salinity	Time	Corrosion
Steel	1000	100	1	10	.7
Steel	1000	100	1	20	.9
"	"	"	"	.	1.2
"	"	"	"	.	1.5
"	"	"	"	.	1.7
"	"	"	"	100	2.3
"	"	"	2	10	.8
"	"	"	2	20	1.0
"	"	"	"	.	1.2
"	"	"	"	.	1.5
"	"	"	"	.	1.8
"	"	"	"	100	2.4
"	"	"	.	.	.
"	"	"	.	.	.
"	"	"	.	.	.
"	"	200	.	.	.
"	"	.	.	.	.
"	"	.	.	.	.
"	"	.	.	.	.
copper	"	.	.	.	.
"	.	.	.	.	.
"	.	.	.	.	.
"	.	.	.	.	.

Multi Factor Parametric Experiment

Fig. 1

A	B	C	D	LP
1	1	1	1	1
1	1	1	2	2
1	1	2	1	3
1	1	2	2	4
1	1	3	1	5
1	1	3	2	6
1	2	1	1	7
1	2	1	2	8
1	2	2	1	9
1	2	2	2	10
1	2	3	1	11
1	2	3	2	12

(a)

A	C	D	B	LP
1	1	1	1	1
1	1	2	1	3
1	2	1	1	5
1	2	2	1	7
1	3	1	1	9
1	3	2	1	11
1	1	1	2	2
1	1	2	2	4
1	2	1	2	6
1	2	2	2	8
1	3	1	2	10
1	3	2	2	12

(b)

A	D	C	B	LP
1	1	1	1	1
1	2	1	1	7
1	1	2	1	3
1	2	2	1	9
1	1	3	1	5
1	2	3	1	11
1	1	1	2	2
1	2	1	2	8
1	1	2	2	4
1	2	2	2	10
1	1	3	2	6
1	2	3	2	12

(c)

$|A|=1$, $|B|=2$, $|C|=3$, and $|D|=2$.
LP=logical position.
(a) is configuration of original category attributes.
(b) is the result of ABCD -> ACDB.
(c) is the result of ABCD -> ADCB.

Fig. 2