Aggregation Algorithms for Very Large Compressed Data Warehouses

Jianzhong Li
Dept. of Computer Science
Heilongjiang University
P. R. China

Doron Rotem
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
USA

Jaideep Srivastava
Dept. of Computer Science
University of Minnesota
Minneapolis

Abstract

Many efficient algorithms to compute multidimensional aggregation and Cube for relational OLAP have been developed. However, to our knowledge, there is nothing to date in the literature on aggregation algorithms on compressed data warehouses for multidimensional OLAP. This paper presents a set of aggregation algorithms on very large compressed data warehouses for multidimensional OLAP. These algorithms operate directly on compressed datasets without the need to first decompress them. They are applicable to data warehouses that are compressed using variety of data compression methods. The algorithms have different performance behavior as a function of dataset parameters, sizes of outputs and main memory availability. The analysis and experimental results show that the algorithms have better performance than the traditional aggregation algorithms.

1. Introduction

Decision support systems are rapidly becoming a key to gaining competitive advantage for businesses. Many corporations are building decision-support databases, called *data warehouses*, from operational databases. Users of data warehouses typically carry out on-line analytical processing (OLAP) for decision making.

There are two kinds of data warehouses. One is for relational OLAP, called *ROLAP data warehouse* (RDW)[2,3,4]. The other one is for multidimensional OLAP, called *MOLAP data warehouses* (MDW) [5,6,7]. RDWs are built on top of standard relational database systems. MDWs are based on multidimensional database systems. A MDW is a set of *multidimensional datasets*. In a simple model, a multidimensional dataset in a MDW consists of *dimensions* and *measures*, represented by $R(D_1, D_2, ..., D_n; M_1, M_2, ..., M_k)$, where D_i's are dimensions and M_j's are measures.

The data structures in which RDWs and MDWs store datasets are fundamentally different. RDWs use relational tables as their data structure. That is, a "cell" in a logically multidimensional space is represented as a tuple with some attributes identifying the location of the cell in the multidimensional space and other attributes containing the values of the measures of the cell. By contrast, MDWs store their datasets as multidimensional arrays. MDWs only store the values of measures in a multidimensional space. The position of the measure values within the space can be calculated by the dimension values.

Multidimensional aggregation and Cube[1] are the most common operations for OLAP applications. The aggregation operation is used to "collapse" away some dimensions to obtain a more concise dataset, namely to classify items into groups and determine one value per group. The Cube operation computes multidimensional aggregations over all possible subsets of the specified dimensions.

Computing aggregation and the Cube are core operations on RDWs and MDWs. Methods of computing single aggregation and the Cube for RDWs have been well studied. In [11], a survey of the single aggregation algorithms for relational database systems is presented. In [1], some rules of thumb are given for an efficient

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

implementation of the Cube for RDWs. In [12] and [13], algorithms are presented for deciding what group-bys to pre-compute and indexing for RDWs. In [14] and [15] , a Cubetree storage organization for RDW aggregation views is presented. In [16] , fast algorithms for computing the Cube operator for RDWs are given. These algorithms extend sort-based and hash-based methods with several optimizations.

Aggregation pre-computing is quite common in statistical databases[17]. Research in this area has considered various aspects of the problem such as developing a model for aggregation computations[18], indexing pre-computed aggregations[19], and incrementally maintaining them[20].

While much work has been done on how to efficiently compute aggregation and the Cube for RDWs, to the best of our knowledge, there is only one published paper on how to compute the Cube for MDWs[10], and there is no published work on how to compute single multidimensional aggregation for MDWs.

MDWs present a different challenge in computing aggregation and the Cube. The main reason for this is the fundamental difference in physical organization of their data. The multidimensional data spaces in MDWs normally have very large size and a high degree of *sparsity*. That has made data compression a very important and successful tool in the management of MDWs. There are several reasons for the need of compression in MOLAP data warehouses. The first reason is that a multidimensional space created by the cross product of the values of the dimensions can be naturally sparse. For example, in an international trade dataset with dimensions exporting country, importing country, materials, year and month, and measure amount, only a small number of materials are exported from any given country to other countries. The second reason for compression is the need to compress the descriptors of the multidimensional space. Suppose that a multidimensional dataset is put into a relational database system. The dimensions organized in tabular form will create a repetition of the values of each dimension. In fact, in the extreme, but often realistic case that the full cross product is stored, the number of times that each value of a given dimension repeats is equal to the product of the cardinalities of the remaining dimensions. Other reasons for compression in MDWs come from the properties of the data values. Often the data values are skewed in some datasets, where there are a few large values and many small values. In some datasets, data values are large but close to each other. Also, sometimes certain values tend to appear repeatedly.

There are many data compression techniques applicable for MDWs [8,9]. A multidimensional dataset can be thought of as being organized as a multidimensional array with the values of dimensions as the indices of the array. The rearrangement of the rows and columns of the array can result in better clustering of the data into regions that are highly sparse or highly dense. Compression methods that take advantage of such clustering can thus become quite effective.

Computing multidimensional aggregation and Cube on compressed MDWs is a big challenge. Since most large MDWs must be compressed for storage, efficient multidimensional aggregation and Cube algorithms working directly on compressed data are important.

Our goal is to develop efficient algorithms to compute multidimensional aggregation and Cube for compressed MDWs. We concentrate on single multidimensional aggregation algorithms for compressed MDWs. This paper presents a set of multidimensional aggregation algorithms for very large compressed MDWs. These algorithms operate directly on compressed datasets without the need to first decompress them. They are applicable to a variety of data compression methods. The algorithms have different performance behavior as a function of dataset parameters, sizes of outputs and main memory availability. The algorithms are described and analyzed with respect to the I/O and CPU costs. A decision procedure to select the most efficient algorithm, given an aggregation request, is also given. The analysis and experimental results show that the algorithms compare favorably with previous algorithms.

The rest of the paper is organized as follows. Section 2 presents a method to compress MDWs. In section 3, description and analysis of the aggregation algorithms for compressed MDWs are given. Section 4 discusses the decision procedure that selects the most appropriate algorithm for a given aggregation request. The performance results are presented in section 5. Conclusions and future work are presented in section 6.

2. Compression of MDWs

This section presents a method to compress MDWs. Each dataset in a MDW is first stored in a multidimensional array to remove the need for storing the dimension values. Then, the array is transformed into a linearized array by an array linearization function. Finally, the linearized array is compressed by a mapping-complete compression method.

2.1 Multidimensional Arrays

Let $R(D_1, D_2, ..., D_n; M_1, M_2, ..., M_m)$ be an n-dimensional dataset with n dimensions, $D_1, D_2, ..., D_n$, and m measures, $M_1, M_2, ..., M_m$, where the cardinality of the i^{th} dimension is d_i for $1 \le i \le n$. Using the *multidimensional array method* to organize R, each of the m measures of R are first stored in a separate array. Each dimension of R is used to form one dimension of each of these n-

dimensional arrays. The dimension values of R are not stored at all. They are the indices of the arrays which are used to determine the position of the measure values in the arrays. Next, each of the n-dimensional arrays is mapped into a *linearized array* by an array linerization function.

Assume that the values of the i^{th} dimension of R is encoded into $\{0, 1, ..., d_i-1\}$ for $1 \le i \le n$. The *array linerization function* for the multidimensional arrays of R is

$$LINEAR(x_1, x_2, ..., x_n)=x_1d_2d_3...d_n+x_2d_3...d_n+\ldots+x_{n-1}d_n+x_n$$
$$=(\ldots(x_1d_2+x_2)d_3+\ldots)d_{n-2}+x_{n-2})d_{n-1}+x_{n-1})d_n+x_n.$$

In each of the m linearized arrays,the position where the measure value determined by array indices $(i_1, i_2, ..., i_n)$ is stored is denoted by $LINEAR(i_1, i_2, ..., i_n)$.

Let $[X]$ be the integer part of X. The *reverse array linerization function* of the multidimensional array of R is

$$R\text{-}LINEAR(Y) = (y_1, y_2, ..., y_n),$$

where, $y_n=Y \bmod d_n$, $y_i=[...[Y/d_n]...]/d_{i+1}] \bmod d_i$ for $2 \le i \le n-1$, $y_1=[[... [[Y/d_n]/d_{n-1}]...]/d_3]/d_2]$. For a position P in a linearized array, the dimension values $(i_1, i_2, ..., i_n)$ determining the measure value in position P, is $R\text{-}LINEAR(P)$.

2.2 Data Compression

The linearized arrays that store multidimensional datasets normally have high degree of sparsity and need to be compressed. It is desirable to develop techniques that can access the data in their compressed form and can perform logical operations directly on the compressed data. Such techniques (see [8]) usually provide two mappings. One is *forward mapping*, it computes the location in the compressed dataset given a position in the original dataset. The other one is *backward mapping*, it computes the position in the original dataset given a location in the compressed dataset.

A compression method is called *mapping-complete* if it provides forward mapping and backward mapping. Many compression techniques are mapping-complete, such as header compression [21] and chunk-offset compression [10]. The algorithms proposed in this paper are applicable to all the MDWs that are compressed by any mapping-complete compression method. To make the description of the algorithms more concrete, we assume that the datasets in the MDWs have been stored in a linearized array, each of which has been compressed using the header compression method[21].

The header compression method is used to suppress sequences of missing data codes, called *constants*, in linearized arrays by counts. It provides an efficient access to the compressed data by forward and backward mappings with I/O and CPU costs of $O(\log_2\log_2 S)$, where S is the size of the header, using interpolation search[22]. This method makes use of a *header* that is a vector of counts. The odd-positioned counts are for the unsuppressed sequences, and the even positioned counts are for suppressed sequences. Each count contains the cumulative number of values of one type at the point at which a series of that type switches to a series of the other. The counts reflect accumulation from the beginning of the linearized array to the switch points. In addition to the header file, the output of the compression method consists of a file of compressed data items, called the *physical file*. The original linearized array, which is not stored, is called the *logical file*. Figure 1 shows an example. In the figure, *LF* is the logical file, *0*'s are the suppressed constants, *v*'s are the unsuppressed values, *HF* is the header and *PF* is the physical file.

3. Multidimensional Aggregation Algorithms

In this section, we assume that datasets in MDWs are stored using the compressed multidimensional arrays method presented in section 2. Without loss of generality we assume that each dataset has only one measure.

Let $R(D_1, D_2, ..., D_n; M)$ be a multidimensional dataset. A *dimension order* of R, denoted by $D_{i_1}D_{i_2}...D_{i_n}$, is an order in which the measure values of R are stored in a linearized array by the array linearization function with D_{i_j} as the j^{th} dimension. Different dimension orders leads to different orders of the measure values in the linearized array. In the following discussion, we assume that R is stored initially in the order $D_1D_2...D_n$.

The input of an aggregation algorithm includes a dataset $R(D_1, D_2, ..., D_n; M)$, a *group-by dimension set* $\{A_1, A_2, ..., A_k\} \subseteq \{D_1, D_2, ..., D_n\}$ and an aggregation function F. The output of the algorithm is a dataset $S(A_1, A_2, ..., A_k; F(M))$, where the values of $F(M)$ are computed from the measure values of R by the aggregation function F. In the rest of the paper, we will use the following symbols for the relevant parameters:

d_i: the cardinality of the dimension D_i of R.

N: the number of data items in the compressed linearized array of R.

N_{oh}: the number of data items in the header of R.

N_r: the number of data items in the compressed linearized array of S.

H_{rh}: the number of data items in the header of S.

B: the number of data items of one memory buffer or one disk block.

3.1. Algorithm G-Aggregation

3.1.1 Description

G-Aggregation is a "general" algorithm in the sense that it can be used in all situations. The algorithm performs a multidimensional aggregation in two phases. In phase one, called *transposition phase*, it transposes the

LF: v_1 v_2 0 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
HF: 2 9 7 11 10 14
PF: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10}

Figure 1.

dimension order of the input multidimensional dataset R into a favorable dimension order so that the aggregation can be easily computed. For example, let $R(A, B, C, D; M)$ be a 4-dimensional dataset that is stored in a linearized 4-dimensional array in the dimension order *ABCD*. Assume that $\{B,C\}$ is the group-by dimension set. The dimension order *BCAD* and *BCDA* are favorable dimension orders for computing the aggregation with group-by dimension set $\{B,C\}$. In phase two, called *aggregation phase*, the algorithm computes the aggregation by one scan of the transposed R. Figure 2 illustrates the algorithm. For expository purposes, we use the relational form in Figure 2. In reality, the algorithm works directly on the compressed linearized array of R.

The *transposition phase* assumes that W buffers are available. Data from the compressed array (physical file) is read into the buffers. For each data item in a buffer, the following is done: (i) backward mapping is performed to obtain the logical position in the logical file, (ii) the dimension values of the item are recovered by the reverse array linearization function, and (iii) a new logical position of the item in the transposed space is computed using the array linearization function. This new logical position, called a "tag", is 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 to disk along with the tags. This process is repeated for the rest of the blocks in the physical file of R. The runs generated and their tags are next merged using the W buffers. A new header file is constructed for the transposed compressed array 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 the (shuffled) data items is the new transposed compressed linearized array. The *aggregation phase* scans the transposed array once, and aggregates the measure values for each combined values of the group-by dimensions one by one.

To transpose the compressed multidimensional array of R, G-Aggregation reads, writes and processes the run files (of the same size as that of the original compressed file) $\left\lceil \log_W \left\lceil \frac{N}{B} \right\rceil \right\rceil$ times in the transposition phase. To perform the final aggregation, another scan is needed. In each of the two phases, each of the original and transposed header files are read once. If the aggregation is performed as early as possible, the size of the run files will be reduced and the I/O and CPU costs will be reduced dramatically. To improve the algorithm, we perform aggregation and merge at the same time. With such "early" aggregation, run files will be smaller than the original file, and the cost for creating and reading the transposed header file is deleted.

The improved G-Aggregation assumes that $W+2$ buffers, each of size B, are available. One buffer is used for input and another for output. W buffers are used as aggregate and merge buffers, denoted by *buffer*[j] for $1 \leq j \leq W$. Let $R(D_1, D_2, \ldots, D_n; M)$ be the operand, and $\{A_1, A_2, \ldots, A_k\} \subseteq \{D_1, D_2, \ldots, D_n\}$ be the group-by dimension set.

The improved G-Aggregation also consists of two phases. The first phase generates the sorted runs of R in the order $A_1A_2 \ldots A_k$. Every value v in each run is a local aggregation result of a subset of R with an identification tuple of the group-by dimension values $(a_1, a_2, \ldots, a_k)$ as its tag. To generate a run, the algorithm reads as many blocks of the compressed linearized array of R as

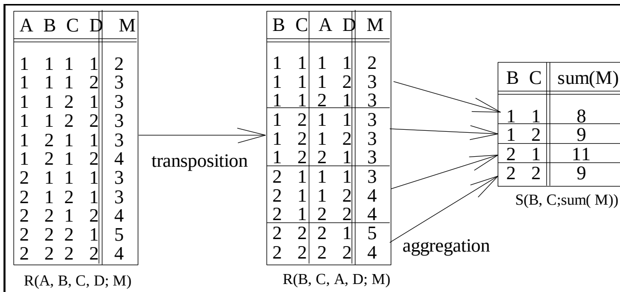


Figure 2.

possible, sorts them in the order $A_1A_2 \ldots A_k$, locally aggregates them and fills the W buffers with the locally aggregated results. For each *buffer*[j], the algorithm reads an unprocessed block of the compressed linearized array of R to the input buffer. For each data item v in the input buffer the following is done: (i) backward mapping is performed to obtain the logical position in the logical file; (ii) the dimension values $\{x_1, x_2, \ldots, x_n\}$ of v are recovered using the reverse array linearization function, and (iii) the values $\{a_1, a_2, \ldots, a_k\}$ of the group-by dimensions $\{A_1, A_2, \ldots, A_k\}$ (called a "tag") are selected from $\{x_1, x_2, \ldots, x_n\}$ and then stored with v in the input buffer. An internal sort is performed on the data items in the input buffer with respect to the tags of the data items. The sorted data items, each of which is in the form (v, *tag*) in the input buffer, are next locally aggregated and stored to *buffer*[j]. The process is repeated until *buffer*[j] is full. When all the W buffers are full, all the data items in the W buffers

are locally aggregated and merged in order of their tags, and written to disk to form a sorted run. The whole process is repeated until all runs are generated.

In the second phase, the sorted runs generated in phase one are aggregated and merged using W buffers. A new header file is constructed for the compressed array in the final pass of the aggregation and merge sequence, and the tags associated with the data items are discarded. The final compressed file produced is the compressed linearized array of the aggregation result. Figure 3 describes the main steps of the algorithm.

3.1.2 Analysis

We first analyze the I/O cost of G-Aggregation. In phase one, $\lceil N/B \rceil+(\lceil N_0/B \rceil-1)+\lceil N_{oh}/B \rceil$ disk block accesses are needed to read the original compressed linearized array of R, read the original header file and write the sorted runs to disk (the last block is kept in memory for use in the second phase). Here N_0 ($\leq N$) is the number of data items in all the runs generated in this phase. In phase two, $\log_W S$ passes of aggregation and merge are needed. Let N_I be the number of data items in the output of the I^{th} pass for 1? I? $\log_W S$. Obviously, $N_r=N_{\log_W S}$. A buffering scheme is used so that in the odd (even) pass, disk block reading is done from the last (first) block to the first (last) block. One block can be saved from reading and writing by keeping the first or last block in memory for use in the subsequent pass. In the last pass, we need to build and write the result header file. Thus,

$$\lceil N_r/B \rceil+(\lceil N_0/B \rceil-1)+\sum_{i=1}^{\lceil \log_W S \rceil-1} 2(\lceil N_i/B \rceil-1)+\lceil N_{rh}/B \rceil$$

disk accesses are required in the phase. In summary, the I/O cost of G-Aggregation is

Iocost(G-Aggregation)=$\lceil N/B \rceil+\lceil N_{oh}/B \rceil+\lceil N_r/B \rceil+\lceil N_{rh}/B \rceil+\sum_{i=0}^{\lceil \log_W S \rceil-1} 2(\lceil N_i/B \rceil-1)$.

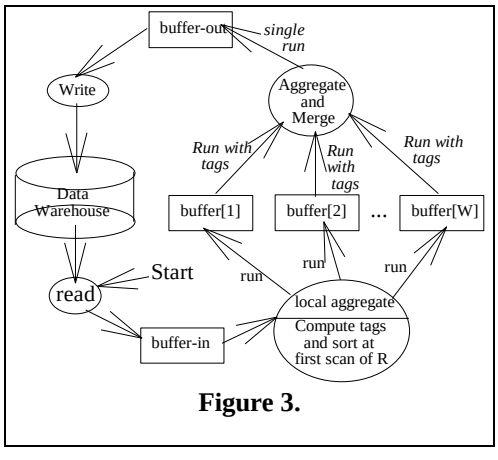


Figure 3.

From the algorithm, $N_r \leq N_0 \leq N$ and $N_r \leq N_I \leq N_{I-1}$. The average value of N_0 is $\frac{(N-N_r+1)(N+N_r)}{2(N-N_r+1)}=\frac{N+N_r}{2}$. The average value of N_I is $\frac{(N_{i-1}-N_r+1)(N_{i-1}+N_r)}{2(N_{i-1}-N_r+1)}=\frac{N_{i-1}+N_r}{2}$. Solving the recursive equation $N_I=\frac{N_{i-1}+N_r}{2}$, we have $N_I \leq \frac{1}{2^{i+1}}(N_r+N)+N_r$. Thus, on the average,

$$\sum_{i=0}^{\lceil \log_W S \rceil-1} 2(\lceil N_i/B \rceil-1) \leq \sum_{i=0}^{\lceil \log_W S \rceil-1} 2\frac{N_i}{B} \leq$$

$$2\sum_{i=0}^{\lceil \log_W S \rceil-1} (\frac{1}{2^{i+1}}\frac{N+N_r}{B}+\frac{N_r}{B}) \leq 2(\frac{N+N_r}{B}+\lceil \log_W S \rceil\frac{N_r}{B}).$$

Since $S \leq \left\lceil \frac{N}{BW} \right\rceil$, the average value of Iocost(G-Aggregation) is

AIOcost(G-Aggregation)=$O(\lceil N/B \rceil+\lceil N_{oh}/B \rceil+\lceil N_r/B \rceil+\lceil N_{rh}/B \rceil+2(\frac{N+N_r}{B}+\frac{N_r}{B}\left\lceil \log_W \left\lceil \frac{N}{BW} \right\rceil \right\rceil))$.

Next, we analyze the CPU cost of G-Aggregation. Let N_I be the same as above for $0 \leq I \leq \log_W S$. In the first phase, for each value in the compressed linearized array of R, we need to perform a backward mapping and a reverse array linearization. A backward mapping requires one computation because we scan the array and header from the beginning. A reverse array linearization operation requires $2(n-1)$ divisions and subtractions. Thus, $2N(n-1)+N$ computations are needed for the backward mapping and reverse array linearization in this phase. $N-N_0$ computations are needed for the local aggregations in this phase. There are also $\lceil N/B \rceil$ blocks, each with size B, to sort. To sort a block with size B requires $B\log_2 B$ computations. Thus, $\lceil N/B \rceil B\log_2 B$ computations are required to sort the $\lceil N/B \rceil$ blocks. The N_0 output data items of the first phase are generated by merging W buffers. Generating a data item requires at most $\log_2 W$ computations. Therefore, the total number of CPU operations for the first phase is $2Nn-N_0+\left\lceil \frac{N}{B} \right\rceil B\log_2 B+N_0\log_2 W$. In the second phase, the algorithm performs $\log_W S$ iterations. The I^{th} iteration involves the aggregating and merging of $\left\lceil \frac{S}{W^{i-1}} \right\rceil$ runs into $\left\lceil \frac{S}{W^i} \right\rceil$ and output N_I data items. In the I^{th} iteration, $N_{I-1}-N_I$ aggregations are needed. The N_I output data items are

generated by merging W buffers. Each data item requires at most $\log_2 W$ computations. In the final iteration, the N_r computations are needed to compute the result header counts. Therefore, the number of computations required by the second phase is

$$\sum_{i=1}^{\lceil \log_W S \rceil} ((N_{i-1} - N_i) + N_i \log_2 W) + N_r = N_0 + \sum_{i=1}^{\lceil \log_W S \rceil} N_i \log_2 W .$$

We can show that the the average value of CPUcost(G-Aggregation) is

$$\text{ACPUcost(G-Aggregation)} = O(2Nn + \left\lceil \frac{N}{B} \right\rceil B \log_2 B + (N + 2N_r + N_r \left\lceil \log_W \left\lceil \frac{N}{BW} \right\rceil \right\rceil) \log_2 W).$$

3.2 Algorithm M-Aggregation

3.2.1 Description

This algorithm is superior to G-Aggregation in case the aggregation result fits into memory. M-Aggregation computes aggregation by only one scan of the compressed linearized array of the operand dataset R. It reads blocks of the compressed linearized array of R one by one. For each data item v in the compressed linearized array of R, the following is done: (I) backward mapping is performed to obtain v's logical position; (ii) the dimension values of v, (x_1, x_2, …, x_d), are recovered by the reverse array linearization function from the logical position of v, and the values (a_1, a_2, …, a_k) of the group-by dimensions are selected from (x_1, x_2, …, x_d); (iii) if there is a w that is identified by (a_1, a_2, …, a_k) in the output buffer, aggregate v to w using aggregation function, otherwise insert v with (a_1, a_2, …, a_k) as a tag into the output buffer using hash method. Finally, the algorithm builds the new header file and writes the output buffer to the result file discarding the tags. M-Aggregation is described as follows.

3.2.2 Analysis

M-Aggregation requires one scan of the original compressed linearized array of R, and a writing of the resulting file. Also, the reading of the original header file and writing of the new header file are needed. The total I/O cost is

$$\text{IOcost(MAggregation)} = \lceil N/B \rceil + \lceil N_r/B \rceil + \lceil N_{oh}/B \rceil + \lceil N_{rh}/B \rceil.$$

The CPU cost of M-Aggregation is, for each data item in the compressed linearized array of R, the cost of performing a backward mapping, a reverse array linearization, a hashing computation, and an aggregation or memory operation (move data to output buffer), and the cost for computing the result header counts. As discussed in 3.1.2, a backward mapping requires only one computation. All the backward mappings for all data items in the compressed linearized array of R requires N computations. All the reverse array linearizations for all data items require $2N(n-1)$ computations. Steps (8) and (9) require N hash computations. Computing the result header counts requires N_r computations. The algorithm requires N-N_r aggregation and N_r memory operations also. Thus, CPU cost of the algorithm is at most

$$\text{CPUcost(M-Aggregation)} = 2Nn + Nh + N_r,$$

where h is the number of computations needed by a hashing computation.

3.3 Algorithm Prefix-Aggregation

3.3.1 Description

This algorithm takes advantage of the situation where

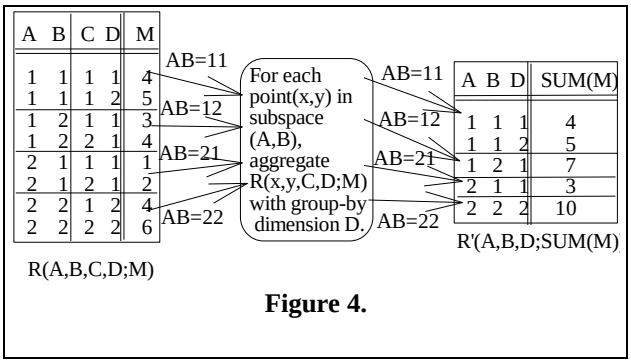


Figure 4.

the group-by dimension set contains a prefix of the dimension order $D_1D_2\ldots D_n$ of the operand dataset $R(D_1, ..., D_n; M)$. It performs aggregation in main memory by one scan of the compressed linearized array of R. It requires a memory buffer large enough to hold each portion of the resulting compressed linearized array for each "point" in the subspace composed by the prefix.

In rest of the paper, $R(D_1, ..., D_k, a_{k+1}, ..., a_{k+p}, D_{k+p+1}, ..., D_n; M)$ represents a subset of $R(D_1, ..., D_n; M)$ whose dimension values on $\{D_{k+1}, ..., D_{k+p}\}$ are $\{a_k, ..., a_{k+p}.\}$. We use an example to illustrate the algorithm. Assume that the operand dataset R has four dimensions A, B, C and D, and is stored in a compressed array in the order $ABCD$. Let us consider the aggregation with group-by dimension set $\{A, B, D\}$ that contains a prefix, AB, of the dimension order of R. Figure 4 shows an example of computing the aggregation with group-by dimension set $\{A, B, D\}$. For each "point" (a, b) in the subspace (A, B) of R, namely (1,1), (1,2), (2,1) or (2,2) in Figure 4, the algorithm performs the aggregation on $R(a, b, C, D; M)$ with D as the group-by dimension and appends to the result file. The new header counts is computed at the same time. This is the partial result of the aggregation under the fixed "point" (a, b). All partial results are

concatenated to form the final aggregation result. The reason is that the subspace (A, B) is stepped through in the same order as the original R, i.e., the rightmost index is varying the fastest. Prefix-Aggregation is as follows.

3.3.2 Analysis

Prefix-Aggregation requires the reading of the original compressed array of R, and writing of the resulting file. Also, the reading of the original header file and writing of the new header file are needed. The total I/O cost is

$$\text{IOcost(Prefix-Aggregation)}= \lceil N/B \rceil + \lceil N_r/B \rceil + \lceil N_{oh}/B \rceil + \lceil N_{rh}/B \rceil.$$

The CPU cost of Prefix-Aggregation is, for each data item in the compressed linearized array of R, the cost for performing a backward mapping, a reverse array linearization, a comparison (step (9)) and an aggregation or a memory operation (move data to output buffer), and the cost of computing new header counts. Thus, the CPU cost of Prefix-Aggregation is at most

$$\text{CPUcost(Prefix-Aggregation)}=N(2n+1)+N_r.$$

3.4 Algorithm Infix-Aggregation

3.4.1 Description

This algorithm takes advantage of the situation where the set of group-by dimensions is an infix of the dimension order $D_1D_2\ldots D_n$ of the operand $R(D_1, D_2, ..., D_n; M)$. Let $\{D_{t+1}, D_{t+2}, ..., D_{t+k}\}$ be the group-by dimensions, and D be the size of the subspace composed by D_1, D_2, ..., and D_t, denoted by $(D_1, D_2, ..., D_t)$. Obviously, $D_{t+1}D_{t+2}...D_{t+k}$ is an infix of the order $D_1D_2...D_n$. For each "point" $(a_1, ..., a_t)$ in $(D_1, ..., D_t)$, there is a sorted run, $R(a_1, ..., a_t, D_{t+1}, D_{t+2}, ..., D_n; M)$ in the order $D_{t+1}D_{t+2}...D_{t+k}$. $R(D_1, D_2, ..., D_n; M)$ is the connection of D such runs. The algorithm merges the D runs in order of $D_{t+1}D_{t+2}...D_{t+k}$ and perform aggregation at the same time. Figure 5 shows an example how the algorithm aggregates with group-by dimension set $\{C, D\}$ on $R(A, B, C, D, E; M)$. To perform the aggregation, R is first partitioned into 4 sorted runs, $R(1,1, C, D, E; M)$, $R(1,2, C, D, E; M)$, $R(2,1, C, D, E; M)$ and $R(2,2, C, D, E; M)$. Then all the runs are projected to $R(C, D; M)$ without removing repeated values. Finally, the projected runs are aggregated and merged to generate the aggregation result.

Infix-Aggregation assumes W buffers, each with size B, are available. If $W \geq D$, the algorithm becomes a main memory algorithm and requires only one scan of the compressed linearized array of R.

Infix-Aggregation is slower than Prefix-Aggregation when $W<D$ but not as memory intensive as Prefix-Aggregation. It requires $\log_W D$ passes to merge the D runs, where each pass merges W runs into one run. While the runs are merged, local aggregations are performed at the same time. When all the runs are merged into one run, the aggregation result is generated. In the algorithm, $R(a_1, a_2, ..., a_{t-1}, D_t, D_{t+1}, ..., D_{t+k}; M)$ represents the run for a "point" $(a_1, a_2, ..., a_t)$ in the subspace $(D_1, D_2, ..., D_t)$. The start position of the run $R(a_1, a_2, ..., a_t, D_{t+1}, D_{t+2}, ..., D_n; M)$ in the compressed linearized array of R can be computed by the following algorithm.

Using interpolation search[22] in step (2) and backward mapping, The I/O cost of the algorithm is at most $2\log_2\log_2 N_h$, and the CPU cost of the algorithms is at most $2\log_2\log_2 N_h + 4(n-1)$, where N_h is the number of data items in *header*.

Infix-Aggregation starts by first computing the start positions of the D runs in the compressed linearized array of R. Then, it computes the aggregation in $\log_W D$ iterations. In the first iteration, it partitions the D runs into $\lceil D/W \rceil$ groups, each with W runs, and aggregates and merges each group into one sorted run in the order $D_{t+1}D_{t+2}...D_{t+k}$. For the j^{th} group $(1 \leq j \leq \lceil D/W \rceil)$, the algorithm reads as many blocks of each run in the j^{th}

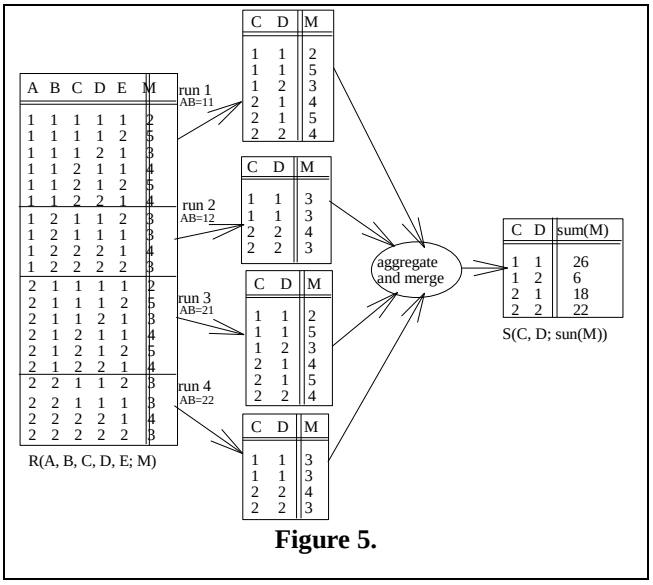


Figure 5.

group as possible, locally aggregates them by aggregation function F, and fills the local aggregation results in one of the W buffers. When all the W buffers are filled, the local aggregation results in the W buffers are aggregated and merged further and are appended to the j^{th} new run. The process is repeated until all the data items in all runs of the j^{th} group have been aggregated and merged into the j^{th} new run. After the first iteration, the D runs are merged into $\lceil D/W \rceil$ sorted runs in the order $D_{t+1}D_{t+2}...D_{t+k}$. In the following iteration i^{th} iteration in general, the algorithm partitions the $\left\lceil \frac{D}{W^{i-1}} \right\rceil$ runs produced in the $(i-1)^{th}$ iteration

into $\left\lceil \frac{D}{W^i} \right\rceil$ groups, each with W runs.

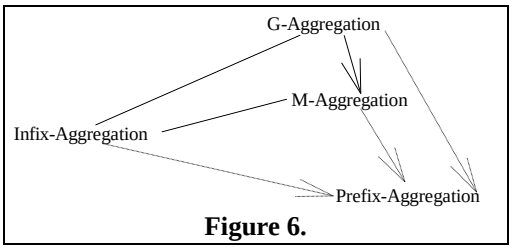


Figure 6.

Full analysis is omitted due to lack of space. We can show that the average value of CPUcost(Infix-Aggregation) is

$$\text{ACPUcost(Infix-Aggregation)}=O(2Nn+2D\log_2\log_2 N_{oh} +4D(n-1)+(N+N_r+N_r\lceil \log_W D \rceil)\log_2 W).$$

4. Comparisons of Algorithms and Selection Procedure

Let X and Y be two algorithms. We use $X \geq_{cost} Y$ to represent the fact that the total cost of X (I/O + CPU costs) is greater than or equal to the total cost of Y. Similarly $X \geq_{CPU} Y$ denotes that the CPU costs of X are larger than that of Y. From the analysis of the I/O and CPU costs of the algorithms proposed in the paper, we have the following observations. All the justifications of the observations are presented in [23].

Observation 1.

G-Aggregation$\geq_{cost}$Prefix-Aggregation,
G-Aggregation$\geq_{cost}$M-Aggregation ,
M-Aggregation$\geq_{cost}$Prefix-Aggregation,
and
Infix-Aggregation$\geq_{cost}$Prefix-Aggregation.

Observation 2.

If Infix-Aggregation$\geq_{CPU}$M-Aggregation,
then Infix-Aggregation$\geq_{cost}$M-Aggregation.

Observation 1 gives partial order of the algorithms in terms of I/O and CPU cost. According to the partial order, Prefix-Aggregation and M-Aggregation have better performance. However, these two algorithms require more memory. Further more, Prefix-Aggregation places special requirements on the group-by dimensions.

Figure 6 presents the order determined by observation 1. Each directed edge expresses a relation "$\geq_{cost}$". A dashed edge between two algorithms represents no cost domination relation can be determined between the two.

Below, a general decision procedure is given which is based on the order graph in Figure 8. In the procedure, a represents "the group-by dimension set contains a infix of the dimension order of the operand", β represents "the size of the aggregation result is not greater than the size of the available memory", γ presents " the group-by dimension set contains a prefix of the dimension order of the operand ", and η presents " available memory satisfies the requirement of Prefix-Aggregation ". Also A= Infix-Aggregation $\geq_{cost}$ G-Aggregation; **B**=Condition of Observation 2; **C**= Infix-Aggregation$\geq_{cost}$ M-Aggregation

5. Experimental Results

To examine the performance of the aggregation algorithms in practice, all the four aggregation algorithms have been implemented. The logical disk block size is 4k bytes.

To compare with the aggregation algorithms in relational database systems, we also implemented the sort and hash based traditional aggregation algorithms[11] in relational database systems. The experimental results show that our algorithms have much better performance than the traditional aggregation algorithms.

There are four factors that affect the performance of the aggregation algorithms. The *first one* is data density, namely the fraction of the cells in a multidimensional space actually containing valid data. The *second one* is compression ratio, this is affected by the number of dimensions and the size of the extra storage space required by compression methods. In the header compression method, the extra storage space is the header size. The *third one* is size of the available memory. The *last one* is dimension size, namely the number of elements in each dimension.

We conducted experiments to investigate the effect of the four factors on the performance of the algorithms. In the experiments, datasets were randomly generated, and stored using the compressed multidimensional array method for our algorithms and relational tables for the traditional aggregation algorithms. In each experiment, we randomly generated 10 aggregation operations, and then let each algorithm perform all the 10 operations, and we took the average execution time of the 10 operations as the final execution time of the algorithm. In the rest of this section, G, M, Infix and Prefix denote the G-Aggregation, M-Aggregation, Infix-Aggregation and Prefix-Aggregation. Sort and Hash denote the sort and hash based relational aggregation algorithms, and "$X>Y$" means "the execution time of algorithm X is greater than that of Y".

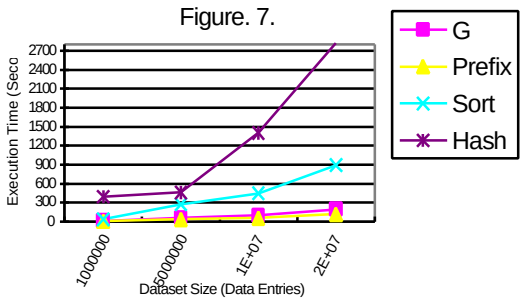
5.1. Performance Related to Number of Valid Data Entries

In these experiments, the benchmark dataset scheme consists of 15 dimensions and one measure. The data types of all dimensions are 4-byte integer. The data type of the measure is 4-byte float number. We randomly generated 4 versions of the benchmark with 1,000,000, 5,000,000, 10,000,000 and 20,000,000 valid data entries. The header size of each dataset is 50% of the dataset size. The aggregation result size of each dataset is 20% of the dataset size. Since M, Infix and Prefix have special requirements on aggregation dimensions and memory size, five sets of experiments were conducted.

In the first set of experiments, available memory size is fixed at 640K bytes. The memory size and the aggregation operations performed in this set of experiments satisfy the requirements of Prefix. Figure 7 presents the execution times of the algorithms while the number of data entries varies from 1,000,000 to 20,000,000. The figures indicate that Hash>Sort>G >Prefix, namely Prefix is the fastest algorithm and Hash is the slowest one. The figures also show that the larger the

dataset size is the larger the ratio of the execution times of Sort and Hash to the execution time of G or Prefix. The reason is that the I/O cost of Sort and Hash increases much faster than that of G and Prefix when the operand

Figure. 7.



dataset size increases. In the figures, we also see that all the execution times have a big jump at the data entry number 5,000,000. It is because that the available memory size can hold the whole aggregation result when the dataset size smaller than 5,000,000.

In the second set of experiments, available memory size is fixed at 640K bytes for G, Infix and Sort, and the aggregation operations performed satisfy the requirements of Infix. In order to get the aggregation results in accept time using Hash algorithm, the available memory size is set to 20% of the aggregation result size for Hash. Figure 8 presents the execution times of the algorithms while the data entry number varies from 1,000,000 to 20,000,000. The figures indicate that Hash>Sort>G >Infix.

In the third set of experiments, available memory size is the size of the maximum aggregation result, 125M bytes, namely 20% of the maximum dataset size 20,000,000, so that all the aggregation results fit in memory. Figure 9 presents the experiment results. It indicates that all the execution times are smaller than the first and second sets of experiments. The reason is that large available memory makes all algorithms faster. The figure also shows Sort>Hash>G >M when the data entry number is greater than 5,000,000. In the figures, we see that the execution times of G are smaller than the

Figure. 8.

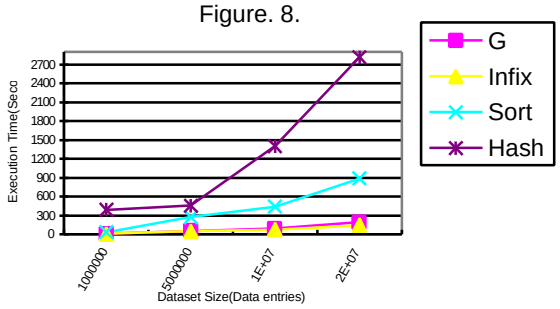


Figure. 9.

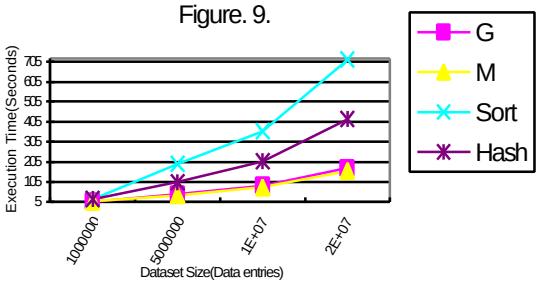
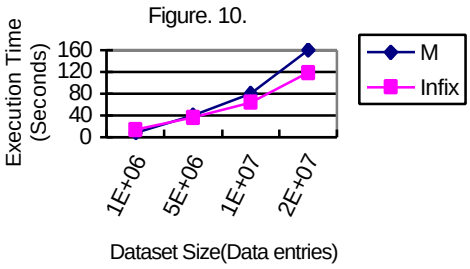


Figure. 10.

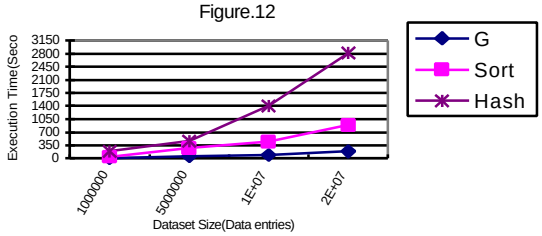


execution times of M when the data entry number is smaller than 5,000,000. It is because that when the dataset fit in memory M spends more CPU time for hashing computation.

The fourth set of experiments is to study the performance of the algorithms M, Infix and Prefix in case of the aggregation results fitting in memory. The parameters are the same as in the third set experiments. Figure 10 presents the comparisons of M and Infix, and figure 11 presents the comparison of M and Prefix.

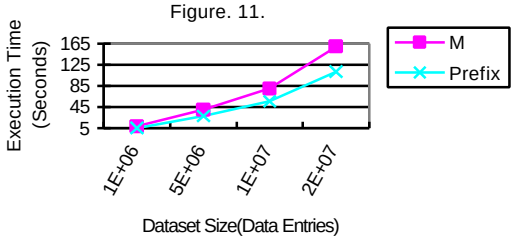
The fifth set of experiments is to compare the performance of the algorithms G, Sort and Hash without

Figure.12



any restriction. The available memory size was fixed at 640K. The experiment results in Figure 12 show Hash>Sort>G.

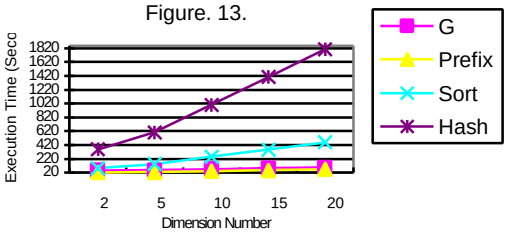
Figure. 11.



5.2. Performance Related to Data Compression Ratio

We first conducted experiments to study the performance of the algorithms while the dataset size is fixed and the dimension number, which has great effect on the compression ratio, varies. For the experiments, the dimension number of the operand datasets was varied from 2 to 20, the number of data entries was fixed at 10,000,000 data entries, each with 64 bytes, and the aggregation result size was kept at 2,000,000 data entries. Similar to section 5.1, we conducted five sets of experiments to meet the requirements of Prefix, Infix and

Figure. 13.



M. In the first two sets of experiments, the available memory size is fixed at 12.5M bytes.

In the first set of the experiments, the aggregation operations performed satisfy the requirements of Prefix. Figure 13 presents the experiment results. This figure shows that the execution times of G and Prefix increase very slowly while the dimension number increases. The reason is that the dimension number of a dataset does not effect the size of the compressed array storing the dataset so that the I/O costs of G and Prefix vary very small when the number of the dimensions increases. On the opposite, when the dimension number of a dataset increases, the size of the relational table storing the dataset increases. Thus, the I/O costs of Hash and Sort increase very fast with the increasing of the dimension number.

In the second set of experiments, the aggregation operations performed satisfy the requirements of Infix. Figure 14 presents the experiment results. This figure shows that the execution times of Sort and Hash still increase much faster than Infix and G with the same reason in the first set of experiments.

In the third set of experiments, To meet the requirement of M, memory size is fixed at 125M bytes to hold the aggregation result. Figure 15 presents the experiment results. The figure shows that the execution times of Sort and Hash increase much faster than M and

Figure. 14.

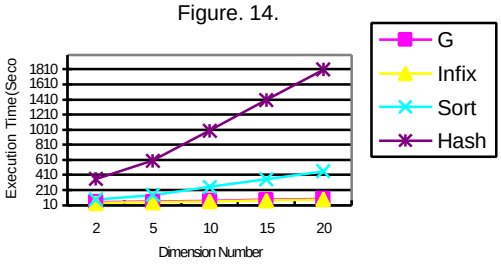
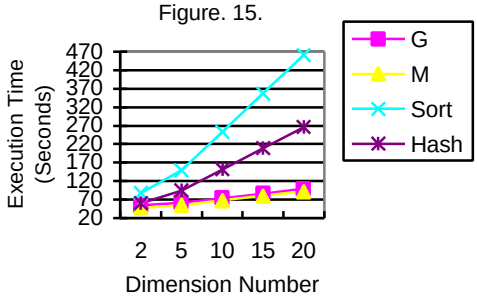


Figure. 15.



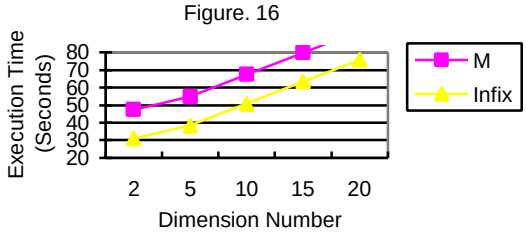
G, the performance of M is only a little better than G in case of aggregation results fitting in memory, and Hash is fast than Sort when available memory size is large enough.

The fourth set of experiments is to study the performance of the algorithms M, Infix and Prefix in case of the aggregation results fitting in memory. The parameters are the same as in the third set experiments. Figure 16 presents the comparison of M and Infix, and figure 17 presents the comparison of M and Prefix.

In the fifth set of experiments, memory size is fixed at 12.5M bytes. Figure 18 presents the experiment results. The figure indicates that the execution times of G increase much slowly than Sort and Hash.

Similar to the above four sets of experiments, we also conducted five sets of experiments to study the performance of the aggregation algorithms while the header size, which also has effect on compression ratio, varies. The experiment results show that the execution times of Sort and Hash kept at the same while the header size increases because that the header size has no effect on the relational table, and the header size has little effect on the performance of G, M, Prefix and Infix. The details of the experiments see [23].

Figure. 16



6. Summary and Conclusion

In this paper, a collection of aggregation algorithms was described and analyzed. These algorithms operate directly on compressed datasets in MDWs without the need to first decompress them. The algorithms are applicable to MDWs that are compressed using mapping complete compression methods. A decision procedure is also given to select the most efficient algorithm based on aggregation request, available memory, as well as the dataset parameters. The analysis and experimental results show that the algorithms have better performance than the traditional aggregation algorithms.

In conclusion, direct manipulation of compressed data is an important tool for managing very large data warehouses. Aggregation is just one (and important) such operation in this direction. Additional algorithms will be needed for OLAP operators on compressed multidimensional data warehouses. We are currently working on other operators such as searching, Cube, and other higher level OLAP operators on compressed MDWs.

Acknowledgement

This work was funded by the Department of Energy, Office of Energy Research, Office of Computational and Technology Research, Division of Mathematical Information and Computational Sciences under contract DE-AC03-76SF00098.

References

[1]. Gray, J., Chaudhuri, S., Bosworth, A., et al, "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-tables and Sub-totals", Data Mining and Knowledge Discovery, Vol.1, No.1, 29-53, 1997.

[2] Yazdani, S. and Wong, S., "Data Warehousing with Oracle", Prentice-Hall, Upper Saddle River, N.J.,

1997.

[3] Gupta, V. R., "Data Warehousing with MS SQL Server Unleashed", Sams, Englewood Cliffs, N. J., 1977.

[4] Chatziantonian, D. and Ross, K, "Querying Multiple Features in Relational Databases", Proc. of VLDB, 1996.

[5] Arbor Sofware, "The Role of Multidimensional Database in a Data Warehousing Solution", White Paper, Arbor Software, http://www. arborsoft.com/papers/ware TOC.html

[6] Inmon, W.H., "Multidimensional Databases and Data Warehousing", Data Management Riview, Feb. 1995.

[7] Colliat, G., "OLAP, Relational and Multidimensional Databases Systems", SIGMOD Record, Vol.25, No.3, 1996.

[8] Bassiouni, M.A., "Data Compression in Scientific and Statistical Databases", IEEE Transactions on Software Engineering, Vol. SE-11, No. 10, 1985.

[9] Roth, M. A. and Van Horn, S.J., "Database Compression", SIGMOD RECORD, Vol.22, No.3, 1993.

[10] Zhao, Y., Deshpande, P.M., and Naughton, J.F., "An Array-Based Algorithm for Simultaneous Multidimensional Aggregations", In Proc. of the ACM-SIGMOD Conference, 1997.

[11] Graefe, G., "Query Evaluation Techniques for Large Databases", ACM Computing Surveys, 25(2), 1993.

[12] Harinarayan, V., Rajaraman, A., and Ullman, J. D., "Implementing Data Cube Efficienly", In Proc. of the 1996 ACM-SIGMOD Coference, 1996.

[13] Gupta, H., Harinarayan, V., Rajaraman, A., and Ullman, J.D, "Index Selection for OLAP", In Proc. of the International Conference on Data Engineering, Binghamton, UK, April, 1997.

[14] Kotidis, Y. and Roussopoulos, N., "An Alternative Storage Organization for ROLAP Aggregation Views Based on Cubtrees", In Proc. of ACM-SIGMOD Conference, pp. 249-258, 1998.

[15] Roussopoulos, N., Kotidis, Y., and Roussopoulos, M, "Cubtree: Organization of and Bulk Incremental Updates on the Data Cube", In Proc. of ACM-SIGMOD Conference, pp. 89-99, 1997.

[16] Agarwal, S, Agrawal, R., Deshpande, P.M., et al, "On the Computation of Multidimensional Aggregations", In Proc. of the 22nd VLDB Conference, India, 1996.

[17] Shoshani, A., "Statistical Databases: Characteristics, Problems and Some Solutions", In Proc. Of the Eighth International Conference on Very Large Data Bases (VLDB), Mexico City, Mexico, 1982.

[18] Chen, M.C. and McNamee, L.P., "The Data Model and Access Method of Summary Data Management", IEEE Transactions on Knowledge and Data Engineering, 1(4), 1989.

[19] Srivastava, J., Tan, J.S.E., and Lum, V.Y., "TBSAM: An Access Method for Efficient Processing of Statistical Queries", IEEE Transactions on Knowledge and Data Engineering, 1(4), 1989.

[20] Michalewicz, Z., ed. "Statistical and Scientific Databases", Ellis Horwood, 1992.

[21] Eggers, S. and Shoshani, A., "Efficient Access of Compressed Data", In Proc. 1980 International Conference on Very Large Data Bases (VLDB), Montreal, Canada, Sept, 1980.

[22] Li, Jianzhong, Wang, H.K., Rotem, D. "Batched International Searching on Databases", Information Sciences (An International Journal). 1989. Vol.48.

[23] Li, Jianzhong, Rotem, Doron, and Srivastava, Jaideep, "Aggregation Algorithms for Very Large Compressed Data Warehouses", Technique Report, http://www.lbl.gov/~rotem/paper/aggr.ps.

Figure. 17.

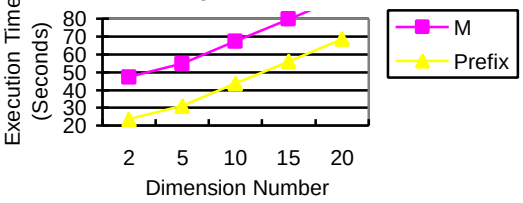


Figure. 18